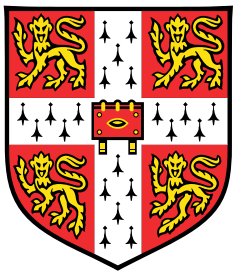
# Model-Architecture Co-design of Deep Neural Networks for Embedded Systems



**Partha Prasun Maji**

Department of Computer Science and Technology
University of Cambridge

This dissertation is submitted for the degree of
*Doctor of Philosophy*

Clare Hall June 2020

# Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. It is not substantially the same as any that I have submitted, or am concurrently submitting, for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or is being concurrently submitted, for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. This dissertation does not exceed the prescribed limit of 60 000 words.

Partha Prasun Maji
June 2020

# Abstract

In deep learning, a convolutional neural network (ConvNet or CNN) is a powerful tool for building interesting embedded applications that use data to make predictions. An application running on an embedded system typically has limited access to memory resources, processing power, and storage. Implementing deep convolutional neural network based inference on resource constrained devices can be very challenging, as these environments cannot usually make use of the massive compute power and storage that are present in cloud server environments. Furthermore, the constantly evolving nature of modern deep network architecture aggravates the problem by making it necessary to balance flexibility against specialisation to avoid the inability to adapt. However, much of the baseline architecture of a deep convolutional neural network stayed the same. With careful optimisation of the most commonly and widely occurring layer architectures, it is typically possible to accelerate these emerging workloads for resource-constrained embedded systems.

This thesis makes four contributions. I first developed a lossy three-stage low-rank approximation scheme that can reduce the computational complexity of a pre-trained model by $3-5\times$ and up to $8-9\times$ for individual convolutional layers. This scheme requires restructuring of the convolutional layers and generally suits the scenario where both the training data and trained model are available.

In many scenarios, the training data is not available for fine-tuning any loss in prediction accuracy if structural changes are made to a model as a post-processing step. Besides the lack of availability of training data, there are other situations where the architecture of a model cannot be changed after training. My second contribution handles this scenario by using a low-level optimisation scheme that requires no changes to the model architecture, unlike the low-rank approximation scheme. This novel scheme uses a modified version of the Cook-Toom algorithm to reduce the computational intensity of commonly occurring dense and spatial convolutional layers and speedup inference time by $2-4\times$.

My third contribution is an efficient implementation of the Cook-Toom class of algorithms on ubiquitous Arm's low-power Cortex processor. Unlike the direct convolution, computing convolutions using the modified Cook-Toom algorithm requires a different data processing pipeline as it involves pre- and post-transformations of the intermediate activa-

tions. I introduced a multi-channel multi-region (MCMR) scheme to enable an efficient implementation of the fast Cook-Toom algorithm. I demonstrate that by effectively using SIMD instructions and the MCMR scheme an average $2-3\times$ and a peak $4\times$ per layer speedup is easily achievable.

My final contribution is the Cook-Toom accelerator, a custom hardware architecture for modern convolutional neural network. This accelerator architecture is designed from the ground up to address some of the limitations of a resource constrained SIMD processor. I also illustrate how new emerging layer types can be mapped efficiently to the same flexible architecture without any modification.

# Acknowledgements

I'd like to start by thanking Robert Mullins, who has been an incredibly encouraging and active supervisor throughout this entire experience. His guidance through ideas of experiments and discussions has been crucial. I'd also like to thank my secondary advisor Simon Moore, for his pragmatic suggestions, and Sean Holden from the AI research group, for helping me through the process of finding the right research question. I'd like to thank a few other members of the Computer Laboratory, especially, Alex Chadwick, Yousun Ko and Daniel Bates for always being there to brainstorm novel ideas. I'd like to thank the Engineering and Physical Sciences Research Council (EPSRC) for funding my research and the opportunity they gave me to travel to conferences around Europe.

A lot of interesting research questions and ideas I learned during my short stint in Arm Ltd., for which I am incredibly grateful. Rune Holm has played an absolutely invaluable role as a mentor while I was in Arm. I'd also like to thank Andrew Mundy, for collaborating with me on the Arm implementation work, Ganesh Dasika, David Mansell, and Matthew Mattina for reviewing our research and providing guidance.

This thesis is indebted to Roshin, my partner, who I met at the University of Edinburgh. Thank you for being there for me during difficult times and providing all the encouragement. Thanks also go out to your family for having welcomed me into their fold. A ton of thanks go out to my family, particularly my father, mother and brother for having believed in me and for putting up with my insistence on exploring the unknown instead of settling down. Without your groundwork much early in my academic life, this PhD simply would not have been possible.

Finally, I'd like to thank all of my ex-teachers, family and friends for being there throughout the past few years of the PhD, and before.

# Table of contents

# List of figures

# List of tables

# Chapter 1

# Introduction

This dissertation describes a collection of co-optimisation schemes that help to reduce the overall inference time of a deep convolutional neural network. The schemes aim to exploit co-optimisation of model, algorithm and underlying hardware structure.

In this chapter I outline the motivation for this work. I also summarise the contributions of the dissertation and list the contents of the following chapters that describe the contributions in detail.

## 1.1 Motivation

Recent advances in deep learning have greatly changed the way that computing devices process human-centric contents such as images, videos, speech, and audio [61]. At the same time, the proliferation of mobile and embedded devices leads to visions of the internet of things (IoT), giving rise to a sensor-rich world where physical things in our everyday environment are increasingly enriched with computing [83]. Indeed, in recent years significant research efforts have been spent toward building cloud-based infrastructure to enable emerging deep-learning-based applications for edge devices [98]. The agility of cloud computing is great - but it simply isn't sufficient. In the near future there will be more demand for machine learning at the edge than in the cloud. As people need to interact with their digitally-assisted technologies (e.g. personal assistants, wearables, autonomous car, healthcare, and other smart IoT devices) in real-time, waiting on a datacentre many miles away isn't going to work. Not only the latency matters, but often these edge devices are not within the range of the cloud needing them to operate autonomously for the most part. Even when these devices are connected in the cloud, moving high-volumes of data to the centralised datacentre is not scalable, due to communication costs that impact performance and energy consumption [87]. Since the latency and security risk of relying on the cloud are intolerable, we need a signifi-

cant portion of computation closer to the edge to permit secure, autonomous, and real-time decision making. This poses an enormous challenge in terms of implementing emerging machine-learning workloads on resource-constrained low-power embedded devices. When it comes to image and video, the performance of many modern embedded applications is enhanced by application of neural networks, and more specifically by convolutional neural networks or ConvNets. Furthermore, with continuous progress made in research in the area of deep learning, the architecture of modern ConvNet is still evolving rapidly. As a consequence, many of the earlier results from model or hardware optimisation research became ineffective in the long run [35, 12, 93, 18, 6]. Over specialising during implementation often leads to loss of generalisation. For example, deep neural network with 6-bit implementations are efficient for some specific cases, but do not generalise well for medium to large datasets. Finding solutions that generalise well for the majority of the application datasets and deep models is extremely difficult. Over the last few years, we have seen a proliferation of innovative and more efficient ConvNet architectures. Although there has been a lot of research done either on model optimisation and hardware design in isolation, little attention has been paid to the model-algorithm-hardware co-design approach. As a result, the research in this space is quite fragmented and the outcome is not always the optimum. The aim of this dissertation is to fill this gap and investigate if emerging compute-heavy deep convolutional networks can benefit from co-design of model architecture, low-level algorithms and underlying hardware architecture.

## 1.2 Research questions

- Considering the rapidly evolving architecture of convolutional neural network and strong demands of such emerging architectures in industry, is it possible to find and accelerate the most commonly occurring layer structures within widely used models?
- How can the co-design of model, low-level algorithm and underlying hardware help in effectively harnessing the full potential of optimisation at the algorithmic level?
- Is it possible to combine lossy and lossless compression schemes to achieve a more efficient implementation compared to applying them in isolation?
- Building a new custom accelerator is an expensive solution and is not always practical due to rapidly changing characteristics of ConvNets. Arm processor-based solutions are ubiquitous across embedded applications and billions of such platforms are already in use. Is it feasible to implement a modern compute-heavy deep ConvNet on an existing mobile platform such as in Arm?

- Finally, considering the architecture of deep networks is still evolving, is it possible to build an efficient custom accelerator for modern ConvNets that is configurable, scalable and flexible enough to support a wide variety of emerging layer types?

## 1.3 Hypothesis

Model-algorithm-architecture co-design of deep convolutional neural networks can produce highly optimised implementations of inference for embedded applications that meet the limited compute, memory and power budgets of the target hardware. By aiming at the most commonly occurring layer architectures within widely used models, and applying such co-optimisations, we can enable the deployment of sophisticated deep-learning models on cutting-edge mobile and embedded systems.

## 1.4 Contributions

This dissertation provides four main contributions to research in the area of efficient deep-neural-network implementation on embedded systems.

First, an easy to implement three-step approximation scheme which can be applied on the commonly occurring spatial convolutional layers in many state-of-the-art pre-trained deep models in use today. The novel ADaPT approximation scheme is mathematically well grounded, easily reproducible and can be applied on ConvNets statically. This has resulted in a publication that illustrates the effectiveness of this scheme primarily in the IoT and embedded application space [68].

Second, a lossless scheme for computing convolution that reduces the total number of required multiplications. The novel scheme uses a modified version of the Cook-Toom algorithm and is very effective on small size filters which are very common in modern ConvNets. Furthermore, the lossless scheme does not require any structural changes to the pre-trained model, unlike the ADaPT approximation scheme. For this particular reason, the Cook-Toom class of algorithm are highly suitable for environments where training data is not available. This work was shortlisted for the best paper award in the ICANN conference organised by the European Neural Network Society (ENNS) [69].

Third, an efficient implementation of the Cook-Toom class of algorithms on one of Arm's ubiquitous low-power cortex processors. Unlike direct convolution, computing convolutions using the modified Cook-Toom algorithm requires a different data-processing pipeline. I introduce a multi-channel multi-region (MCMR) scheme to enable an efficient implementation of the fast Cook-Toom algorithm. The novel MCMR scheme can also be extended to other

similar resource-constrained SIMD processors. The speed up achieved from the MCMR scheme is significant for embedded systems and has been adopted by Arm for their Cortex class of processors. This work resulted in a joint publication with Arm holdings [71].

Fourth, a custom architecture for accelerating convolutional neural network using the Cook-Toom class of fast arithmetic, that addresses some of the limitations of a traditional general-purpose processor. The novel architecture is designed from the ground up based on a few fundamental design principles. The accelerator is designed based on the previously introduced MCMR scheme and it uses a core Hadamard engine. The accelerator is configurable, scalable and handles the most commonly occurring spatial convolutional layers efficiently.

## 1.5 Outline

I now outline the organisation of the rest of this dissertation.

In Chapter 2, I provide all the necessary technical background on convolutional neural-network architectures that is required to understand the rest of the thesis. I also highlight various challenges involved when implementing modern neural networks on low-power embedded systems. In this chapter, I also survey previous research work on model optimisation and hardware accelerators.

In Chapter 3, I explore redundancies in modern deep networks. I show how these redundancies can be exploited to compress deep models significantly. I introduced the concept of separable filters that builds up the background for the ADaPT approximation algorithm. Finally, I illustrate how layers in deep convolutional neural networks can be decomposed and approximated across two consecutive stages to reduce overall compute complexities of the model.

In Chapter 4, I focus on a lossless scheme to speed up convolution. Convolution with small filter size is the fundamental building block of modern convolutional neural networks and thus speeding up such convolution will improve the overall performance of the model. I introduce the Cook-Toom fast arithmetic scheme that can be used in modern convolutional neural network to reduce the total number of multiplications. I then extend the base version with a modified version and illustrate many variants of the algorithm that can help to accelerate modern convolutional neural networks.

In Chapter 5, I present an implementation of the modified Cook-Toom fast convolution algorithm on a widely used low power Arm mobile processor. I then introduce a novel multi-channel multi-region (MCMR) scheme that enables efficient implementation of the algorithm on Arm's SIMD architecture. By implementing several widely used ConvNets I illustrate the effectiveness of the MCMR scheme for SIMD processors. I conclude this

chapter discussing several limitations of implementing the modified Cook-Toom algorithm on a resource-constrained mobile processor that motivates the need for a custom accelerator.

In Chapter 6, I tackle the limitations discussed in the previous chapter by introducing a novel Cook-Toom custom accelerator. I start by introducing a number of design principles for domain specific accelerator design. I then describe the details of the Cook-Toom accelerator and the core Hadamard engine. I show a few example passes on how to map convolutional layers to the core Hadamard engine. In the evaluation section, I illustrate the effectiveness of the custom accelerator through benchmarking a number of widely used ConvNets today.

In Chapter 7, I conclude and provide some suggestions for further research on model/hardware co-design of deep neural networks.

## 1.6 Publications

Some of the research described in this dissertation also appears in the following publications:

- Partha Maji, Daniel Bates, Alex Chadwick, and Robert Mullins. ADaPT: Optimizing CNN inference on IoT and mobile devices using approximately separable 1-D kernels. In *the Proceedings of the ACM International Conference on Internet of Things and Machine Learning, ACM-IML, 2017* [68].

- Partha Maji, and Robert Mullins. 1D-FALCON: Accelerating deep convolutional neural network inference by co-optimization of models and underlying arithmetic implementation. In *the Proceedings of the International Conference on Artificial Neural Networks, ICANN, 2017* [69].

- Partha Maji, and Robert Mullins. On the Reduction of Computational Complexity of Deep Convolutional Neural Networks. In *the Proceedings of the Journal Entropy, 2018* [70].

- Partha Maji, Andrew Mundy, Jesse Beu, Ganesh Dasika, Matthew Mattina, and Robert Mullins. Efficient Winograd or Cook-Toom Convolution Kernel Implementation on Widely Used Mobile CPUs. In *the Proceedings of the Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications, IEEE HPCA.EMC2, 2019* [71]

# Chapter 2

# Background and related work

This chapter focuses on the background literature of deep convolutional neural networks and a survey of prior research on the optimisation of modern deep networks. I begin by introducing the detailed architecture of a convolutional neural network or ConvNet. I then illustrate various types of convolutional layers widely used today. This section is followed by an overview section on different data-reuse patterns which builds up a strong background for the chapters on hardware implementation. I end this chapter with a brief survey on prior work on model optimisation and hardware acceleration.

## 2.1 Deep convolutional neural network architecture

ConvNets are generally used in computer-vision tasks. However they've shown promising results when applied to various other tasks as well, including various natural-language-processing tasks, text based sentiment analysis, generating time-series data, speech emotion detection, audio classification [54, 53, 26, 74, 27, 90]. More and more diverse and interesting use cases are being found for ConvNet architectures [32, 91, 45]. A typical convolutional neural network (ConvNet) architecture consists of a series of stages. The first few stages are composed mostly of two types of layers: convolutional layers and pooling layers. As illustrated in Figure 2.1 there are a number of feature maps within a convolutional layer, within which each output feature map is connected to local patches in the feature maps of the previous layer through a set of weights called a filter (or a kernel) bank. The output of this weighted sum is then passed through a non-linearity layer. In most ConvNets, the non-linearity layer is a rectifier linear unit or, in short, ReLU. The role of the convolutional layer is to detect local conjunctions of features in a layer, whereas the role of the pooling or sub-sampling layer is to merge semantically similar features into one. Pooling also helps in reduction of feature size in the subsequent layers. The convolutional and pooling

layer combinations are then followed by fully-connected layers as shown in the figure. The choice of depth or number of layers of each type is one of the widely studied ConvNet design parameters. In the ImageNet competition, the accuracy of the classification task has improved year-over-year using deeper networks. For example, in 2012, AlexNet [56] achieved a top-5 error rate of 16.4% using 8 layers, whereas, in 2015, Microsoft's ResNet [44] achieved a staggering error rate of just 3.57% using 152 layers. The choice of connections across multiple layers is also an important area of model architecture research. For example, the ResNet architecture introduced bypass connections which skip over multiple layers. The ResNet model achieves a 2% improvement over the baseline architecture on top-5 accuracy. The ConvNet architecture design space is still evolving rapidly.

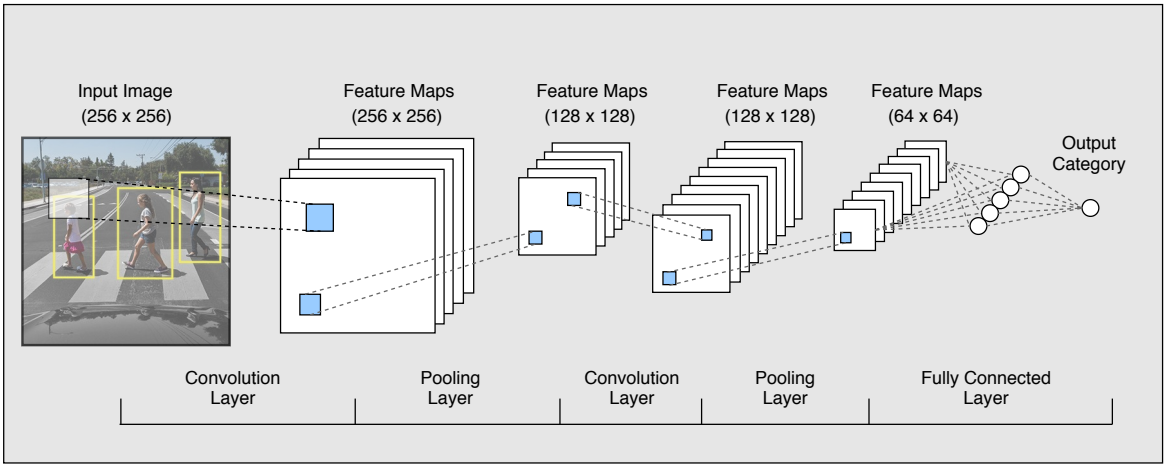



Fig. 2.1 A generic deep convolutional neural network (ConvNet)

Convolution and the convolutional layer are the major building blocks used in modern deep ConvNets and they are extremely compute heavy. Many modern deep ConvNets often require billions of FLOPs to compute a single frame of image. In this thesis, I am primarily interested at optimisation and efficient implementation of the convolutional layers. Figure 2.2 illustrates a typical convolutional layer in deep ConvNets. The example layer consists of $M$ 3-dimensional $N \times K \times K$ kernels and $N \times H \times L$ input feature maps. Each kernel performs a convolution on the input maps with a sliding stride of S, which generates a $R \times C$ output map. Since there are M sets of kernels $M$ output maps are produced. The computation of a convolutional layer can be expressed as the multi-layer loop shown in listing 2.1, where $X$, $W$ and $Out$ are the input maps, kernels and the output maps.

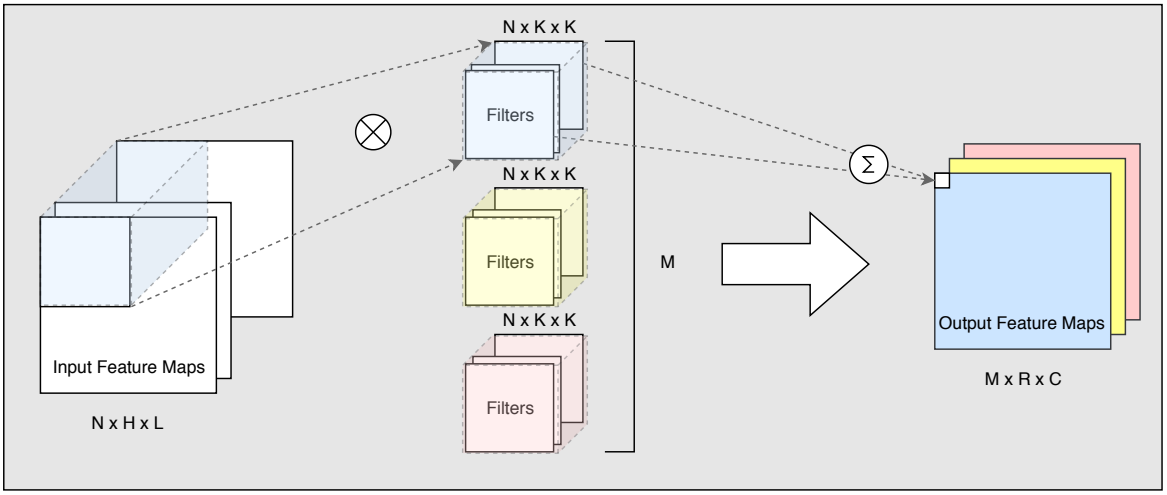



Fig. 2.2 A generic convolutional layer

```
for (r = 0; r < R; r++) //Loop R
  for (c = 0; c < C; c++) //Loop C
    for (n = 0; n < N; n++) //Loop N
      for (m = 0; m < M; m++) //Loop M
        for (i = 0; i < K; i++) //Loop K
          for (j = 0; j < K; j++) //Loop K
            Out[m][r][c] += W[m][n][i][j]*X[n][r*S+i][c*S+j]; // Convolution
```

Listing 2.1 Pseudo code of a convolution layer

## 2.2 Challenges of deploying ConvNets on embedded systems

Today, ConvNets are very popular in image classification tasks and already used in many cloud-based services, such as Google Image Translate and Prisma - an instagram like app. Although in the commercial sector many companies deploy them using on-line large clusters of GPUs and server-grade CPUs, large-scale adoption of state-of-the-art ConvNet-based off-line solutions on mobile and embedded consumer products is still not yet prominent. The majority of the existing off-line solutions support inference using cut-down versions of the state-of-the-art models. These devices are battery powered, with very limited resources and power budget, and thus pose a significant challenge on deploying ConvNets in embedded platforms. A desktop or server grade CPU or GPU consumes about $60-200Watts$, whereas a smart-phone is limited at approximately $5-6Watts$. For wearable consumer products the power margin even gets worse and usually is limited to $1Watt$ or less.

In contrast, current state of the art ConvNets are extremely power hungry due to their size, in both the number of layers and parameters. Table 2.1 summarises the storage requirement

Table 2.1 A snapshot of complexity in winning CNNs

| Release Year | CNN Model | #Conv layers | #MACC [millions] | #Weights [millions] | #Activations [millions] | ImageNet [top-5 err%] |
|---|---|---|---|---|---|---|
| 2012 | AlexNet | 5 | 1140 | 62.4 | 2.4 | 19.7 |
| 2014 | VGGNet-16 | 16 | 15470 | 138.3 | 29.0 | 8.1 |
| 2014 | GoogLeNet | 22 | 1600 | 7.0 | 10.4 | 9.2 |
| 2015 | ResNet-50 | 50 | 3870 | 25.6 | 46.9 | 7.0 |
| 2015 | Inception-v3 | 48 | 5710 | 23.8 | 32.6 | 5.6 |
| 2016 | SqueezeNet | 18 | 860 | 1.2 | 12.7 | 19.7 |

Table 2.2 Approximate energy cost of common operations and data transfers (at 45nm process node) Source: [42]

| Operation | Energy (pJ) | Relative cost |
|---|---|---|
| 16b Int ADD | 0.06 | 1 |
| 16b Int MULT | 0.8 | 13 |
| 16b FP ADD | 0.45 | 8 |
| 16b FP MULT | 1.1 | 18 |
| 32b FP ADD | 1.0 | 17 |
| 32b FP MULT | 4.5 | 80 |
| Register File, 1kB | 0.6 | 10 |
| L1 Cache, 32kB | 3.5 | 58 |
| L2 Cache, 256kB | 30.2 | 500 |
| DRAM | 640 | 10667 |
| Wireless Transfer | 60000 | 1000000 |

of the most popular state-of-the-art ConvNets from the ImageNet competition. As the table depicts, to achieve any realistic performance they would require lot of on-chip memory, significant memory bandwidth and lots of computational resources. Table 2.2 shows an approximate estimate of the energy costs of basic arithmetic operations and data transfers at a 45nm process node. Using those estimates, for example, a ConvNet with 1 billion connections to run at just 30Hz would require a minimum $(30Hz)(1Billion)(640pJ) = 19.2Watts$ just to access the parameters from off-chip storage, which is well beyond the limit of any mobile device. So, the current approach to alleviate the power-performance challenge is to run them in the cloud and download the result on client devices. This significantly limits the real-time performance that an application can achieve and also raises serious privacy concerns. In addition, sending an 720p or 1080p HD image at 30fps over the wireless network consumes a lot of power. As a result, the true potential of ConvNets can never be achieved without being able to run them on mobile platforms.

## 2.3 Arithmetic intensity and roof-line model

In analysing system performance bottlenecks, a roof-line model [97], a visually intuitive tool, is often used in multicore, manycore, or accelerator architectures. Using the roof-line model, one can examine the resultant ceilings in order to determine both the implementation and inherent performance limitations. The computational roof provides just an upper bound (the theoretical maximum) to performance. The computational ceilings impose a limit on the attainable performance that is below the actual roof-line, and indicate that the application cannot break through any one of these ceilings without first performing the associated optimisation. Computation and communication are two main constraints in achieving system throughput. As shown in Figure 2.3, a system can either be compute-bound or memory-

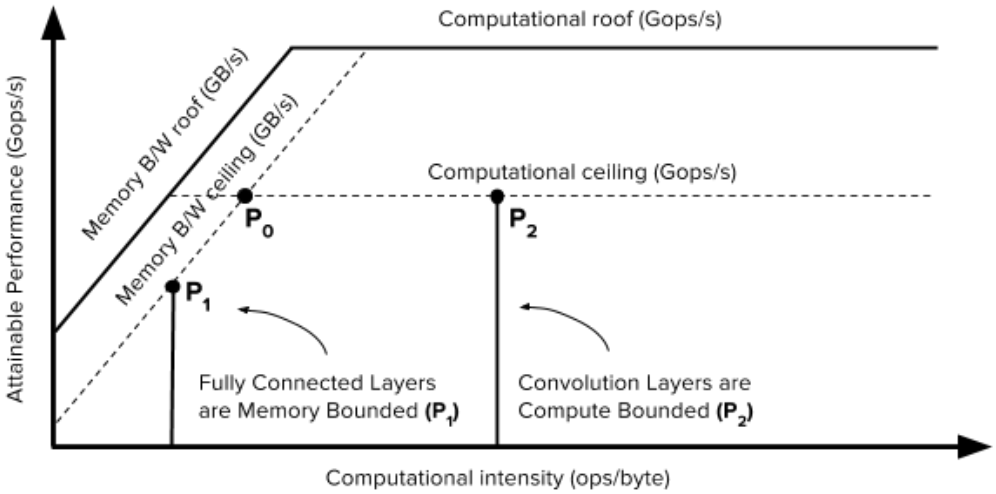



Fig. 2.3 Compute vs Memory bound

bound. The roof-line sets an upper bound on performance of a model depending on its computational or arithmetic intensity (ops/byte). If we think of computational intensity as a column that hits the roof, either it hits the flat part of the roof, meaning performance is compute-bound, or performance is ultimately memory-bound.

Using the roof-line model (see Figure 2.3), $P_2$ reflects the compute bound limit in conv-layers and $P_1$ reflects the memory bound limit in FC-layers. One has to be careful about which bottleneck to improve first. We can think of each optimisation as a performance ceiling below the appropriate roof-line, meaning you cannot break through a ceiling without first performing the associated optimisation.

In case of ConvNets, both types of limitation exist depending on the architecture - either the system can be ill designed so as to be starved or its peak compute capability is very

limited. Ideally, a system should be tuned in such a way that it coincides with point $P_0$ as this is the break-even point. In the following section, I discuss different types of convolutional layer architectures that are commonly used on modern ConvNets. I then compare those layer types using the arithmetic intensity metric.

## 2.4 Types of convolutional layers

Convolution is a widely used technique in signal processing. Although convolution in ConvNets is named after generic convolution, it is essentially the cross correlation in signal processing. In ConvNets, the filters in convolution are not reversed and elementwise multiplication and addition are performed to obtained the result. In deep learning, several variants of basic convolution are used within the convolutional layers. In this section, I summarise several types of convolution commonly used in deep convolutional neural networks.

### 2.4.1 Spatial convolution

The most common form of convolutional layer used in ConvNets is spatial convolution. Figure 2.4 illustrates an example spatial convolution layer. In this example, there are $Tm$ filters each of size $Tn \times Th \times Tl$. These are applied on the three dimensional input tensor of size $Tn \times H \times L$. After accumulation of the results for all the input channels it produces output of size $Tm \times R \times C$. As a concrete example, let us consider an input $5 \times 5$ RGB image which has 3 channels. Each set of filters has a dimension of $3 \times 3 \times 3$. First, each of the filters are applied to three channels in the input layer, separately. Three convolutions are performed, which result in 3 channels with size $3 \times 3$. Then these three channels are summed together using element-wise addition to form one single channel of size $3 \times 3 \times 1$. If there are multiple output channels then this process is repeated for each of the output channels.

### 2.4.2 1 x 1 convolution

$1 \times 1$ convolution is a special case of spatial convolution which is mainly used for dimensionality reduction for efficient computation [63]. By applying $1 \times 1$ convolution, the depth-wise dimension can be reduced significantly. It also can be used as an efficient low dimensional embedding or feature pooling. For example, if the original input layer has 512 channels, the $1 \times 1$ convolution will embed these channels (or features) into a single channel. Figure 2.5 illustrates an example $1 \times 1$ convolutional layer. The operation is trivial for layers with only one feature map. In this case, every pixel is multiplied by a number. In the example in

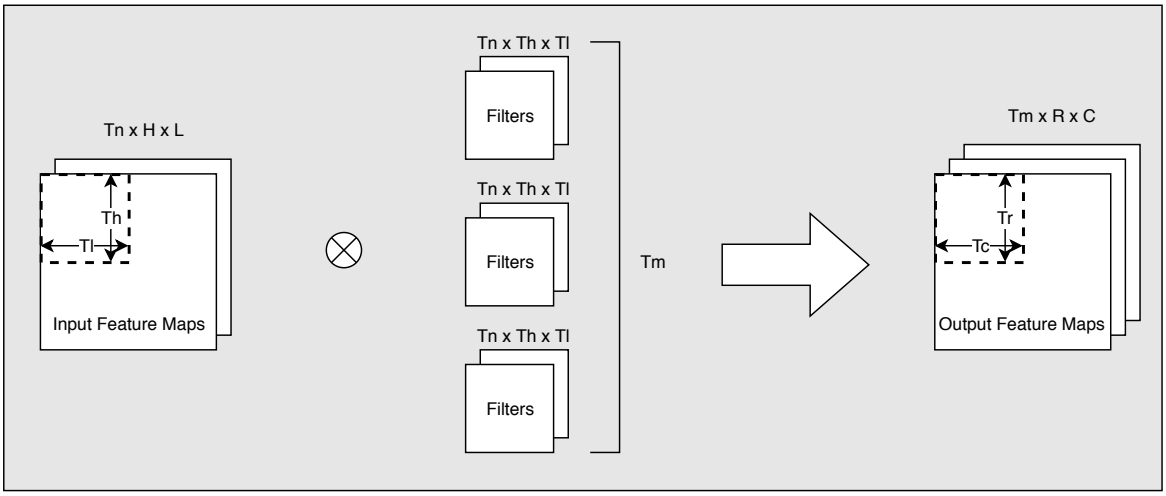



Fig. 2.4 Spatial convolution

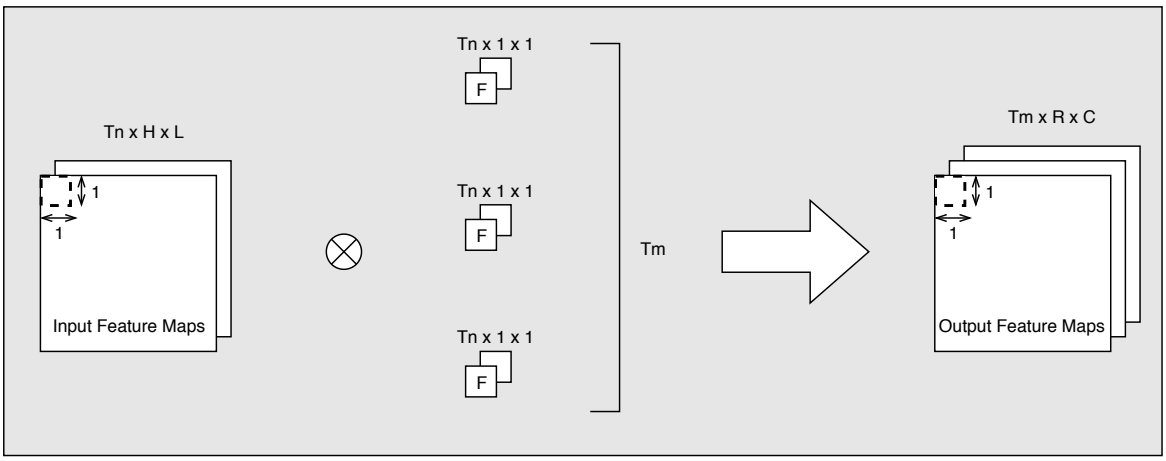



Fig. 2.5 1-by-1 convolution

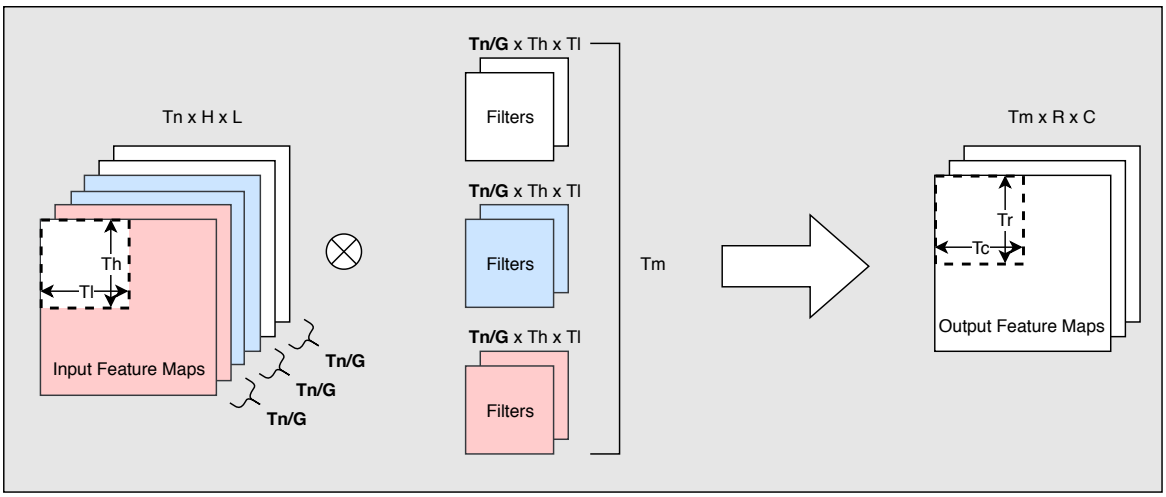



Fig. 2.6 Group convolution

the figure, $Tm$ sets of $Tn \times 1 \times 1$ filters are applied on a three dimensional input tensor of dimension $Tn \times H \times L$. This operation produces an output tensor of size $Tm \times R \times C$.

### 2.4.3 Grouped convolution

Grouped convolution helps to exploit model parallelisation by grouping channels. It was first introduced in AlexNet in 2012 to allow model training over two GPUs with limited memory [57]. In grouped convolution, the filters are separated in multiple groups. Each group is responsible for a conventional spatial convolution with a certain depth. Figure 2.6 illustrates an example layer of a group convolution. In this example, input feature maps are divided into G separate groups. In a similar manner, each set of filters has dimensions of $Tn/G \times Th \times Tl$ which are applied on the input tensors. This operation produces an output tensor of size $Tm \times R \times C$. One of the drawbacks of grouped convolution is a reduction in arithmetic intensity. As input channels are grouped into a small number of sets and they don't interact among different groups, filters are not reused across all the input channels.

### 2.4.4 Depth-wise separable convolution

Depth-wise separable convolution became a popular choice with the introduction of MobileNet and Xception [17]. The depth-wise separable convolutions consist of two phases: depth-wise convolutions and $1 \times 1$ convolutions. I have introduced $1 \times 1$ convolution earlier (see Figure 2.5). Figure 2.7 illustrates an example of a depth-wise convolution layer. The key property of the depth-wise convolution is that each kernel is applied to only one input channel. In the example, each filter of size $Th \times Tl$ is applied only on corresponding input channels. This process produces $Tm \times R \times C$ output maps. As the second phase of depth-wise separable convolution, to extend the depth, $1 \times 1$ convolution is applied. By separating the spatial convolution into two separate phases, the depth-wise separable convolution reduces the total number of operations. But, this comes at the cost of reduced arithmetic intensity (ops/byte).

## 2.5 Characteristics of convolutional layers

Modern deep ConvNets are dominated by small filters (e.g. $3 \times 3$, $1 \times 3$ etc.). They are the fundamental building blocks of many widely used ConvNets today. Therefore, optimising convolutions consisting of small kernels would help the overall performance of most models. The next most common filters in use in modern ConvNets are $1 \times 1$s. Table 2.3 presents the percentage of small and $1 \times 1$ kernels present in some of the widely used ConvNets.

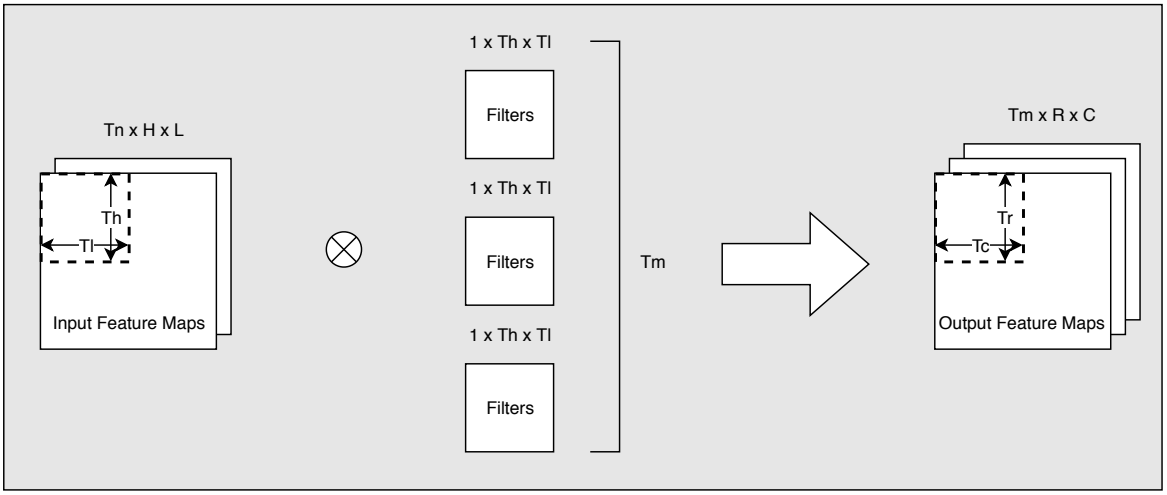



Fig. 2.7 Depth-wise convolution

There is no way the computation involved in a $1 \times 1$ convolution can be reduced further mathematically as it is just a pointwise multiplication between a pixel value and coefficient. I introduced a few other types of convolutional layers earlier. The magnitude of data reuse is defined by a metric called arithmetic intensity. Arithmetic intensity is the ratio between number of operations and memory accesses. The higher the number the more data reuse is possible. Unfortunately, many of the emerging convolution layer architectures do not yield high enough arithmetic intensity. Therefore, although theoretically they need fewer operations, accelerating them on hardware is difficult. Table 2.4 illustrates this point by comparing normalised arithmetic intensity between four widely used convolutional layer architectures. To compute the numbers in this table, it is assumed a generic case with 512 input and output channels. It also assumed that the input image has a dimension of $14 \times 14$. The group size is assumed to be 4 in the grouped convolution and 512 in the depth-wise convolution. As can be seen from the table, depth-wise convolution layers have 50 times poorer arithmetic intensity than that of basic spatial convolution.

## 2.6 Data reuse patterns

The diverse layer architectures and sizes of ConvNets result in substantial and varying data movement among the processing cores, on-chip storage and off-chip DRAM. To improve power efficiency an appropriate data-reuse pattern is required for ConvNet. The aim of the data-reuse pattern is to reduce redundant memory access to off-chip DRAM and maximise the utilisation of the already available data. In this section, I introduce the most common and widely used data-reuse patterns for ConvNets.

Table 2.3 Dominance of small kernels in modern ConvNets

| ConvNet | Percentage of 3x3, 5x5, 1x3, 1x5, 1x7 kernels | Percentage of 1x1 kernels |
|---|---|---|
| VGG16 | 100 | 0 |
| Inception-v3 | 57 | 43 |
| Inception-v4 | 59 | 41 |
| GoogLeNet | 35 | 65 |
| ResNet50 | 32 | 68 |
| MobileNet | 7 | 93 |
| SqueezeNet | 75 | 25 |
| DenseNet | 50 | 50 |
| Yolo-v3 | 55 | 45 |

Table 2.4 Arithmetic intensity of different types of convolutional layers

| Convolution Types | Arithmetic FLOPs (Millions) | Memory Ops Parameters + Activations (Millions) | Arithmetic Intensity (FLOPs/Mem Ops) | Normalised Arithmetic Intensity |
|---|---|---|---|---|
| Spatial Convolution (3x3) | 462 | 2.56 | 180 | 1.0 |
| Pointwise Convolution (1x1) | 51 | 0.463 | 110 | 0.6 |
| Group Convolution | 116 | 0.79 | 146 | 0.8 |
| Depthwise Convolution | 0.9 | 0.205 | 4.3 | 0.02 |

### 2.6.1 Input stationary (IS)

Input stationary is the first basic data-reuse pattern that requires minimum access to the input feature maps stored in on-chip SRAM. Figure 2.8 illustrates the typical usage of the input tiles for the input stationary data-reuse pattern. The compute engine loads input maps that are necessary to the activation register file (shown in red). These input maps are then fully reused to update all corresponding partial sums stored in the output partial-sum register file. The partial sums are sent back to the output on-chip SRAM buffer. The partial sums are later fetched by the processing engine to continue accumulation in depth when the next input feature map is loaded. Listing 2.2 presents the corresponding loop order for the input stationary data reuse. In the IS reuse pattern, loop N is the outer loop of loop M, meaning each input feature map is fully reused for computation before loading the next feature map.

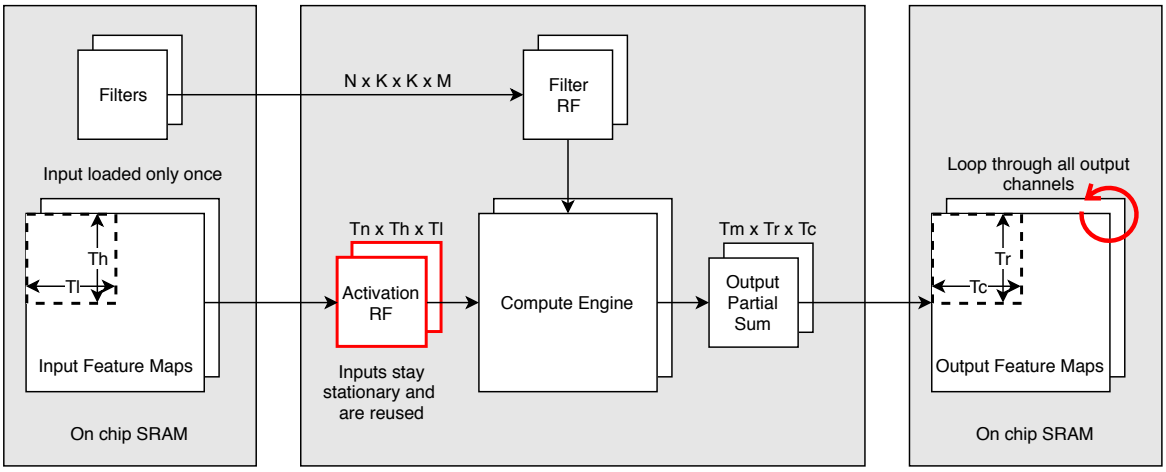



Fig. 2.8 Input-stationary data-reuse patterns in modern ConvNets

### 2.6.2 Output stationary (OS):

If the underlying algorithm requires frequent access to the partial sum, then the output-stationary reuse pattern reduces the overall memory accesses to the output feature maps stored in on-chip SRAM. Figure 2.9 illustrates how the output partial sum is reused in the OS reuse pattern. In OS reuse, different input feature maps of the same tile region are successively loaded by the compute engine. The partial sums in the output register files are reused until all the corresponding input feature maps are completed (shown in red). The final output activations are stored back to the on-chip SRAM and no further access is required. Any algorithms that require inverse transforms may benefit from the OS data-reuse pattern. Listing 2.3 presents the corresponding loop order for the output-stationary data-reuse. In OS

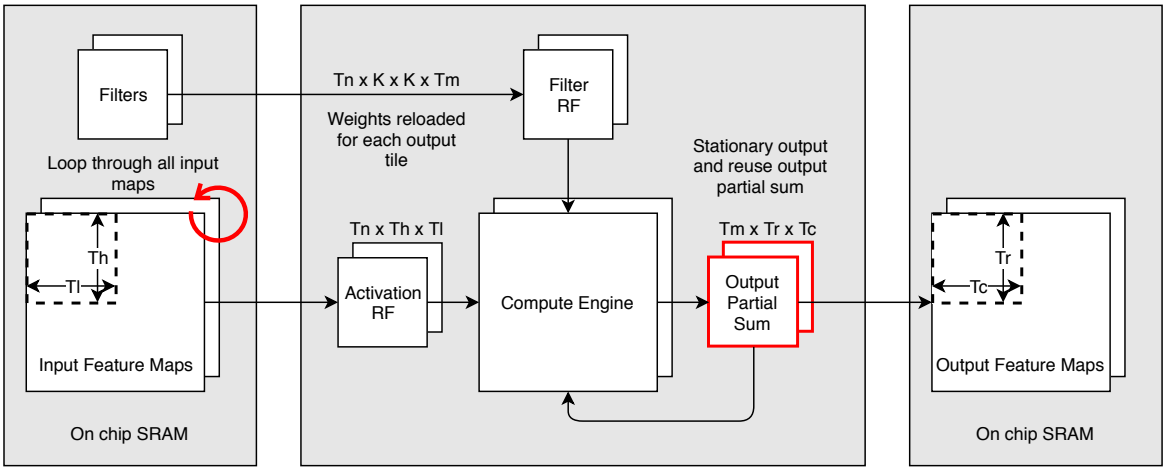



Fig. 2.9 Output-stationary data-reuse patterns in modern ConvNets

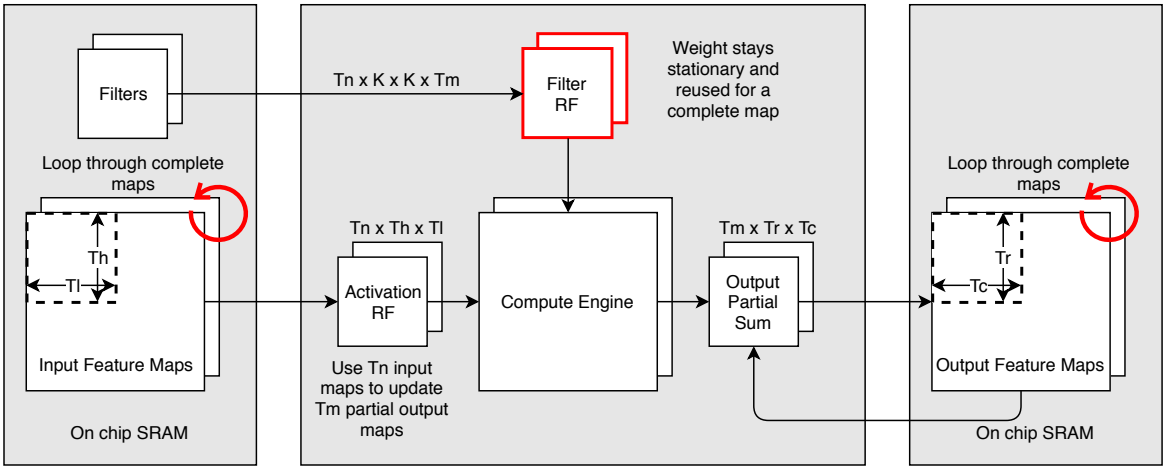



Fig. 2.10 Weight-stationary data-reuse patterns in modern ConvNets

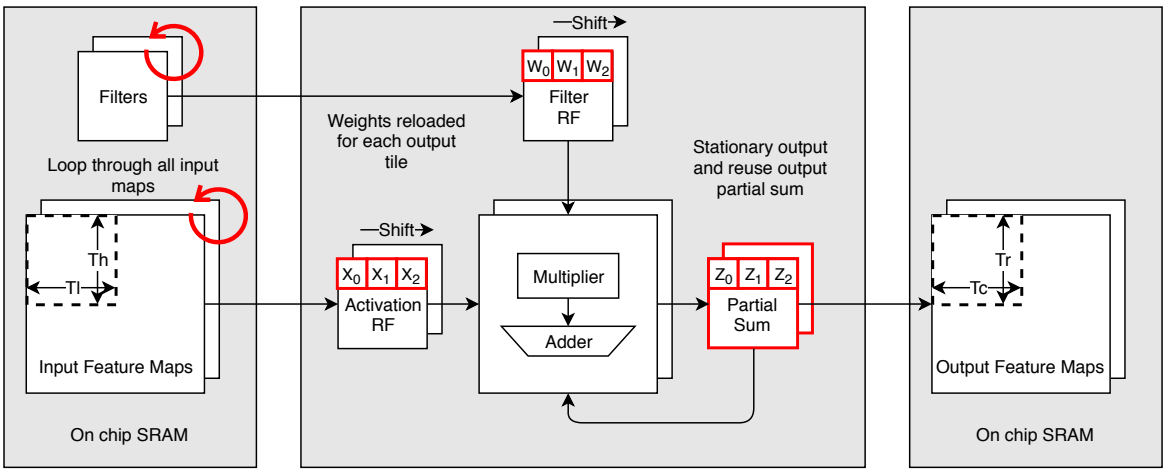



Fig. 2.11 Row-stationary data-reuse patterns in modern ConvNets

reuse pattern, loop M is the outer loop of loop N, meaning each output feature map is fully reused for computation before loading the next target output map.

```
for (r = 0; r < R; r += Tr)
  for (c = 0; c < C; c += Tc)
    for (n = 0; n < N; n += Tn)
      for (m = 0; m < M; m += Tm)
      {/* Inside the compute engine */
          for (tm = m; tm < min(m + Tm, M); tm++)
            for (tn = n; tn < min(n + Tn, N); tn++)
              for (tr = r; tr < min(r + Tr, R); tr++)
                for (tc = c; tc < min(c + Tc); tc++)
                  for (i = 0; i < K; i++)
                    for (j = 0; j < K; j++) {
                      Out[tm][tr][tc] += W[tm][tn][i][j]*X[tn][tr*S+i][tc*S+j];
                    }
      }/* End of single run inside the compute engine*/
```

Listing 2.2 Input-stationary data-reuse pattern

```
for (r = 0; r < R; r += Tr)
  for (c = 0; c < C; c += Tc)
    for (m = 0; m < M; m += Tm)
      for (n = 0; n < N; n += Tn)
      {/* Inside the compute engine */
          for (tm = m; tm < min(m + Tm, M); tm++)
            for (tn = n; tn < min(n + Tn, N); tn++)
              for (tr = r; tr < min(r + Tr, R); tr++)
                for (tc = c; tc < min(c + Tc); tc++)
                  for (i = 0; i < K; i++)
                    for (j = 0; j < K; j++) {
                      Out[tm][tr][tc] += W[tm][tn][i][j]*X[tn][tr*S+i][tc*S+j];
                    }
      }/* End of single run inside the compute engine*/
```

Listing 2.3 Output-stationary data-reuse pattern

### 2.6.3 Weight stationary (WS):

In both input-stationary and output-stationary reuse patterns, repeated weight access to off-chip DRAM would be necessary if the total number of required parameters exceeds the filter buffer size in the on-chip SRAM. In these cases the weight-stationary data-reuse pattern would minimise off-chip accesses due to a lack of on-chip space for the parameters. Figure 2.10 illustrates the typical reuse pattern of weight-stationary computing. The compute

engine loads $Tn$ tiled input feature maps to the activation register file. These input feature maps are used to compute $Tm$ output target maps. The $Tn \times Tm$ filter weights in the filter register file are reused to compute $Tm$ target output maps. Listing 2.3 presents the corresponding loop order for weight-stationary data reuse. In the WS reuse pattern, both the loop M and loop N are switched to the outer-most loops to maximise the reuse of the filter parameters.

```
for (m = 0; m < M; m += Tm)
  for (n = 0; n < N; n += Tn)
    for (r = 0; r < R; r += Tr)
      for (c = 0; c < C; c += Tc)
      {/* Inside the compute engine */
          for (tm = m; tm < min(m + Tm, M); tm++)
            for (tn = n; tn < min(n + Tn, N); tn++)
              for (tr = r; tr < min(r + Tr, R); tr++)
                for (tc = c; tc < min(c + Tc); tc++)
                  for (i = 0; i < K; i++)
                    for (j = 0; j < K; j++) {
                      Out[tm][tr][tc] += W[tm][tn][i][j]*X[tn][tr*S+i][tc*S+j];
                    }
      }/* End of single run inside the compute engine*/
```

Listing 2.4 Weight-stationary data-reuse pattern

### 2.6.4 Row stationary (RS):

The row-stationary data-reuse pattern was introduced in the Eyeriss system by reducing the size of the local register file [15]. By reducing the size of the local registers the RS data-reuse pattern helped the Eyeriss system to achieve an additional 25% energy savings. Figure 2.11 illustrates the row-stationary data-reuse pattern. The RS reuse pattern divides the MAC (multiply-accumulate) units into mapping primitives. Each mapping primitive performs a 1D (one-dimensional) row convolution in this setting. The row-stationary reuse pattern also exploits partial-sum reuse as in output stationary data-reuse pattern described earlier. The compute engine loads a row of filter weights and applies them to a row of input feature maps. This process generates a row of partial sums as shown in the figure.

### 2.6.5 On-chip interconnect for varying reuse and flexibility

The complexity of the on-chip interconnect depends on the level of data reuse supported by the hardware. In order to build a flexible accelerator the design must be able to adapt to a wide range of bandwidth requirements. When data reuse is low, a unicast network can

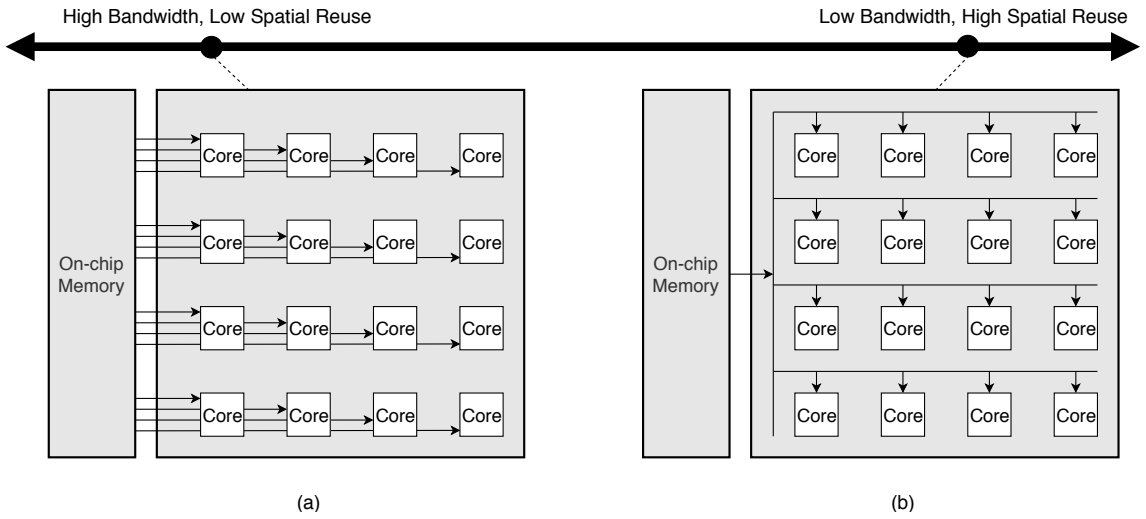



Fig. 2.12 Trade-off between on-chip interconnect complexity and spatial reuse. (a) Unicast network, (b) Broadcast network

provide high bandwidth from the memory and can keep the cores busy. However, when data reuse is high, the interconnect must support multicast or broadcast to exploit spatial data reuse. This helps to improve energy efficiency and processing array (or core) utilisation [86]. If the hardware is not required to support high reuse, then unicast is sufficient. Figure 2.12 shows these two extreme cases side-by-side as a comparison. Ideally to support any type of data-reuse pattern, an all-to-all interconnect is needed. But, for a realistic system this is very expensive and often not scalable. Therefore a trade-off is required between the amount of data-reuse necessary and the complexity of the interconnect that can be supported in the hardware.

## 2.7 A survey of prior research on optimising ConvNets

Recent research on ConvNets has focused primarily on improving accuracy. As more and more companies try to deploy deep-learning in their products and services, in the last couple of years, attention started to shift towards efficient implementation. There has been some fragmented research work conducted in both the machine learning and computer architecture communities to reduce the size and complexity of a ConvNet such that the overall storage and computation requirements can be reduced. In this section, I will highlight some of these initial research efforts.

### 2.7.1 Pruning, sparse Networks and regularisation

Model pruning has been the most widely studied topic among the various model optimisation techniques. Pruning introduces sparsity in the model, which can be exploited in the low level implementation. In an early work LeCun et al. [21] proposed pruning as an effective method to reduce over-fitting. Recently, Srivastava et al. proposed **Dropout** [85], a new technique to address the problem of over-fitting, which helps pruning models over time during the training process. The key idea in Dropout is to randomly drop activations along their connections from models during training. This prevents units from co-adapting too much. Today, Dropout is very common and believed to be one of the best methods for regularisation. While in Dropout a randomly selected subset of activations are set to zero, in **DropConnect** [95] a randomly selected subset of weights are set to zero. Dropconnect is a generalisation of dropout for regularising fully-connected layers. Goodfellow et al. proposed a new activation function called **Maxout** [35] that is particularly well-suited for training with Dropout, and they have shown Maxout improves accuracy of Dropout by exploiting model averaging behaviour. A more mathematically grounded work by Denton et al. showed how singular value decomposition **(SVD)** [24] can be used on a pre-trained network to speed up the bottleneck convolution operations in the first few layers with very negligible loss of performance. Their method also reduces the memory footprint of weights in the first two layers by 2-3x and the fully-connected layers by a factor of 5-13x.

In **ElasticNet** [101], Zou et al. proposed a novel shrinkage and selection method which produces a sparse model with good prediction accuracy. In this method, the authors use a complex cost function to train a model which consists of a $l_1$ penalty part and a quadratic part which encourages a grouping effect. Han et al. developed **Deep Compression** [40], a heuristic-based pruning technique, which begins with a pre-trained model, and then replaces weights that are below a certain threshold with zeros to form a sparse matrix. If the accuracy drops significantly, then they re-train the network with a few iterations to gain back the accuracy. Liu et al. [64] proposed the **Sparse Convolutional Neural Networks (SCNN)** model that exploits both inter-channel and intra-channel redundancy to maximise sparsity in a model. This procedure zeros out more than 90% of parameters, with a drop in accuracy less than 1% on ImageNet dataset. The authors also propose an efficient sparse matrix multiplication algorithm on CPU to further accelerate the optimised model.

### 2.7.2 Model simplification and separable layers

Many recent studies have shown that simplifying a model architecture may help to reduce overall compute complexity. For example, pooling layers in between convolutional layers

can be the bottleneck in many models while trying to exploit data parallelism. The same pooling effect can be achieved solely by a convolutional layer with increased stride without loss in accuracy. Springenberg et al. developed a ConvNet architecture by solely using convolutional layers. The authors showed that a very simple and streamlined **All-Conv-Net** [84] architecture can perform very well. Lin et al. took a very radical approach using a concept called **Network in network** [63] or multi-layer-perceptron-conv (in short mlpconv). Instead of simplifying the model with just one type of layer, the authors built a micro-network with a multilayer perceptron. The intermediate feature maps are obtained by sliding those micro-networks over the input in similar manner as in ConvNets. The outputs of a micro-network are then fed into the next layer. A deep ConvNet can be implemented by stacking multiple micro-networks instead of traditional convolutional layers. The Mlpconv work showed improvement in performance in image classification and the idea was later adopted by Google and others. Google's popular **Inception** [88] architecture started out as a case study based on the previous mlpconv work. The main idea of the architecture is to show how an optimal local sparse structure in a convolutional network can be approximated. Once an optimum local construction is found, the next step is to repeat it spatially. The current architecture of the Inception network is built upon an "Inception-module", a combination of filter sizes 1x1, 3x3, and 5x5, which are stacked on top of each other. The main advantage of this architecture is the improved utilisation of the computing resources inside the network. In the **SqueezeNet** [50] architecture, Iandola et al. take this concept further and proposed a Fire-module. The Fire-module is comprised of a squeeze convolution 1x1 filter feeding into an expand layer that has a mix of 1x1 and 3x3 convolution filters. The Fire module is configurable and a designer can choose the number of squeeze 1x1 filters, expand 1x1 and 3x3 filters. SqueezeNet can achieve AlexNet-level accuracy on the ImageNet dataset with 50x fewer parameters.

### 2.7.3 Quantisation, low precision arithmetic and BinaryNet

ConvNets are trained by applying tiny nudges to the weights using the back propagation algorithm [61] and these small increments require floating-point precision to work. Deploying a pre-trained fully optimised network to run forward inference does not require high precision. This is due to the fact that ConvNets can cope very well with noise. For example, in image classification tasks, the network has already learned to ignore sensor noise, lighting conditions, partial occlusion, and various other distortions due to different viewing angles. Precision loss in calculations is just another source of noise in ConvNets. While investigating a defect-tolerant hardware, Temam et al. showed that neural-network-based hardware accelerators have an intrinsic capability to cope with hardware faults without having to identify the

location of the faults. This **fault tolerance** [89] property of neural networks can be exploited in optimising ConvNets to achieve reasonable accuracy but with a smaller memory footprint. Courbariaux et al. carried out several experiments on running inference using ConvNets with three distinct formats, namely floating-point, fixed-point and dynamic fixed-point. They showed that the **Maxout** [20] network can use only 10 bits for computation and 12 bits for storage without significantly dropping accuracy. They further proposed that very low precision multipliers are sufficient for ConvNet and the idea can be exploited to design a very power-efficient hardware dedicated to deep-learning. **Gupta et al.** [37] studied the effect of limited precision data representation in the context of training a ConvNet. They observed that a ConvNet can be trained using only a 16-bit fixed-point number representation with little to no degradation in the classification accuracy. Subsequently, Han et al. extended their work on **Deep Compression** [40] by combining pruning with trained quantisation. They used 8 bits for each convolution layer and 5 bits for each fully-connected layer in AlexNet and achieved no accuracy loss. Gysel et al. in his master's thesis, developed **Ristretto** [38], a hardware-oriented approximation technique, where he condensed SqueezeNet to 8 bit with minimal loss of classification accuracy. Very recently, Google released a **TensorFlow** [36] extension where one can convert any pre-trained model to 8 bit format. In their current technique, the minimum and maximum range out of every operation becomes input to the following operation. This adds further complexity to the design of each layer. Recently, Courbariaux et al. introduced an extreme version of quantisation using binary neural networks. In their work on **Binarized Neural Network** [19], they have constrained the weights and activations to +1 or -1. The authors also showed that it is possible to train binary networks using the back-propagation algorithm. Binary networks can drastically reduce memory footprint while achieving nearly state-of-the-art accuracy. In the following year, Rastegari et al. proposed a variant of the binary neural network called **XNOR-Net** [79], which uses mostly bit-wise operations to approximate convolutions. The authors proposed that binarised neural networks offer the possibility of running state-of-the-art ConvNets on CPUs. But, it is still not clear if it would be possible to train a deep model using just two bits. In deep models, the gradient-vanishing problem becomes the main bottleneck in training. The trade-off between the minimum number of bits required in implementing a ConvNet and being able to train using back propagation is an interesting topic for research.

### 2.7.4 Data compression, hashed-net, weight sharing

**Denil et al.** [23] demonstrated in their paper that a surprisingly large amount of redundancy is present among the weights in most neural networks. The authors showed that a small subset of the weights are sufficient to reconstruct the entire network. This motivated some

researchers to exploit weight sharing to achieve compression. Chen et al. in their work **HashedNet** [12] exploit this redundancy using a low-cost hash function to randomly group connection weights into hash buckets. All connection weights in the same hash bucket share the single parameter. Han et al. use a **code book** [42] which consists of pre-decided weights and re-train the network using weights only from the code book. In addition, they apply **Huffman coding** [40] to take advantage of the biased distribution of the effective weights in the code book. The authors also showed that a lossless data compression technique like Huffman coding can save non-uniformly distributed weight values in 20-30% less storage.

### 2.7.5 Fast arithmetic, logarithmic representation, and FFT

The runtime of forward inference is profoundly dominated by the convolution layers. Reducing any part of this time-consuming computation process can bring a massive improvement in performance. Cong *et al.* proposed an algorithmic modification in convolution arithmetic to reduce their computational workload. The authors used classical **Strassen's algorithm** [18] in computing convolutional matrix multiplication. Their experiment showed that by using such a method computation load can be reduced by up to 47%. The authors further proposed that other algorithms from linear algebra can be extended to their convolutional matrix multiplication. Vasilache *et al.* from Facebook introduced Fast-Fourier-Transform-based [93] convolution implementation. The authors showed that both their ***fbfft*** implementation and Nvidia's ***cuFFT*** library are faster than time-domain cuDNN-based implementations. They also highlight the fact that, for small filters, straightforward time-domain convolutions outperform Fourier frequency-domain convolutions. FFT-based implementations tend to win for large filters which are very uncommon in modern ConvNets. Recently, Nervana Systems, now part of Intel, proposed **Winograd** [16], a minimal filtering algorithm which yields speed-ups of 2-3x over Nvidia's **cuDNN** v4 kernels. Miyashita *et al.* proposed a new data representation that enables ConvNets to be encoded in 3 bits with negligible loss of classification accuracy. Since weight and activations have a non-uniform distribution, the authors used base-2 **logarithmic representation** [72] to encode weights and activations. They also showed that end-to-end training can be performed using log representation at 5 bits which achieves higher test accuracy than linear 5 bits models.

### 2.7.6 Architecture learning, hyper-parameter tuning and network architecture search (NAS)

In the realm of deep learning, *hyper-parameter tuning* is a "meta" learning task. Hyper-parameters could have a large impact on the capacity and achievable accuracy of a model.

The final architecture of a ConvNet depends on how one comes up with how big or small the network should be, how many layers it should have, how the neurons between layers are connected, etc. Optimal hyper-parameter settings also highly depend on target datasets. In the recent past, machine-learning researchers used heuristic-based techniques to come up with a reasonable model. But, almost no one can say for sure that is the most optimised model. The biggest challenge in hyper-parameter tuning is that the quality of those hyper-parameters cannot be written down in a closed-form formula. Grid search and manual search are the most widely used strategies for hyper-parameter optimisation. In **Grid search** [46], the designer chooses a grid of hyper-parameter values, evaluates every one of them and picks the best. Grid search is simple enough to parallelise; but it is the most expensive method in terms of total computation time. Bergstra et al. proposed a random-search algorithm to get an answer in a small fraction of computation time compared to traditional grid-search method. **Random search** [7] experiments are also easier to carry out than grid-search-based experiments for practical reasons related to the statistical independence of every trial so that they can be carried out asynchronously. Maclaurin et al. proposed a **gradient-based hyper-parameter optimisation** [67] technique. The authors accomplish this by computing exact gradients of cross-validation performance with respect to all hyper-parameters by chaining derivatives backwards through the entire training procedure. But, this method has several limitations. Learning long-term dependencies with gradient descent is difficult. Since they use gradients to optimise functions which depend on their hyper-parameters through hundreds of iterations of stochastic gradient descent, the learning difficulties are compounded. Verbancsics et al. proposed that the **Hypercube-based NeuroEvolution of Augmenting Topologies (HyperNEAT)** [94] technique can be effective in training a feature extractor for backward propagation learning. The authors further hypothesise that HyperNEAT could provide a potentially interesting path for combining reinforcement learning and supervised learning in image classification. Very recently, network-architecture search (NAS) with underlying hardware parameters in a loop became popular [65, 9]. NAS methods have outperformed many manually designed architectures on some tasks such as image classification [100, 80], object detection [100] or semantic segmentation [11]. NAS can be seen as subfield of AutoML [49] and has significant overlap with hyperparameter optimisation [31] and meta-learning [92].

## 2.8 A survey of hardware-based specialisation of ConvNet

In recent years, many computer-architecture and systems researchers have proposed a number of convolutional neural-network accelerators to achieve high performance and low power

consumption. The main purpose of this section is to briefly highlight some of these platforms and as well as a few commercial ones.

## 2.8.1 ML accelerators from academic research

### 2.8.1.1 NeuFlow

**Neuflow** [30] was one of the first widely known ASIC implementation of a ConvNet accelerator targeted to embedded system. The architecture of NeuFlow consists of runtime-configurable tiles connected on a 2D grid. Each tile can be configured to do one of the tasks, namely, convolution, DMA, basic arithmetic, non-linear functions. Implemented in IBM 45nm SOI technology the chip sits on a $12.5mm^2$ silicon footprint. It can achieve a performance of about 300GOPS/s at 0.6Watt operating at 400 MHz with an external memory bandwidth of 4 x 1.6 GB/s full-duplex.

### 2.8.1.2 Origami

Cavigelli et al. published **Origami** [10], another ASIC implementation of a ConvNet accelerator targeting mainly computer-vision workloads. It consists of 4 sum-of-product blocks, each of which can compute a 7x7 patch of an image in one cycle. The architecture has a silicon footprint of $3.09mm^2$ in the UMC 65nm process technology and is capable of a throughput of 274 GOPS/s at 369GOp/s/W with an external memory bandwidth of just 525MB/s full-duplex. The authors further claimed that if the design is scaled to 28nm and uses a LPDDR3 memory, the performance can reach up to 630 GOPS/s/W.

### 2.8.1.3 ShiDianNao

Du et al. published **ShiDianNao** [28], a near-camera ConvNet accelerator for visual workloads. ShiDianNao consists of 8x8 array of processing element, each of which can perform one of the layers of a network. The processing elements are connected in a systolic-array fashion, which they call a Neural Functional Unit. Data is shared among the processing elements within the NFU and thus avoids frequent DRAM accesses. The ASIC implementation has an area of $4.85mm^2$ in a 65nm process, and can achieve a peak performance of 194 GOPS/s while consuming 320.10mW at 1GHz.

### 2.8.1.4 Eyeriss

**Eyeriss** [14] is a spatial architecture particularly aimed at accelerating the convolution layer rather than a complete ConvNet. It supports a very limited number of filter sizes

and activation functions. The test chip features a spatial array of 168 processing elements arranged in a systolic-array style fed by a reconfigurable multicast on-chip network that handles many shapes and minimises data movement by exploiting data reuse. The chip can run the convolutions in AlexNet at 35fps with 278mW power consumption,

#### 2.8.1.5 EIE

Han et al. proposed an energy-efficient inference engine **(EIE)** [39] that performs forward inference on a compressed network and accelerates the resulting sparse matrix-vector multiplication with weight sharing. In the analysis, the authors presume that the network is compressed using **deep-compression** [40] which was their earlier work. Evaluated on nine different ConvNet models, EIE is 189x and 13x faster when compared to a CPU and GPU without their compression technique. The chip has a peak processing power of 102 GOPS/s while using the compressed network, corresponding to 3 TOPS/s on an uncompressed network. The EIE chip consumes 0.58W in 45nm.

#### 2.8.1.6 AngelEye

**Angel Eye** [78] is an FPGA-based ConvNet accelerator, unlike the ASICs. The authors also used a number of processing elements connected in a systolic-array fashion. In addition, they used 8-bit quantisation in the weight space. Their implementation, using a Xilinx Zynq ZC706 board, achieves a frame rate of 4.45fps with AlexNet-level accuracy. Angel Eye can achieve 137 GOPS/s running at 150 MHz and consumes less than 9.63W power.

### 2.8.2 ML accelerators from industry

#### 2.8.2.1 Myriad from Movidius (now Intel)

In the commercial space, Movidius recently released their Myriad 2 MA2450 based **Fathom** [99] neural USB stick. The company claims that GoogleNet-level complex ConvNets can be run on Fathom at a nominal 15 fps with FP-16 precision. Fathom's performance ranges from 80 to 150 GFLOPS, depending on the ConvNet complexity and precision. The performance requires less than 1.2 W of power which is more than 12 times as efficient as, for example, NVidia's Tegra X1 processor.

#### 2.8.2.2 ML accelerator from Arm

Very recently, Arm announced a machine-learning accelerator that enables ConvNet inference on edge devices, saving power and enhancing user privacy [4]. Arm's custom accelerator pro-

vides 4 TOPS/s performance, and an efficiency of 5 TOPS/Watt in a 7nm process technology. The accelerator supports both int-8 and int-16 data types for its operations. The configurable architecture of the accelerator can be connected as 8 processing units in a cluster and up to 64 processing units in a mesh topology. The main interface of the accelerator uses standard AXI4 and ACE-5 lite interconnect standards for easy integration with existing SoCs.

#### 2.8.2.3 NVDLA from Nvidia

NVDLA is a microarchitecture designed by Nvidia for the acceleration of deep-learning workloads. Since the original implementation targeted Nvidia's own Xavier SoC, the architecture is specifically optimised for convolutional neural networks as the main types of workloads deal with images and videos, although other networks are also supported [76]. NVDLA primarily targets edge devices, IoT applications, and other lower-power inference designs. The accelerator supports some level of configurability that includes 8 or 16-bit data paths, 1 or 2 RAM interfaces, flexible core size. The company claims 5.1 TOPS/Watt peak performance for ResNet50 that uses an int-8 data type at 16nm process technology.

#### 2.8.2.4 TPU from Google

A TPU or tensor processing unit is a custom accelerator for ML workloads developed by Google [52]. The first generation TPU was an 8-bit matrix multiplication engine driven with CISC instructions. It was manufactured on a 28nm process with a die size of $331mm^2$. The thermal power budget of the first-generation chip was more suited to data centres. The main core of the accelerator consisted of a 256x256 systolic array of 8-bit multipliers. It also had a massive 28MB of on-chip memory. While the third generation of the same TPU has now 8-fold improvements in performance, Google also released an Edge TPU for mobile applications. The Edge TPU only supports 8-bit arithmetic.

### 2.8.3 A power-performance comparison among ASICs, FPGA and GPU-based implementation

Finally, I compare a few publicly available throughput and energy efficiency numbers for different hardware-based ConvNet accelerators (see Table 2.5). According to the comparison, GPUs can achieve the highest throughput, although they are not as energy efficient as an ASIC. Generally, GPUs are designed to serve multiple applications and thus are dominated by dark silicon. A custom ASIC can achieve the highest throughput per Watt, while FPGA-based accelerator fall somewhere in between the other two.

Table 2.5 Power-Performance comparison of different hardware implementations

| Platform (Source: [1, 10, 75, 78]) | Tech Node | Datatype | Throughput | Power | Throughput/Watt |
|---|---|---|---|---|---|
| ASIC (Intel Myriad X) | 16nm | FP16 | 4 TOPS/s | 1 Watt | 4 TOPS/s/Watt |
| ASIC (Arm ML processor) | 7nm | INT8 | 4.6 TOPS/s | 1.5 Watt | 3 TOPS/s/Watt |
| ASIC (Origami) | 65nm | INT12 | 74 GOPS/s | 0.09 Watt | 803 GOPS/s/Watt |
| ASIC (NeuFlow) | 45nm | INT16 | 294 GOPS/s | 0.6Watt | 490 GOPS/s/Watt |
| FPGA (AngelEye) | 20nm | INT8 | 137 GOPS/s | 9.63 Watt | 14.2 GOPS/s/Watt |
| GPU Embedded grade (Nvidia TK1) | 28nm | FP32 | 155 GOPS/s | 10.2 Watt | 15.2 GOPS/s/Watt |
| GPU Server grade (Nvidia Titan X) | 16nm | INT8 | 44 TOPS/s | 250 Watt | 176 GOPS/s/Watt |

## 2.9 Summary

To summarise, in this chapter, I began by providing a general overview of a ConvNet architecture, followed by an exhaustive survey of current state-of-the-art techniques for ConvNet optimisation. I also described detailed architectures of most the widely used convolutional layers. In the latter part, I highlighted some of the popular hardware-based ConvNet specialisations.

# Chapter 3

# Model optimisation by decomposing convolutional layers into approximately separable one-dimensional kernels

In this chapter I begin with three main motivating ideas, namely, redundancy in ConvNet, efficient mapping primitives, and separable filters. I then explain how layers of deep ConvNets can be restructured by combined application of these three ideas that helps to reduce overall compute complexity of the convolutional layers. I then continue with the details of the rank-search algorithms and evaluation results. I conclude the chapter with an in-depth discussion section on applicability of approximation methods.

## 3.1 Redundancy in the parameterisation of ConvNets

Research has shown that significant redundancy exists in the parameterisation of most modern deep networks [23]. Filters in the convolutional layers learn to detect particular patterns at multiple spatial locations in the input feature map. It is very natural that these patterns occur in different orientations: for example, edges in images can be arbitrarily oriented. As a result, ConvNets often learn multiple copies of the same filter in different orientations. This is especially prominent when the feature maps exhibit some level of rotational symmetry. It may therefore be effective to encode a form of such symmetry in the architecture of a ConvNet, just like parameter sharing resulting from a convolution operation encodes translational symmetry. To achieve this invariance to a class of transformations of the feature map, it is very common practice to randomly perturb the training examples with transformations to encourage the ConvNet to produce the correct classification regardless of

how a particular feature is transformed, as shown in Figure 3.1. Provided that the model has enough capacity, it should be able to learn such invariances from examples in most cases [25]. But the training using data augmentation often results in a very large model consisting of redundant neurons. To speed up inference and reduce test-time power consumption, this redundancy property can be exploited to approximate the model with negligible impact on accuracy. As described in the previous chapter, pruning unwanted neurons is one of the very popular approaches to removing this redundancy. In many early works in training ConvNets, statistical pruning-based techniques proved to be effective in reducing over-fitting ([62],[43]). More recent works extend this idea to reduce the number of nodes by iterative pruning and re-training ([41],[73]). The biggest limitation of this approach is that other than heuristic approaches, there is no concrete numerical way to control this process of pruning. One often arrives at a better solution by iterative trial and error, which is not reproducible. A more mathematically grounded approach is to apply filter approximation by using dimensionality reduction ([24],[51] ). In this method, the original filters are reconstructed back from a new basis of representation. The new basis function is learnt simply by minimising the error (e.g. L2 regularisation), whilst penalising the rank of the filters. My layer restructuring scheme, ADaPT (Adapt-Decompose-Tune), follows from this fundamental idea and extends it further to a more hardware-suitable scheme that is numerically tractable and metrics driven. The ADaPT scheme helps to approximate any large pre-trained ConvNet statically that is more hardware friendly.

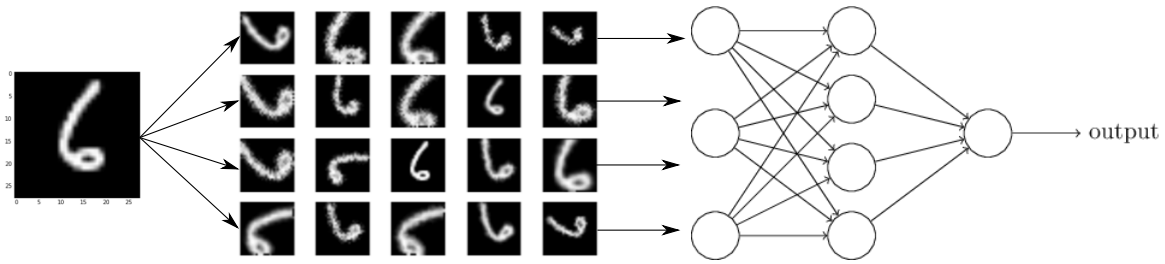



Fig. 3.1 A Typical data augmentation used during training ConvNet for MNIST dataset

## 3.2 Need for energy-efficient data-reuse pattern

Efficient implementation of convolutional layers on embedded devices always has been a challenge due to the limited on-chip storage (e.g. scratchpad or register files) and power budget. One of the challenging design decisions is the choice of appropriate mapping primitive that optimises data reuse per PE (i.e. processing engine) yet still having enough

flexibility to allow different spatial and temporal mappings. To this end, research has shown that the row-stationary data-reuse pattern is the most effective and energy efficient one that fully exploits partial-sum reuse [13]. In this scheme, each mapping primitive performs a one-dimensional row or column convolution. The row-stationary dataflow divides the MACs (multiply and accumulate) into mapping primitives, each of which comprises a subset of MACs that run on the same PE in a fixed order. The ordering of these MACS enables the use of a sliding window (shifts) for input sequences, as shown in Figure 3.2. The ADaPT scheme not only applies static approximation on any pre-trained model but also decomposes each layer into two cascaded one-dimensional convolutional layers that optimises the on chip data reuse and simplifies the hardware.

## 3.3 Separable filters in the context of convolution

The concept of creating separable filters by splitting convolution operations into convergent sums of matrix-valued stages was proposed by Hummel and Lowe in the 1980s before ConvNets became popular for automatic feature learning [48]. This property was exploited in many early image-processing filters—e.g., the Sobel edge-detection filter, the Gaussian-blurring filter, etc. This approach is very powerful but restricted to filters that are decomposable, which is often not the case for a trained filter such as in ConvNets. However, due to the presence of inherent redundancy between different filters or feature maps within a layer, this property can be exploited in the acceleration of ConvNet models.

Consider an arbitrary kernel of a ConvNet described by the $(m \times n)$ matrix $\mathcal{W}$.

$$\mathcal{W} = \begin{bmatrix} \alpha_{00} & \alpha_{01} & .. & \alpha_{0n} \\ \alpha_{10} & \alpha_{11} & .. & \alpha_{1n} \\ .. & .. & .. & .. \\ \alpha_{m0} & \alpha_{m1} & .. & \alpha_{mn} \end{bmatrix} \tag{3.1}$$

We say that kernel $\mathcal{W}$ is separable when it can be split into the outer product of an $m$-length column vector $v$ and an $n$-length row vector $h$ as follows:

$$\mathcal{W} = \mathcal{V}\mathcal{H}^{\mathcal{T}} = \begin{bmatrix} v_0 \\ v_1 \\ .. \\ v_m \end{bmatrix} \begin{bmatrix} h_0 & h_1 & .. & h_n \end{bmatrix} \tag{3.2}$$

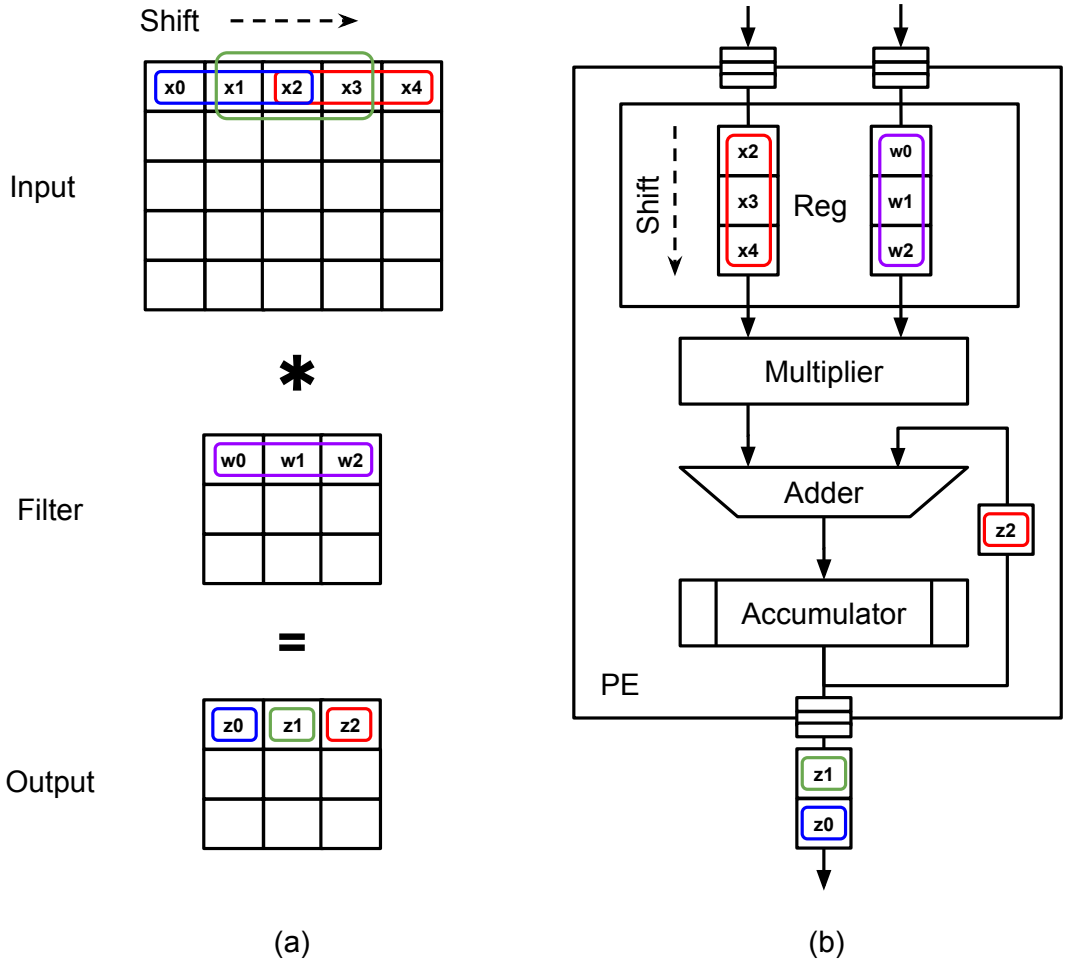



Fig. 3.2 Mapping of one-dimensional convolution primitive to a Processing Engine (PE). (a) Input:x is shifted right and multiplied with one-dimensional filter:w (b) In the PE, the filter:w stays stationary and input:x is shifted down, the partial sum is accumulated and shifted out of the PE

or, $\mathcal{W}$ can be explicitly expressed as:

$$\mathcal{W} = \mathcal{V}\mathcal{H}^{\mathcal{T}} = \begin{bmatrix} v_0h_0 & v_0h_1 & .. & v_0h_n \\ v_1h_0 & v_1h_1 & .. & v_1h_n \\ .. & .. & .. & .. \\ v_mh_0 & v_mh_1 & .. & v_mh_n \end{bmatrix} \tag{3.3}$$

From Equations (3.1) and (3.3), it is apparent that a separable kernel has equivalent rows and columns. To store the original kernel $\mathcal{W}$ in Equation (3.1), it would require $(mn)$ space. However, if the kernel $\mathcal{W}$ is a separable matrix, then we see from Equation (3.3) that it would require $(m+n)$ space. As $m$ and $n$ becomes large and the original kernel is separable, $\mathcal{W}$, one can see that substantial savings in computational time and storage will be achieved.

Unfortunately, we cannot generally expect that any trained kernel in a ConvNet satisfies such stringent conditions. The collection of kernels in a ConvNet is generally of full rank and expensive to convolve with large images. However, we can aim for $\mathcal{W}$ to be approximately separable such that

$$\mathcal{W} = \mathcal{V}\mathcal{H}^{\mathcal{T}} + \mathcal{E} \tag{3.4}$$

where $\mathcal{E}$ is an error kernel, whose importance we would like to be as small as possible in relation to the original kernel $\mathcal{W}$. We can further generalize Equation (3.4) in the following form:

$$\begin{aligned} \mathcal{W} &= \mathcal{V}_1\mathcal{H}_1^T + \mathcal{V}_2\mathcal{H}_2^T + ... + \mathcal{V}_i\mathcal{H}_i^T + ... + \mathcal{V}_r\mathcal{H}_r^T + \mathcal{E}_r \\ &= \mathcal{U}_1 + \mathcal{U}_2 + ... + \mathcal{U}_i + ... + \mathcal{U}_r + \mathcal{E}_r \end{aligned} \tag{3.5}$$

where each term,

$$\mathcal{U}_i = \mathcal{V}_i\mathcal{H}_i^T \tag{3.6}$$

is an exactly separable rank-1 outer product of a column vector of length $m$ and row vector of length $n$, and $\mathcal{E}_r$ is the error matrix associated with $r$-term approximation of the original kernel $\mathcal{W}$ as shown in Figure 3.3. Eckart and Young showed that the SVD (Singular-Value Decomposition) is the solution to the problem of minimising $\mathcal{E}_r$ [29]. Furthermore, if the original kernel $\mathcal{W}$ can be well approximated by $r$ rank-1 updates, we will only require $r(m+n)$ parameters to describe the kernel instead of the original $mn$ elements. The key idea here is that if we choose $r$ such that $r(m+n) << mn$, then it would require less storage and computation. We can extend this idea to the convolutional neural network to reduce the overall cost of computation.

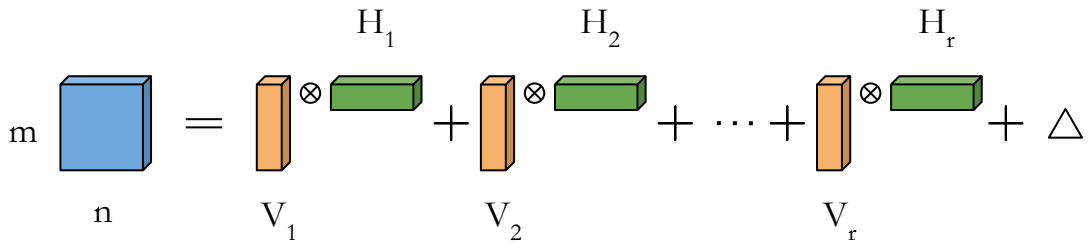



Fig. 3.3 A two-dimensional (2D) matrix can be represented by the sum of $r$ rank-1 updates.

## 3.4 Layerwise approximation and convolution by separability - putting it together

In ConvNets, multiple layers of convolutional filter (also known as kernel) banks are stacked on top of each other, followed by a non-linear activation function. Significant redundancy exists between those spatial filter dimensions and also along cross-channel feature maps. Most previous research focused on either exploiting approximation along spatial filter dimensions or along one of the feature-channel dimensions. In the ADaPT scheme, I aim at approximating the redundancy across both the input, output feature maps and filters. To exploit the inherent redundancy in each layer, I extend the concept of a separable filter as described earlier and obtain a unique approximation reconstruction of each convolution kernel.

To be more specific, mathematically, let us assume in a convolutional neural network a four-dimensional kernel can be represented as $\mathcal{W} \in \mathbb{R}^{F_I \times (m \times n) \times F_O}$, where spatial two-dimensional kernels are of size $(m \times n)$ and $F_I$, $F_O$ are the input and output channels within a layer, respectively. We can also represent an input feature map as $\mathcal{X} \in \mathbb{R}^{M \times N \times F_I}$ and corresponding kernels as $\mathcal{W}_i \in \mathbb{R}^{m \times n \times F_I}$ for $i^{th}$ set of weights, where each input feature map is of size $(M \times N)$. The original convolution for the $i^{th}$ set of weights in a given layer now becomes

$$\mathcal{W}_i * \mathcal{X} = \sum_{f=1}^{F_I} \mathcal{W}_i^f * x^f \tag{3.7}$$

Our goal is to find an approximation of kernel $\mathcal{W}_i$, such that $\mathcal{W}_i = \widetilde{\mathcal{W}_i} + \mathcal{E}$. Using the concept of separable filters [81], let us assume that for a small error $\mathcal{E}$, the chosen rank is $R$. I will explain the rank $(R)$ selection algorithm in detail in the next section. The modified kernel now can be represented by Equation (3.8), where $\mathcal{V} \in \mathbb{R}^{R \times (m \times 1 \times F_I)}$ is the approximate column kernel, and $\mathcal{H} \in \mathbb{R}^{F_O \times (1 \times n \times R)}$ is the approximate row kernel.

$$\widetilde{\mathcal{W}_i} * \mathcal{X} = \sum_{f=1}^{F_I} \sum_{r=1}^{R} \mathcal{H}_i^r (\mathcal{V}_r^f)^T * x^f = \sum_{r=1}^{R} \mathcal{H}_i^r * (\sum_{f=1}^{F_I} \mathcal{V}_r^f * x^f) \tag{3.8}$$

Note that in Equation (3.8) the approximate two-dimensional kernel $\widetilde{\mathcal{W}_i}$ is replaced with two one-dimensional kernels, namely, $\mathcal{V}$ and $\mathcal{H}$, that satisfy the requirement of an efficient mapping primitive described earlier. Figure 3.4 depicts the idea of re-constructing the convolution layer using the newly constructed column and row low-rank kernels and compares them with the original two-dimensional (2D) direct convolution. The column and row kernels $(\mathcal{V}, \mathcal{H})$ are computed statically using generalised eigenvalue decomposition by minimising the error $\mathcal{E}$. Unlike techniques where rank is learnt by minimising L2 reconstruction error, in the case of ADaPT, the magnitude of the rank is decided statically. Thus in ADaPT, the long running time of learning-based approximation techniques are altogether avoided. Additionally, as the approximation is an inherent property of each layer, all the convolutional layers in a ConvNet can be reconstructed in parallel, which saves time.

## 3.5 Rank search and layer-restructuring algorithm

Now, I will turn our attention to the core rank-search algorithm. Typically, all modern deep ConvNets consist of many convolution layers. The rank $R$ of each convolution layer is chosen by the one-shot minimisation criterion described below. Firstly, the original four-dimensional (4D) tensor $\mathbb{R}^{F_I \times m \times n \times F_O}$ for each convolution layer is reshaped to a 2-D tensor $\mathbb{R}^{(F_I m) \times (n F_O)}$ and then eigenvalues are obtained by the application of singular value decomposition. Unlike other minimisation criteria, such as the Mahalanobis distance metric or the data covariance distance metric [24], this simple criterion gives us an exact decomposition. Each tensor is then approximated based on a given compression factor and then decomposed into cascaded vertical and horizontal one-dimensional convolution layers. Algorithm (1) describes the main steps of the ADaPT low-rank approximation and ConvNet layer-restructuring scheme. In the algorithm, the compression factor of each convolution layer and pre-trained weights of the individual layers are provided as arguments. Typically, the compression factors can be obtained as requirements by running power-performance simulation on a target system using a specific model. The algorithm returns the reduced weight of each vertical and horizontal filter. After running this approximation algorithm for each convolution layer, the accuracy of the final model is validated with a test set.

**Algorithm 1:** Rank search and the layer restructuring algorithm

**Function LayerwiseReduce** $(M, C, W)$;
**Input :** Target ConvNet model: M, Kernel Dimension: $p_i$,
Compression factor of each layer: $[c_1, c_2, .., c_n]$,
Pre-trained weights of individual layer:$[w_1, w_2, .., w_n]$
**Output :** Reduced ConvNet Model: $M^*$,
Reduced weights of each layer: $[v_1, v_2, .., v_n]$, $[h_1, h_2, .., h_n]$
**for** $i \leftarrow 1$ ***to*** *Layers* **do**
  **if** $layerType == Conv$ **then**
    $targetRank \leftarrow \frac{p_i F_I F_O}{c_i(F_I + F_O)}$;
    $U\Lambda V^T \leftarrow SVD(w_i)$;
    $disconnectLayers(w_i)$;
    $v_i \leftarrow U\sqrt{\Lambda}$;
    $h_i \leftarrow V\sqrt{\Lambda}$;
    $addNewLayer(targetRank)$;
    $M^* \leftarrow reconstructModel(M, v_i, h_i)$;
  **end**
**end**

## 3.6 Evaluation

To evaluate the ADaPT compression scheme both the original and corresponding approximate models were deployed to four different target hardware, namely, Arm Cortex-A15, NVidia Tegra-K1 (Kepler architecture), Intel Core i7-5930k and NVidia's Titan-X (Maxwell architecture). Furthermore, to demonstrate the general applicability of this scheme on a wider variety of platforms, both embedded-grade and desktop-grade hardware were chosen. The Cortex-A15 was used as a single-threaded target running at up to 2.3GHz. The Tegra K1 with 192 CUDA cores were run at 852MHz. The Intel Core i7-5930k was also used as a single-threaded target running at up to 3.5GHz. The Maxwell generation Titan-X with 3072 CUDA cores was run at 1.08GHz. On top of standard CUDA 6.5, TK1 was used with cuDNN v2 (max supported version) and Titan X was used with cuDNN v4. For evaluation in the CPUs the OpenBLAS 0.2.18 optimised BLAS library was used. In all cases, the FP32 datatype was used. Table 3.1 presents the configuration of each target platform. In each case the entire network was run 20 times and I reported the mean run-time; noise in the system contributes to slight variations in the time spent in each layer of the network.

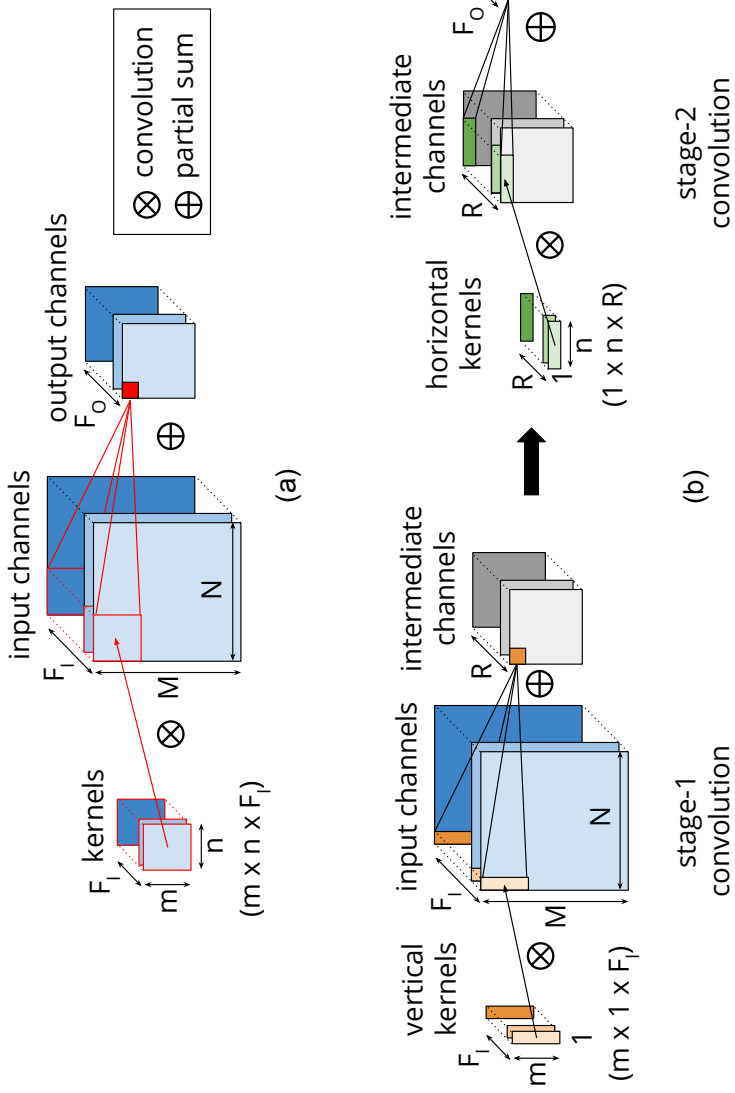



Fig. 3.4 **(a)** The original convolution with a $(m \times n)$ kernel. **(b)** The two-stage approximate convolution using a $(m \times 1)$ column kernel in stage 1 followed by a $(1 \times n)$ row kernel in stage 2. There are R channels in the intermediate virtual layer.

Table 3.1 Configurations of the target platforms

| Core | Operating System | Library | Clock Speed |
|---|---|---|---|
| Intel i7-5930k | Ubuntu 16.04 | OpenBLAS 0.2.18 | 3.5GHz |
| NVidia TitanX | Ubuntu 16.04 | cuDNN v4 | 853MHz |
| ARM Cortex-A15 | Ubuntu 16.04 | Eigen (TF1.x) | 2.3GHz |
| NVidia Tegra-K1 | Ubuntu 16.04 | cuDNN v2 | 1.08GHz |

A number of ConvNet models associated to four widely used data-sets (MNIST, CIFAR-10, ILSVRC and PASCAL VOC) with varying complexity and size were modified and deployed on real hardware. The following section provides an overview of each of these datasets and associated models used in the evaluation.

### 3.6.1 MNIST - Digit Recognition

The MNIST dataset consists of handwritten digit images and it is divided into 60,000 examples for the training set and 10,000 examples for testing. In the evaluation the official training set is again divided into an actual training set of 50,000 examples and 10,000 validation examples.

Table 3.2 MNIST model approximation summary

| Layer | Original $(F_I \times m \times n \times F_O)$ | Original Parameter Count | Compressed Column $(F_I \times m \times n \times V_R)$ | Compressed Row $(V_R \times m \times n \times F_O)$ | Reduction in Layer Size |
|---|---|---|---|---|---|
| Conv-1 | $1 \times 5 \times 5 \times 32$ | 1K | $1 \times 5 \times 1 \times 4$ | $4 \times 1 \times 5 \times 32$ | 1.2x |
| Conv-2 | $32 \times 5 \times 5 \times 64$ | 51K | $32 \times 5 \times 1 \times 16$ | $16 \times 1 \times 5 \times 64$ | 6.6x |

All digit images have been size-normalised and centred to a fixed-size image of 28x28 pixels. I've adapted a two layer convolutional neural network similar to the LeNet architecture which had a baseline accuracy of 99.23% [59]. The adapted model also has a fully connected dense layer at the end. In all the results and analysis going forward I focus only on the convolution layer as the approximation scheme is targeted to convolution layers. Using the ADaPT scheme, an approximation of the original model was reconstructed. A summary of the original and the newly constructed models is presented in Table 3.2. The modified model was then deployed in all the four different target hardware platforms. The approximate model achieve a maximum speed up of 2.8x without drop of any accuracy. A comparison of speed up between the original and corresponding approximate model for all four target platforms is shown in Figure 3.7. I observed that for the MNIST model, the speed up in the GPUs is less than that of CPUs. I believe this may be due to some of the CUDA cores being under-utilised. A layerwise breakdown of speed up is also shown in Table 3.3. The model

can be approximated further by trading off negligible accuracy by choosing a lower rank in each layer during approximation.

Table 3.3 MNIST speedup of convolution layers on the Arm Cortex A15 without loss of base-line accuracy of 99.23%

| Layer | Original (ms) | Compressed (ms) | Speed-up |
|---|---|---|---|
| Conv-1 | 2.6 | 1.2 | 2.1x |
| Conv-2 | 8.3 | 1.6 | 5.7x |
| Total | 15.7 | 5.6 | 2.8x |

### 3.6.2 CIFAR-10 - Object Classification

The CIFAR-10 classification is a common benchmark problem in machine learning. The CIFAR-10 dataset consists of 60,000 32x32 colour images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images. The dataset is divided into five training batches and one test batch, each with 10,000 images. For the evaluation purposes, I've adapted the CIFAR-10 model, an all convolutional neural network available in the Google's open source Tensorflow library. This model, consisting of alternating convolutions and non-linearities, now comes as a default example with the Tensorflow source code.

Table 3.4 CIFAR-10 model approximation summary

| Layer | Original $(F_I \times m \times n \times F_O)$ | Original Parameter Count | Compressed Column $(F_I \times m \times n \times V_R)$ | Compressed Row $(V_R \times m \times n \times F_O)$ | Reduction in Layer Size |
|---|---|---|---|---|---|
| Conv-1 | $3 \times 5 \times 5 \times 32$ | 2K | $3 \times 5 \times 1 \times 4$ | $4 \times 1 \times 5 \times 32$ | 3.4x |
| Conv-2 | $32 \times 5 \times 5 \times 64$ | 51K | $32 \times 5 \times 1 \times 16$ | $16 \times 1 \times 5 \times 64$ | 6.6x |
| Conv-3 | $64 \times 5 \times 5 \times 128$ | 205K | $64 \times 5 \times 1 \times 32$ | $32 \times 1 \times 5 \times 128$ | 6.6x |

Table 3.5 CIFAR-10 layerwise speedup of convolutions on the Arm Cortex A15 without any loss of base-line accuracy of 86%

| Layer | Original (ms) | Compressed (ms) | Speed-up |
|---|---|---|---|
| Conv-1 | 2.4 | 1.1 | 2.2x |
| Conv-2 | 8.8 | 1.5 | 5.9x |
| Conv-3 | 9.1 | 1.5 | 6.1x |
| Total | 25.6 | 8.8 | 2.9x |

Using SGD-based training this CIFAR-10 model achieves a peak performance of about 86% accuracy within a few hours of training time on the GPU. This model is then compressed

using the three-stage ADaPT approximation technique to speedup the convolution layers. A summary of the decomposition is shown in Table 3.4. The approximate model achieved a maximum speed-up of 2.9x from twenty runs without any loss of base-line accuracy. Table 3.5 presents the layerwise speedup of the model. Both the original and the approximate model were also deployed in all four target platforms. The results obtained from these evaluation are summarised in Figure 3.7. It is worth noticing that some layers achieve a speedup up to 6x, which is very impressive for convolution layers. The model can further be approximated by trading off accuracy, as shown in Fig 3.5.

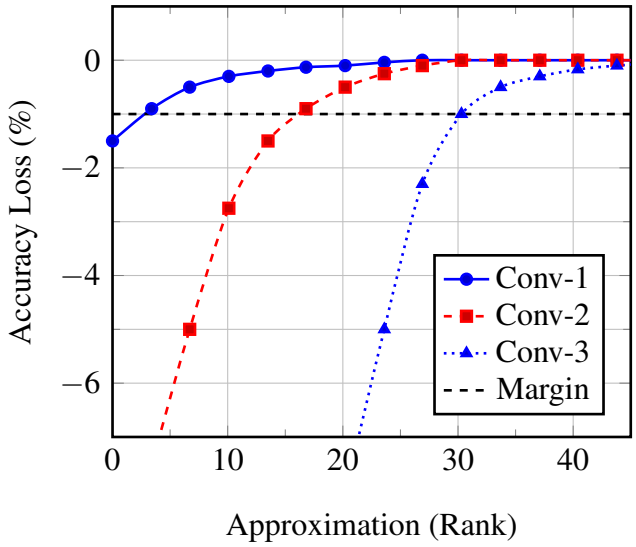



Fig. 3.5 Layerwise accuracy loss vs rank-approximation trade-off for the CIFAR-10 model

### 3.6.3 ILSVRC-2012 - Image Classification

Since 2010, the annual ImageNet Large-Scale Visual-Recognition Challenge (ILSVRC) has been a competition where research teams from academia and industry submit models that classify and detect objects and scenes. For evaluation of the ADaPT scheme, I have specifically focused on the image classification task using the ImageNet data-set. The classification task on the ILSVRC challenge is made up of 1.2 million images in the training set, each labelled with one of 1,000 categories that cover a wide variety of objects, animals, scenes, and even some abstract geometric concepts such as a "hook", "spiral" etc. The 100,000 test set images are released with the dataset, but the labels are withheld to prevent teams from over-fitting the test set. The teams have to predict 5 (out of 1,000) classes and an image is considered correct if at least one of the predictions is the ground truth.

Table 3.6 VGG16 layerwise speed-up of convolution on the i7-5930k (per image) with no loss of base-line accuracy of 90.5%

| Layer | Original (ms) | Compressed (ms) | Speed-up |
|---|---|---|---|
| conv3-64-1.1 | 28.5 | 15.0 | 1.9x |
| conv3-64-1.2 | 168.1 | 32.3 | 5.2x |
| conv3-128-2.1 | 74.1 | 25.6 | 2.9x |
| conv3-128-2.2 | 147.3 | 42.1 | 3.5x |
| conv3-256-3.1 | 67.3 | 15.7 | 4.3x |
| conv3-256-3.2 | 134.2 | 25.8 | 5.2x |
| conv3-256-3.3 | 134.5 | 27.4 | 4.9x |
| conv3-512-4.1 | 65.2 | 12.8 | 5.1x |
| conv3-512-4.2 | 129.9 | 22.0 | 5.9x |
| conv3-512-4.3 | 130.1 | 21.3 | 6.1x |
| conv3-512-5.1 | 33.4 | 4.3 | 7.8x |
| conv3-512-5.2 | 33.5 | 4.2 | 7.9x |
| conv3-512-5.3 | 33.4 | 4.2 | 7.9x |
| Total | 1432.3 | 252.8 | 5.7x |

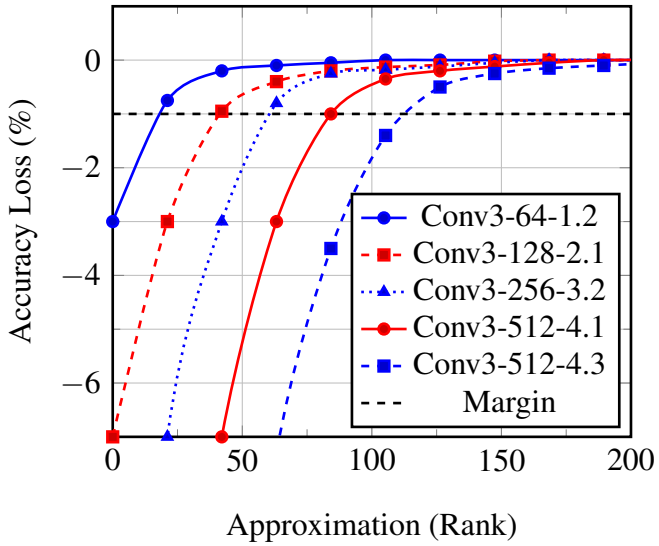



Fig. 3.6 Accuracy loss vs rank approximation trade-off in selected layers from VGG16

For evaluation purposes, the widely used VGG-16 model is used in this experiment. The VGG-16 model won the ImageNet competition in the ILSVRC-2014 challenge. The VGG-16 model is a very deep architecture and consists of 13 convolutional layers out of a total of 16 layers. Researchers and engineers often apply transfer learning to modify the weights in the last few dense layers to re-target the model for a different class of problem. The first 13 convolution layers are very performance critical as they stay unchanged after the modification. I have re-used the weights from the pre-trained model which I obtained from the webpage of the authors of the model [82]. This model achieves a top-5 accuracy of 90.6%. The ADaPT scheme is applied on this deep model to approximate and decompose each convolution layer. Table 3.7 provides the layerwise structure of the approximate model and compares with the original model. Both models were then deployed to target hardware platforms for evaluation. The new model achieves an impressive 5.7x maximum speed-up without loss of base-line accuracy. It is worth mentioning that this speed-up comes solely from the approximation of the convolutional layer. A detailed layerwise speed-up is presented in Table 3.6 for a single-threaded implementation in Intel platform. The model can further be approximated by trading off accuracy as shown in Figure 3.6. A further comparison between the original and approximate model targeting different hardware is presented in Figure 3.7.

### 3.6.4 PASCAL VOC - Image Segmentation

For the final experiment, I chose an image segmentation task using an all-convolutional deep neural network. In computer vision, image segmentation is the process of partitioning a digital image into multiple segments. I believe that the future of surveillance, defence, security, deliveries and many other commercial sectors will be shaped by drone technology. In air drone technology image segmentation is a fundamental and challenging task. To achieve this researchers adapt pre-trained state-of-the-art deep convolutional neural networks and transfer their learned representations to the segmentation task by fine-tuning.

The PASCAL Visual Object Classes (VOC) challenge is a benchmark in visual object category recognition, detection and segmentation, providing the vision research communities with a standard dataset of images and annotations. The VOC dataset consists of annotated consumer photographs collected from the flickr photo-sharing web-site. For evaluation purposes, the segmentation-equipped FCN-VGG16 model is considered [66]. This fine-tuned model achieves a state-of-the-art 56.0% mean IU score on it's validation set. I applied our three stage ADaPT approximation method to get rid of inherent redundancies in the model. The newly constructed approximate model achieves a 2.8x speed-up without any loss in mean IU score. The layerwise approximation almost follows the VGG16 model, which is shown in Table 3.7. The approximate model was also deployed on four different hardware

Table 3.7 VGG16 model approximation summary

| Layer | Original ($F_I \times m \times n \times F_O$) | Original Parameter Count | Compressed Column ($F_I \times m \times n \times V_R$) | Compressed Row ($V_R \times m \times n \times F_O$) | Reduction in Layer Size |
|---|---|---|---|---|---|
| conv3x3-64-1.1 | $3 \times 3 \times 3 \times 64$ | 2K | $3 \times 3 \times 1 \times 4$ | $4 \times 1 \times 3 \times 64$ | 2.1x |
| conv3x3-64-1.2 | $64 \times 3 \times 3 \times 64$ | 37K | $64 \times 3 \times 1 \times 12$ | $12 \times 1 \times 3 \times 64$ | 8.0x |
| conv3x3-128-2.1 | $64 \times 3 \times 3 \times 128$ | 74K | $64 \times 3 \times 1 \times 40$ | $40 \times 1 \times 3 \times 128$ | 3.2x |
| conv3x3-128-2.2 | $128 \times 3 \times 3 \times 128$ | 148K | $128 \times 3 \times 1 \times 40$ | $40 \times 1 \times 3 \times 128$ | 4.8x |
| conv3x3-256-3.1 | $128 \times 3 \times 3 \times 256$ | 295K | $128 \times 3 \times 1 \times 50$ | $50 \times 1 \times 3 \times 256$ | 5.1x |
| conv3x3-256-3.2 | $256 \times 3 \times 3 \times 256$ | 590K | $256 \times 3 \times 1 \times 60$ | $60 \times 1 \times 3 \times 256$ | 6.4x |
| conv3x3-256-3.3 | $256 \times 3 \times 3 \times 256$ | 590K | $256 \times 3 \times 1 \times 70$ | $70 \times 1 \times 3 \times 256$ | 5.5x |
| conv3x3-512-4.1 | $512 \times 3 \times 3 \times 512$ | 1M | $512 \times 3 \times 1 \times 80$ | $80 \times 1 \times 3 \times 512$ | 6.4x |
| conv3x3-512-4.2 | $512 \times 3 \times 3 \times 512$ | 2M | $512 \times 3 \times 1 \times 100$ | $100 \times 1 \times 3 \times 512$ | 7.7x |
| conv3x3-512-4.3 | $512 \times 3 \times 3 \times 512$ | 2M | $512 \times 3 \times 1 \times 110$ | $110 \times 1 \times 3 \times 512$ | 7.0x |
| conv3x3-512-5.1 | $512 \times 3 \times 3 \times 512$ | 2M | $512 \times 3 \times 1 \times 80$ | $80 \times 1 \times 3 \times 512$ | 9.6x |
| conv3x3-512-5.2 | $512 \times 3 \times 3 \times 512$ | 2M | $512 \times 3 \times 1 \times 78$ | $78 \times 1 \times 3 \times 512$ | 9.8x |
| conv3x3-512-5.3 | $512 \times 3 \times 3 \times 512$ | 2M | $512 \times 3 \times 1 \times 78$ | $78 \times 1 \times 3 \times 512$ | 9.8x |

platforms to compare its performance with the original model. A summary of the speed-up comparison is shown in Figure 3.7. It is interesting to note that the speed-up achieved in each case varies between platforms. This gap appears due the differences in underlying hardware architecture and the linear-algebra libraries used to accelerate convolutions. Often the lack of enough on-chip memory is the problem as the intermediate activations cannot be kept on-chip for further processing. As a result, intermediate activations must be stored back to the main memory and retrieved back later for further processing.

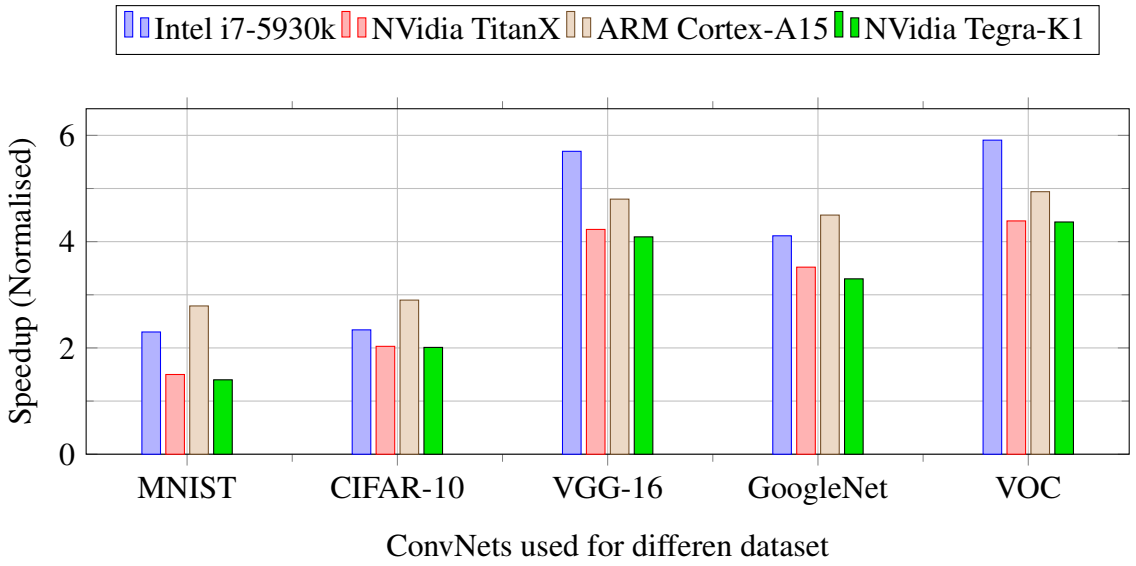



Fig. 3.7 Speed-up of selected ConvNets on different target platforms solely by approximating the convolutional layers without any loss of baseline accuracy

Table 3.8 VGG16 layerwise comparison of the number of strong operations, i.e. multiplications (MULs) between (a) a direct 2-D approach, (b) Tucker decomposition [55], (c) Pruning [41], and (d) the ADaPT

| Layers (mxn-maps-name) | Feature Size (MxN) | Speedup (x) (a) 2-D Direct (#MULs) | (b) Tucker [55] | (c) Pruning [41] | (d) ADaPT[Mine] |
|---|---|---|---|---|---|
| conv3x3-64-1.1 | 224x224 | 1.0x (0.1B) | 1.0x | 1.7x | 2.1x |
| conv3x3-64-1.2 | 224x224 | 1.0x (1.85B) | 10.2x | 8.3x | 8.0x |
| conv3x3-128-2.1 | 112x112 | 1.0x (0.9B) | 9.2x | 3.3x | 3.2x |
| conv3x3-128-2.2 | 112x112 | 1.0x (1.85B) | 6.7x | 3.4x | 4.8x |
| conv3x3-256-3.1 | 56x56 | 1.0x (0.9B) | 4.0x | 2.3x | 5.1x |
| conv3x3-256-3.2 | 56x56 | 1.0x (1.85B) | 7.8x | 6.3x | 6.4x |
| conv3x3-256-3.3 | 56x56 | 1.0x (1.85B) | 3.0x | 3.5x | 5.5x |
| conv3x3-512-4.1 | 28x28 | 1.0x (0.9B) | 4.5x | 4.8x | 6.4x |
| conv3x3-512-4.2 | 28x28 | 1.0x (1.85B) | 3.9x | 7.1x | 7.7x |
| conv3x3-512-4.3 | 28x28 | 1.0x (1.85B) | 5.4x | 6.7x | 7.0x |
| conv3x3-512-5.1 | 14x14 | 1.0x (462.5M) | 5.2x | 8.3x | 9.6x |
| conv3x3-512-5.2 | 14x14 | 1.0x (462.5M) | 5.7x | 11.1x | 9.8x |
| conv3x3-512-5.3 | 14x14 | 1.0x (462.5M) | 5.4x | 9.1x | 9.8x |

## 3.7 Comparison with prior work

As described in the background and the related work chapters of this dissertation a number of other techniques are also used to reduce the overall compute complexity of the convolution layers. To evaluate the effectiveness of the ADaPT approximation scheme, which solely aims at reducing computation complexity of the convolution layers, I compare this scheme with a pruning scheme [41] and an alternative low-rank approximation scheme based on Tucker decomposition [55]. Both these techniques help to reduce the number of convolutions in ConvNet and hence these are worth comparing. The reader may notice that I do not compare the ADaPT approximation scheme with any of the existing quantisation schemes. This is due to the orthogonal nature of the two schemes. The ADaPT scheme exploits the inherent redundancy among filters, whereas all quantisation schemes exploit the precision and range of the numbers in the computation that helps to reduce both the storage and computation cost. Similarly, I do not compare ADaPT with any of the existing fast arithmetic scheme (e.g. Winograd convolution) which reduce the total number of multiplications per convolution, as it is completely orthogonal. In fact, both the quantisation and fast arithmetic schemes can be applied in addition to ADaPT to further reduce the computation complexity of the model.

For a layerwise comparison, I considered the commonly used VGG-16 model (which uses the IMAGENET dataset) as both other research works used the same network as a common use case. Table 3.8 provides a layerwise comparison of the size of feature maps along with the number of multiplications required in case of a direct 2-D convolution on the VGG-16 baseline model. The last three columns compare speedups from three different

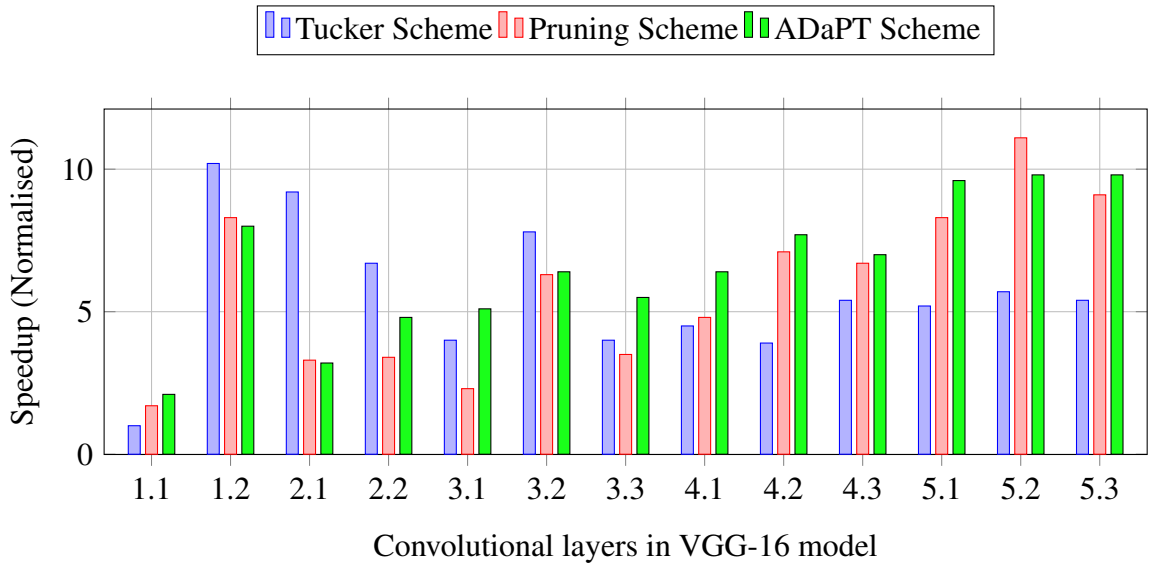



Fig. 3.8 VGG layerwise compute complexity (number of FLOPs) comparison between Tucker Decomposition [55], Pruning [41] and ADaPT scheme.

schemes, namely, Tucker decomposition, pruning while learning, and the ADaPT scheme. Figure 3.8 presents a graphical comparison of the number of flops for VGG-16 for the three selected schemes. As can be seen from the figure, the ADaPT scheme wins in almost all the convolution layers against pruning. The Tucker-decomposition-based scheme achieves better speedup for the first few layers except the very first input layer. Tucker's scheme based on the higher-order SVD algorithm is very effective on large feature sizes which are mostly present in the earlier layers. As we go deeper in the network, the feature sizes start to increase and ADaPT-based scheme becomes more effective. Since the layers in the middle of the network dominate in the number of computations, as can be seen in Table 3.8, ADaPT yields overall more speedup compared to the Tucker-decomposition-scheme. Pruning also helps speedup the later layers compared to the Tucker-decomposition-based technique. But, this comes at the cost of a very long iterative re-training phase. From the results, we can conclude that the ADaPT approximation scheme achieves the best speedup without the need for any long re-training time.

## 3.8 Discussion

In this chapter I showed how the ADaPT compression scheme can be applied statically on a deep pre-trained model that arrives at a deterministic solution. Furthermore, to achieve additional speedup in inference other orthogonal schemes, such as quantisation and fast

arithmetic, can still be applied after applying this scheme. Unlike pruning, the ADaPT scheme helps to achieve a significant reduction in compute complexity without the need for any costly interactive prune and train cycles. I conclude this chapter with an analysis on compute and storage cost reduction that can be achieved through this scheme.

### 3.8.1 Reduction of Compute and Storage Cost

The computational cost of the original direct convolution is of order $\mathcal{O}(F_I MNmnF_O)$. But, using the approximation technique as above, the computational cost for first stage convolution is $\mathcal{O}(F_I MNmR)$ and for second stage convolution is $\mathcal{O}(RMNnF_O)$, resulting in a total computational cost of $\mathcal{O}((mF_I + nF_O)MNR)$. If we choose R such that $R(mF_I + nF_O) << mn(F_I F_O)$, then computational cost can be reduced. In practice, a convolutional neural network uses square kernels. Hence, let us assume $m = n = p$, which is the size of the kernel. Using this assumption the condition can be simplified to $R(F_I + F_O) << pF_I F_O$. In addition, most modern ConvNets use more channels in the higher layers than the corresponding lower layers, i.e. the channel ratio $\frac{F_O}{F_I} >> 1$. The higher the ratio, the larger the value of $R$ can be. In most layers the computation cost can be reduced by $p$, which is the dimension of the kernel in the respective layer.

Similarly, the initial cost of storage is $F_I F_O p^2$, whereas cost is $(F_I pR + RpF_O)$ after approximation and separating the kernels into two rectangular ones. If we choose $R << p\frac{F_I F_O}{(F_I + F_O)}$, significant savings can be made for the storage costs of the kernels. Tables 3.2, 3.4 and 3.7 show relative savings made in storage costs in each convolutional layer of MNIST, CIFAR-10, and VGG16 model, respectively.

### 3.8.2 Efficient Use of Memory Bandwidth

Fetching data from off-chip main memory (DRAM) costs an order of magnitude more than from on-chip or local storage ([22], [47]). Chen et al. in their Eyeriss research project showed that a row-stationary 1-D convolution is a more optimal solution for throughput and energy efficiency than traditional 2-D convolution [15]. Separable filters enable row-stationary 1-D convolutions by reducing the number of unnecessary data loads in padded convolution by dividing the convolution into two 1-D stages. To preserve information many convolutional networks use zero-padding in many layers. Around the image tile, there is an *apron* of pixels that is required in order to filter the image tile (see figure 3.9). Note that the *apron* of one block also overlaps with the adjacent blocks. If we separate the convolution into row and column passes, it is no longer necessary to load the top and bottom *apron* regions for the row stage of computation. Similarly, for the column stage, left and right *apron* regions are no

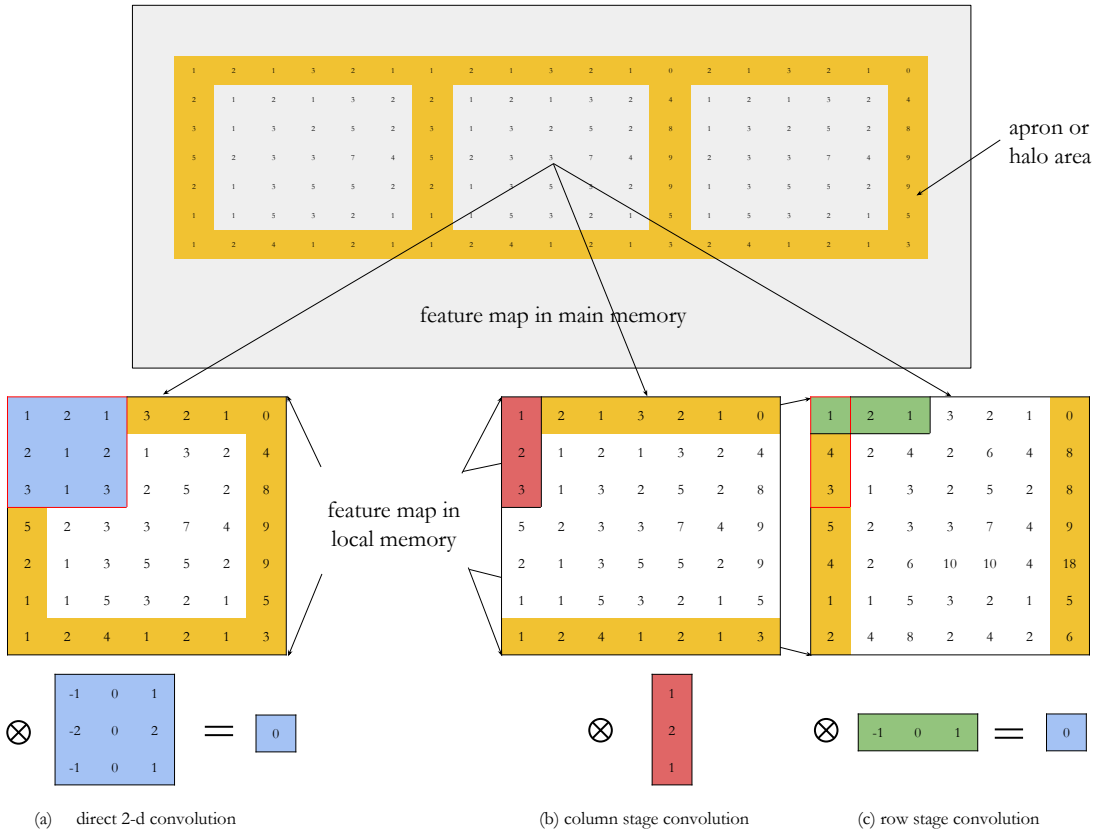



Fig. 3.9 (a) The original convolution with a $(m \times n)$ 2-D kernel. The yellow marked apron is all around the input tile. (b) The column-stage convolution using a $(m \times 1)$ column 1-D kernel. The apron is only at the top and bottom. (c) The row-stage convolution using a $(1 \times n)$ row 1-D kernel. The apron is only at the left and right.

longer necessary to load. This allows more efficient use of the available memory bandwidth and on-chip storage.

### 3.8.3 Summary

Convolutional neural networks that are lightweight both in terms of storage and compute cost and at the same time highly accurate in the classification task are of high demand in industry. In this chapter I have demonstrated that a correlation-based mathematically well-grounded technique can be used to speed-up deep networks by exploiting redundancies among feature maps in ConvNets. I started this chapter by introducing three key concepts, namely, redundancies in deep networks, efficient mapping primitives and separable filters. I then showed how theses three concepts can be combined to exploit the inherent redundancies present in modern ConvNets to yield a more optimised and hardware-friendly architecture. I have introduced ADaPT, an easy-to-implement three-step approximation scheme which can be applied on state-of-the-art pre-trained models statically. The availability of several pre-tuned models with different performance-accuracy targets can be a significant advantage for deploying ConvNets on fast time-to-market applications. Although my research is primarily aimed at embedded and IoT applications, this approximation scheme can also enable acceleration at semi-embedded (e.g. autonomous cars, military drones) or server-grade platforms. I have evaluated the ADaPT scheme on a variety of ConvNets targeting different class sizes and numbers of layers. The evaluation running on a range of real hardware provides strong evidence that kernel decomposition combined with approximation is a promising approach for speeding up pre-trained ConvNets without sacrificing accuracy significantly.

# Chapter 4

# Low-level lossless optimisation using fast arithmetic

In the previous chapter, I discussed a lossy compression scheme that helps to reduce the overall compute complexity of convolutional neural networks. In this chapter, I switch focus to a lossless scheme that involves use of fast arithmetic that helps to reduce the computational intensity of convolutional layers. In the first section, I provide a theoretical foundation of fast arithmetic schemes in general and, in particular, Cook-Toom and the modified version of the algorithm. I then show how to design such an algorithm from scratch using first principles. After the evaluation section, I conclude the chapter with an in-depth discussion on practical implementation strategies for the recommended algorithm.

## 4.1 Fast algorithm for convolutional neural network

In digital signal processing the best known technique for computing convolutions efficiently is to use FFT (Fast Fourier Transform) convolution. FFT convolution uses the principle that multiplication in the frequency domain corresponds to convolution in the time domain. Both the input signal and the filters are transformed into the frequency domain to be multiplied, and then transformed back into the time domain using the inverse DFT. The performance gain using this lossless technique is often satisfactory for most DSP applications. While a direct convolution has a computation complexity of $O(N^2)$, using FFT convolution one can arrive at the same result but with a complexity of $O(Nlog_2N)$. But when the filter length is small, FFT convolution is not very effective. In modern ConvNets most filters are small - $(1 \times 1)$, $(3 \times 3)$ or $(5 \times 5)$ being the most common ones. In this chapter, I show that computing convolution using small filters can be very efficient using a variant of the Cook-Toom algorithm that is

based on polynomial interpolation. Furthermore, the Cook-Toom-based strength-reduction algorithm is a lossless scheme, meaning that unlike model compression techniques there is no loss in model accuracy. This algorithm reduces the total number of strong operations (in this case multiplications) in convolutions at the expense of an increase in number of weaker operations, such as additions.

## 4.2 Cook-Toom algorithm for fast convolution

The Cook-Toom algorithm is based on the idea of evaluating polynomials at a few distinct chosen points and the Lagrange polynomial interpolation theorem [96]. The algorithm can be used for polynomial multiplication, long integer multiplication and convolution. Using the Cook-Toom method, a linear convolution can be expressed as a polynomial product of two polynomials

$$s(p) = w(p)x(p) \tag{4.1}$$

where

$$deg[x(p)] = N-1 \ , \ deg[w(p)] = L-1$$

Given the degrees of the input polynomials ($x(p) : input, w(p) : filter$) are known, the output polynomial $s(p)$ is of degree $L+N-2$ and has $L+N-1$ different coefficients. Let us assume $\beta_0$, $\beta_1$,...,$\beta_{L+N-2}$ are $L+N-1$ distinct real numbers that are chosen carefully. If we know $s(\beta_k)$ for $k = 0,...,L+N-2$, then we can compute the output polynomial $s(p)$ by Lagrange interpolation as follows

$$s(p) = \sum_{i=0}^{n-1} s(\beta_i) \frac{\prod_{j \neq i}(x-\beta_j)}{\prod_{j \neq i}(\beta_i - \beta_j)} \tag{4.2}$$

The interested reader can find more details on the *Lagrange Interpolating Polynomial* in Appendix A. In the above equation, $s(p)$ is a unique polynomial of degree $n-1$ that has the value $s(\beta_k)$ when $x$ takes the value $\beta_k$ for $k = 0,....,n-1$. The idea of the Cook-Toom algorithm is to first compute $s(\beta_k)$ for $k = 0,....,n-1$. and then use Lagrange interpolation. Figure 4.1 illustrates the main steps involved in the Cook-Toom algorithm.

The advantage of the Cook-Toom algorithm is that it requires fewer strong operations (i.e. multiplications) per convolution. These multiplications are computed in step 3 of the algorithm

$$s(\beta_k) = w(\beta_k)x(\beta_k) \tag{4.3}$$

More precisely, there are $L+N-1$ such equations, so there are $L+N-1$ multiplications here. If we pick $\beta_k$ carefully, those multiplications will be the only major strong operations. Therefore, the Cook-Toom algorithm will require $(L+N-1)^2$ multiplications compared to a direct implementation of the convolution, which will require $(N-L+1)\times(N-L+1)\times L\times L$ multiplications. As an example, using a $(3\times 3)$ filter in the $(6\times 6)$ input case as above, a direct implementation will require $4\times 4\times 3\times 3 = 144$ multiplications. But, using the above transform-based technique will require $(3+4-1)^2 = 36$ multiplications, resulting a 4x savings in compute complexity.

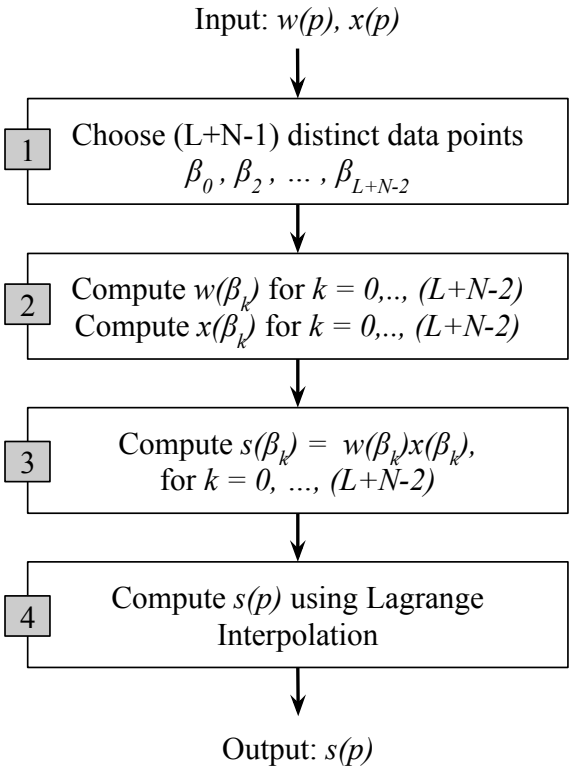



Fig. 4.1 Steps involved in the Cook-Toom algorithm

The Cook-Toom algorithm can also be expressed using compact matrix-vector notation as follows:

$$\boxed{(w*x)_{1D} = S^T[(Ww)\odot(X^T x)]} \tag{4.4}$$

$$\boxed{(w*x)_{2D} = S^T[(WwW^T)\odot(X^T xX)]S} \tag{4.5}$$

where $w$ is the kernel and $x$ is the input. In the 1D case (Equation 4.4) both are vectors, whereas in the 2D case (Equation 4.5) they are matrices. The final performance of the Cook-Toom algorithm not only depends on the number of distinct points using which the polynomials are constructed, but also on the exact values of the points. This in turn decides the quality of the $X, W$ and $S$ transform matrices.

**Algorithm 2:** Compute input, filter and output transformation matrices

**Function ComputeXWS** $(N, L, \beta^*)$;
**Input :** Output Length: N,
Kernel Length: L,
$\beta_1$, $\beta_2$, .., $\beta_n$ are distinct $n$ points
**Output :** Transformation matrices: $X^T$, $W$, $S$
$n = N + L - 1$
**for** $i \leftarrow 1$ ***to*** $n$ **do**
  $m^{(i)}(p) = symbolic(p - \beta_i)$;
**end**
$M(p) = \prod_i m^{(i)}(p)$; **for** $i \leftarrow 1$ ***to*** $n$ **do**
  $M^{(i)}(p) = symbolic(\frac{M(p)}{m^{(i)}(p)})$;
  $N^{(i)}(p) = \frac{1}{\prod_{j \neq i}(\beta_i - \beta_j)}$;
**end**
**for** $i \leftarrow 1$ ***to*** $N$ **do**
  **for** $j \leftarrow 1$ ***to*** $N + L - 1$ **do**
    $S^T[i][j] = (\beta_j)^{i-1}$;
  **end**
**end**
**for** $i \leftarrow 1$ ***to*** $n$ **do**
  **for** $j \leftarrow 1$ ***to*** $L$ **do**
    $W[i][j] = (\beta_i)^{j-1} * N^{(i)}$;
  **end**
**end**
**for** $i \leftarrow 1$ ***to*** $n$ **do**
  **for** $j \leftarrow 1$ ***to*** $n$ **do**
    $X^T[i][j] = M_j^{(i)}$;
  **end**
**end**

Algorithm (2) describes the construction process of these transformation matrices. To compute a one dimensional convolution of length $N$ with filter of length $L$, an input of length $N + L - 1$ is needed. In Algorithm (2) I assume $n$ distinct real points $\beta_1$, $\beta_2$, .., $\beta_n$. I then construct $n$ linear polynomials whose roots are given by $m^{(i)}(p) = symbolic(p -$

$\beta_i$). To compute the Lagrange interpolation matrix $M^{(i)}(p) = symbolic(\frac{M(p)}{m^{(i)}(p)})$ I compute polynomial $M(p) = \prod_i m^{(i)}(p)$ for $i = 1, ..., n$.

To construct the $S^T$ matrix, I compute its elements as 0 to $N-1$ powers of $N$ distinct points inside a for loop. This results in the $S^T$ matrix which is a transposed Vandermonde matrix of dimensions $(N+L-1) \times N$. By following a similar procedure I then construct the matrix $W$ of size $(N+L-1) \times L$. Each element of the matrix $W$ is obtained by scaling the Vandermonde matrices by $N^{(i)}$. To obtain the coefficients of $N^{(i)}$ I use the Euclidean algorithm (see Appendix A). I then construct matrix $W$ as the 0 to $L-1$ powers of selected points multiplied by the appropriate coefficient $N^{(i)}$. The input transformation matrix $X^T$ is obtained by the polynomial $M^{(i)}(p)$ coefficients. Note that the matrix $X^T$ is also a scaled inverse Vandermonde matrix. The final generalised form of transformation matrices $(X, W, S)$ obtained by the Cook-Toom algorithm is as follows:

$$S = \begin{bmatrix} 1 & \beta_1 & ... & \beta_1^{N-1} \\ 1 & \beta_2 & ... & \beta_2^{N-1} \\ ... & ... & ... & ... \\ 1 & \beta_n & ... & \beta_n^{N-1} \end{bmatrix} \tag{4.6}$$

$$W = \begin{bmatrix} 1 & \beta_1 * N^1 & ... & \beta_1^{L-1} * N^1 \\ 1 & \beta_2 * N^2 & ... & \beta_2^{L-1} * N^2 \\ ... & ... & ... & ... \\ 1 & \beta_n * N^n & ... & \beta_n^{L-1} * N^n \end{bmatrix} \tag{4.7}$$

$$X = \begin{bmatrix} M_0^1 & ... & M_0^n \\ ... & ... & ... \\ M_{n-1}^1 & ... & M_{n-1}^n \end{bmatrix} \tag{4.8}$$

## 4.3 Modification to the Cook-Toom algorithm to achieve further reduction in computation cost

The Cook-Toom algorithm helps to reduce the total number of multiplications at the expense of an increase in the number of additions. Tables 4.3 and 4.4 shows the reductions in multiplications per single output point achieved for different block sizes. Ideally, we can significantly reduce the number of multiplications by processing larger input blocks/tiles at once. But, this comes at the expense of a larger number of additions for each transform. Furthermore, with increased input block size the floating point error in the Cook-Toom algorithm grows exponentially. To address both of these issues with the increased number

of additions and the magnitude of the error, the Cook-Toom algorithm can be modified to reduce the total number of terms.

Let us assume we start with a sequence of length $n-1$ instead of the original $n$ (see Algorithm 2). Also, notice that the value of $s_{L+N-2}$ is given by $w_{L-1}x_{N-1}$. Now for the full-length convolution, we just have to compute only the last output element. Therefore, this coefficient can be computed with one multiplication. The modified polynomial

$$s(p) - s_{L+N-2}p^{L+N-2} = w(p)x(p) - s_{L+N-2}p^{L+N-2} \tag{4.9}$$

has a degree $L+N-3$ and can be computed using the Cook-Toom algorithm with $L+N-2$ multiplications. The extra multiplication $w_{L-1}x_{N-1}$ brings the total back to $L+N-1$ multiplications just as before, but fewer additions are required for this algorithm. In this approach we need $n-1$ distinct real points than the original $n$.

Therefore, in the modified version of the Cook-Toom algorithm, we can solve the smaller problem size of $n-1$ and at the end just add the missing values. To compute the Lagrange interpolation matrix the degree needs to be changed to $deg(M^{'}(p)) = deg(M(p)) - 1$. The final generalised form of the modified version of the Cook-Toom transformation matrices $(X,W,S)$ are obtained by filling the additional columns and rows by appropriate values:

$$S = \begin{bmatrix} 1 & \beta_1 & \dots & \beta_1^{N-1} \\ 1 & \beta_2 & \dots & \beta_2^{N-1} \\ \dots & \dots & \dots & \dots \\ 1 & \beta_n & \dots & \beta_{n-1}^{N-1} \\ 0 & 0 & \dots & 1 \end{bmatrix} \tag{4.10}$$

$$W = \begin{bmatrix} 1 & \beta_1 * N^1 & \dots & \beta_1^{L-1} * N^1 \\ 1 & \beta_2 * N^2 & \dots & \beta_2^{L-1} * N^2 \\ \dots & \dots & \dots & \dots \\ 1 & \beta_n * N^{n-1} & \dots & \beta_n^{L-1} * N^{n-1} \\ 0 & 0 & \dots & 1 \end{bmatrix} \tag{4.11}$$

$$X = \begin{bmatrix} M_0^1 & \dots & M_0^{n-1} & M_1 \\ \dots & \dots & \dots & \dots \\ M_{n-1}^1 & \dots & M_{n-1}^{n-1} & M_{n-1} \\ 0 & 0 & \dots & 1 \end{bmatrix} \tag{4.12}$$

## 4.4 Design of a modified Cook-Toom algorithm from first principles

Designing variations of the modified Cook-Toom algorithm that are hardware efficient requires careful optimisation. Often these optimisation choices heavily depend on the underlying architecture of the processing hardware. In this section, I show the design of a modified Cook-Toom convolution algorithm step-by-step in detail. As an example, I consider an input block size of $(6 \times 1)$ to be convolved with $(3 \times 1)$ 1D filters. This results in a $(4 \times 1)$ block as output and I denote this algorithm as $F(4 \times 1, 3 \times 1, \{6 \times 1\})$. Alternatively, I can also start with a $(4 \times 1)$ input block that results in a $\{6 \times 1\})$ output block and swap output and input transforms to obtain the same result, as shown in Equation (4.33). At the final part of this section, I show this particular swapping of transform steps using the matrix exchange theorem. Using this alternative approach I now compute the necessary transformation matrices, namely, $S$, $W$, and $X$.

Since I start with a $(4 \times 1)$ input block and a $(3 \times 1)$ filter, the corresponding polynomials can be written as follows:

$$w(p) = w_0 + w_1 p + w_2 p^2 \tag{4.13}$$

$$x(p) = x_0 + x_1 p + x_2 p^2 + x_3 p^3 \tag{4.14}$$

The polynomial corresponding to the output block can be then computed as follows:

$$s(p) = w(p)x(p) = s_0 + s_1 p + s_2 p^2 + s_3 p^3 + s_4 p^4 + s_5 p^5 \tag{4.15}$$

Since $L = 3$ and $N = 4$, the degree of the modified polynomial is given by $L + N - 3 = 4$. Let us choose the following distinct points, namely $\beta_0 = 0, \beta_1 = 1, \beta_2 = -1, \beta_3 = 2$, and $\beta_4 = -2$. Now, let us calculate individual $w(\beta_k)$ and $x(\beta_k)$ as follows:

$$\beta_0 = 0, \;\; w(\beta_0) = w_0, \;\; x(\beta_0) = x_0 \tag{4.16}$$

$$\beta_1 = 1, \;\; w(\beta_1) = w_0 + w_1 + w_2, \;\; x(\beta_1) = x_0 + x_1 + x_2 + x_3 \tag{4.17}$$

$$\beta_2 = -1, \;\; w(\beta_2) = w_0 - w_1 + w_2, \;\; x(\beta_2) = x_0 - x_1 + x_2 - x_3 \tag{4.18}$$

$$\beta_3 = 2, \;\; w(\beta_3) = w_0 + 2w_1 + 4w_2, \;\; x(\beta_3) = x_0 + 2x_1 + 4x_2 + 8x_3 \tag{4.19}$$

$$\beta_4 = -2, \;\; w(\beta_4) = w_0 - 2w_1 + 4w_2, \;\; x(\beta_3) = x_0 - 2x_1 + 4x_2 - 8x_3 \tag{4.20}$$

According to the modified version of the Cook-Toom algorithm, the polynomial of degree $(L+N-3)$ now can be expressed as follows

$$s^{'}(\beta_0) = w(\beta_0)x(\beta_0) - w_2x_3\beta_0^5 = w(\beta_0)x(\beta_0) \tag{4.21}$$
$$s^{'}(\beta_1) = w(\beta_1)x(\beta_1) - w_2x_3\beta_1^5 = w(\beta_1)x(\beta_1) - w_2x_3 \tag{4.22}$$
$$s^{'}(\beta_2) = w(\beta_2)x(\beta_2) - w_2x_3\beta_2^5 = w(\beta_2)x(\beta_2) + w_2x_3 \tag{4.23}$$
$$s^{'}(\beta_3) = w(\beta_3)x(\beta_3) - w_2x_3\beta_3^5 = w(\beta_3)x(\beta_3) - 32w_2x_3 \tag{4.24}$$
$$s^{'}(\beta_4) = w(\beta_4)x(\beta_4) - w_2x_3\beta_4^5 = w(\beta_4)x(\beta_4) + 32w_2x_3 \tag{4.25}$$

Using Lagrange polynomial interpolation we obtain the following expression for the intermediate polynomial

$$\begin{aligned} s^{'}(p) = s^{'}(\beta_0)&\frac{(p-\beta_1)(p-\beta_2)(p-\beta_3)(p-\beta_4)}{(\beta_0-\beta_1)(\beta_0-\beta_2)(\beta_0-\beta_3)(\beta_0-\beta_4)} \\ &+ s^{'}(\beta_1)\frac{(p-\beta_0)(p-\beta_2)(p-\beta_3)(p-\beta_4)}{(\beta_1-\beta_0)(\beta_1-\beta_2)(\beta_1-\beta_3)(\beta_1-\beta_4)} \\ &+ s^{'}(\beta_2)\frac{(p-\beta_0)(p-\beta_1)(p-\beta_3)(p-\beta_4)}{(\beta_2-\beta_0)(\beta_2-\beta_1)(\beta_2-\beta_3)(\beta_2-\beta_4)} \\ &+ s^{'}(\beta_3)\frac{(p-\beta_0)(p-\beta_1)(p-\beta_2)(p-\beta_4)}{(\beta_3-\beta_0)(\beta_3-\beta_1)(\beta_3-\beta_2)(\beta_3-\beta_4)} \\ &+ s^{'}(\beta_4)\frac{(p-\beta_0)(p-\beta_1)(p-\beta_2)(p-\beta_3)}{(\beta_4-\beta_0)(\beta_4-\beta_1)(\beta_4-\beta_2)(\beta_4-\beta_3)} \end{aligned} \tag{4.26}$$

The above equation can be simplified further and can be re-arranged in the polynomial form as follows

$$\begin{aligned} s^{'}(p) = s^{'}(\beta_0) &+ p(\frac{4}{6}s^{'}(\beta_1) - \frac{4}{6}s^{'}(\beta_2) - \frac{2}{24}s^{'}(\beta_3) + \frac{2}{24}s^{'}(\beta_4)) \\ &+ p^2(-\frac{5}{4}s^{'}(\beta_0) + \frac{4}{6}s^{'}(\beta_1) + \frac{4}{6}s^{'}(\beta_2) - \frac{1}{24}s^{'}(\beta_3) - \frac{1}{24}s^{'}(\beta_4)) \\ &+ p^3(-\frac{1}{6}s^{'}(\beta_1) + \frac{1}{6}s^{'}(\beta_2) + \frac{2}{24}s^{'}(\beta_3) - \frac{2}{24}s^{'}(\beta_4)) \\ &+ p^4(\frac{1}{4}s^{'}(\beta_0) - \frac{1}{6}s^{'}(\beta_1) - \frac{1}{6}s^{'}(\beta_2) + \frac{1}{24}s^{'}(\beta_3) + \frac{1}{24}s^{'}(\beta_4)) \end{aligned} \tag{4.27}$$

Since I am using the modified version of the Cook-Toom algorithm, we can get back the final $s(p)$ by using the following equation

$$s(p) = s^{'}(p) + w_2 x_3 p^5 \tag{4.28}$$

Finally, we have the output in matrix form by replacing all $\beta_k$ in the previous equation,

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & -4 & -2 & 2 & 4 \\ -5 & 4 & 4 & -1 & -1 & 0 \\ 0 & -1 & 1 & 2 & -2 & -5 \\ 1 & -1 & -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} W_0 & 0 & 0 & 0 & 0 & 0 \\ 0 & W_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & W_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & W_3 & 0 & 0 \\ 0 & 0 & 0 & 0 & W_4 & 0 \\ 0 & 0 & 0 & 0 & 0 & W_5 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 2 & 4 & 8 \\ 1 & -2 & 4 & -8 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{4.29}$$

where

$$\begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \\ W_4 \\ W_5 \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \\ \frac{1}{6} & -\frac{1}{6} & \frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} \tag{4.30}$$

The Cook-Toom algorithm can be viewed as a method of factoring matrices and can be expressed as the following form ($\odot$ denotes element-wise multiplication):

$$s = X[(Ww) \odot (Sx)] \tag{4.31}$$

We can transpose this solution for a larger block size using the matrix exchange theorem from linear algebra. According to the matrix exchange theorem, if we have a matrix M which can be factored as:

$$s = XDS \tag{4.32}$$

where D is a diagonal matrix, then it can also be factored as:

$$s = (\bar{S})^T D(\underline{X})^T \tag{4.33}$$

where $\bar{S}$ is the matrix obtained from $S$ by reversing the order of its columns, and $\underline{X}$ is the matrix obtained from $X$ by reversing the order of its rows. We can now apply the same on our final equation and have an alternative form as follows:

$$s = S^T[(Ww) \odot (X^T x)] \tag{4.34}$$

Finally, we obtain the transformation matrices $S^T$, $X^T$, and $W$ from Equations (4.35), (4.36), and (4.37) respectively.

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix} \begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \end{bmatrix} \tag{4.35}$$

where

$$\begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \end{bmatrix} = \begin{bmatrix} W_0 & 0 & 0 & 0 & 0 & 0 \\ 0 & W_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & W_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & W_3 & 0 & 0 \\ 0 & 0 & 0 & 0 & W_4 & 0 \\ 0 & 0 & 0 & 0 & 0 & W_5 \end{bmatrix} \odot \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & -1 & 0 \\ 0 & 4 & -4 & -1 & -1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_6 \end{bmatrix} \tag{4.36}$$

and

$$\begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \\ W_4 \\ W_5 \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \\ \frac{1}{6} & -\frac{1}{6} & \frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} \tag{4.37}$$

The Cook-Toom algorithm reduces the total number of multiplications required in convolution operations at the expense of an increase in number of additions. As an example, in the algorithm F(4×1, 3×1, {6×1}), when used in 2D, 36 multiplications are required, while in the direct convolution 144 multiplications are required. This results in a computational complexity reduction of 4. However, the 2D input-transform requires 144 additions, the 2D filter-transform requires 72 additions and the output-transform requires 100 additions. Note that, many of those additions can be implemented using shift operations which are

cheap. Furthermore, the filter-transforms can be pre-computed and stored in memory. The number of additions required for computing the output-transforms can be further reduced by a multi-channel implementation scheme. This particular optimisation scheme to reduce the cost of output-transforms is introduced in Chapter 5.

## 4.5 Variants of the modified Cook-Toom algorithm

In this section I show a few other variations of the modified Cook-Toom algorithm that I used in the implementation and experiments. For brevity, I only provide the transformation matrices $X$, $W$, and $S$ in this report and omit the detailed derivations.

### 4.5.1 Variants of the modified Cook-Toom algorithm for 3x1 filters

The most common filter size used in modern ConvNet architectures is $3 \times 1$. This section provides a number of alternative realisations of the modified Cook-Toom algorithm.

#### 4.5.1.1 Algorithm F(4×1, 3×1, {6×1})

This variation of the Cook-Toom algorithm uses 8 multiplications compared to 18 multiplications in the direct implementation, resulting in an arithmetic intensity reduction of 2.25×:

$$W = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix}; \quad X = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & -4 & 4 & -2 & 2 & 4 \\ -5 & -4 & -4 & -1 & -1 & 0 \\ 0 & 1 & -1 & 2 & -2 & -5 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}; \quad S = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 2 & 4 & 8 \\ 1 & -2 & 4 & -8 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.38}$$

#### 4.5.1.2 Algorithm F(2×1, 3×1, {4×1})

This variation of the Cook-Toom algorithm uses 6 multiplications compared to 12 multiplications in the direct implementation, resulting in an arithmetic intensity reduction of

2×:

$$W = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}; \qquad X = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}; \qquad S = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 1 \end{bmatrix} \tag{4.39}$$

#### 4.5.1.3 Algorithm F(3×1, 3×1, {5×1})

This variation of the Cook-Toom algorithm uses 7 multiplications compared to 15 multiplications in the direct implementation, resulting in an arithmetic intensity reduction of 2.14×:

$$W = \begin{bmatrix} \frac{1}{2} & 0 & 0 \\ -\frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{6} & \frac{1}{3} & \frac{2}{3} \\ 0 & 0 & 1 \end{bmatrix}; \qquad X = \begin{bmatrix} 2 & -1 & -2 & 1 & 0 \\ 0 & -2 & -1 & 1 & 0 \\ 0 & 2 & -3 & 1 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 \end{bmatrix}; \qquad S = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & 0 \\ 0 & 1 & 1 & 4 & 1 \end{bmatrix} \tag{4.40}$$

#### 4.5.1.4 Algorithm F(6×1, 3×1, {8×1})

This variation of the Cook-Toom algorithm uses 10 multiplications compared to 24 multiplications in the direct implementation, resulting in an arithmetic intensity reduction of 2.4×:

$$W = \begin{bmatrix} \frac{1}{36} & 0 & 0 \\ \frac{1}{48} & \frac{1}{48} & \frac{1}{48} \\ \frac{1}{48} & -\frac{1}{48} & \frac{1}{48} \\ -\frac{1}{120} & -\frac{1}{60} & -\frac{1}{30} \\ -\frac{1}{120} & \frac{1}{60} & -\frac{1}{30} \\ \frac{1}{720} & \frac{1}{240} & \frac{1}{80} \\ \frac{1}{720} & -\frac{1}{240} & \frac{1}{80} \\ 0 & 0 & 1 \end{bmatrix} \qquad X = \begin{bmatrix} 36 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 36 & -36 & 18 & -18 & 12 & -12 & -36 \\ -49 & 36 & 36 & 9 & 9 & 4 & 4 & 0 \\ 0 & -13 & 13 & -20 & 20 & -15 & 15 & 49 \\ 14 & -13 & -13 & -10 & -10 & -5 & -5 & 0 \\ 0 & 1 & -1 & 2 & -2 & 3 & -3 & -14 \\ -1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.41}$$

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 2 & 4 & 8 & 16 & 32 \\ 1 & -2 & 4 & -8 & 16 & -32 \\ 1 & 3 & 9 & -27 & 81 & -243 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.42}$$

## 4.5.2 Variants of the modified Cook-Toom algorithm for 5x1 and 7x1 filters

The other most common filters used in modern ConvNets are $5 \times 1$ and $7 \times 1$. This section provides a number of alternative realisations of the modified Cook-Toom algorithm.

### 4.5.2.1 Algorithm F(4×1, 5×1, {8×1})

This variation of the Cook-Toom algorithm uses 12 multiplications compared to 40 multiplications in the direct implementation, resulting in an arithmetic intensity reduction of 3.33×:

$$W = \begin{bmatrix} \frac{1}{36} & 0 & 0 & 0 & 0 \\ \frac{1}{48} & \frac{1}{48} & \frac{1}{48} & \frac{1}{48} & \frac{1}{48} \\ \frac{1}{48} & -\frac{1}{48} & \frac{1}{48} & -\frac{1}{48} & \frac{1}{48} \\ -\frac{1}{120} & -\frac{1}{60} & -\frac{1}{30} & -\frac{1}{15} & -\frac{2}{15} \\ -\frac{1}{120} & \frac{1}{60} & -\frac{1}{30} & \frac{1}{15} & -\frac{2}{15} \\ \frac{1}{720} & \frac{1}{240} & \frac{1}{80} & \frac{3}{80} & \frac{9}{80} \\ \frac{1}{720} & -\frac{1}{240} & \frac{1}{80} & -\frac{3}{80} & \frac{9}{80} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad X = \begin{bmatrix} 36 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 36 & -36 & 18 & -18 & 12 & -12 & -36 \\ -49 & 36 & 36 & 9 & 9 & 4 & 4 & 0 \\ 0 & -13 & 13 & -20 & 20 & -15 & 15 & 49 \\ 14 & -13 & -13 & -10 & -10 & -5 & -5 & 0 \\ 0 & 1 & -1 & 2 & -2 & 3 & -3 & -14 \\ -1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.43}$$

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 2 & 4 & 8 \\ 1 & -2 & 4 & -8 \\ 1 & 3 & 9 & -27 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.44}$$

#### 4.5.2.2 Algorithm F(2×1, 7×1, {8×1})

To design this variant of the algorithm I use the points $\{0,-1,1,-2,2,-3,3,\infty\}$. This algorithm uses 14 multiplications compared to 56 multiplications in the direct implementation, resulting in an arithmetic intensity reduction of 4×:

$$W = \begin{bmatrix} \frac{1}{36} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{48} & \frac{1}{48} & \frac{1}{48} & \frac{1}{48} & \frac{1}{48} & \frac{1}{48} & \frac{1}{48} \\ \frac{1}{48} & -\frac{1}{48} & \frac{1}{48} & -\frac{1}{48} & \frac{1}{48} & -\frac{1}{48} & \frac{1}{48} \\ -\frac{1}{120} & -\frac{1}{60} & -\frac{1}{30} & -\frac{1}{15} & -\frac{2}{15} & -\frac{4}{15} & -\frac{8}{15} \\ -\frac{1}{120} & \frac{1}{60} & -\frac{1}{30} & \frac{1}{15} & -\frac{2}{15} & \frac{4}{15} & -\frac{8}{15} \\ \frac{1}{720} & \frac{1}{240} & \frac{1}{80} & \frac{3}{80} & \frac{9}{80} & \frac{27}{80} & \frac{81}{80} \\ \frac{1}{720} & -\frac{1}{240} & \frac{1}{80} & -\frac{3}{80} & \frac{9}{80} & -\frac{27}{80} & \frac{81}{80} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad X = \begin{bmatrix} 36 & 0 & -49 & 0 & 14 & 0 & -1 & 0 \\ 0 & 36 & 36 & -13 & -13 & 1 & 1 & 0 \\ 0 & -36 & 36 & 13 & -13 & -1 & 1 & 0 \\ 0 & 18 & 9 & -20 & -10 & 2 & 1 & 0 \\ 0 & -18 & 9 & 20 & -10 & -2 & 1 & 0 \\ 0 & 12 & 4 & -15 & -5 & 3 & 1 & 0 \\ 0 & -12 & 4 & 15 & -5 & -3 & 1 & 0 \\ 0 & -36 & 0 & 49 & 0 & -14 & 0 & 1 \end{bmatrix} \tag{4.45}$$

$$S = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 3 & -3 & 1 \end{bmatrix} \tag{4.46}$$

#### 4.5.2.3 Algorithm F(2×1, 7×1, {8×1})

To design this variant of the algorithm I use the points $\{0, -\frac{1}{2}, \frac{1}{2}, -1, 1, -2, 2, \infty\}$. This algorithm uses 14 multiplications compared to 56 multiplications in the direct implementation resulting in an arithmetic intensity reduction of 4×:

$$W = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{2}{9} & -\frac{2}{9} & -\frac{2}{9} & -\frac{2}{9} & -\frac{2}{9} & -\frac{2}{9} & -\frac{2}{9} \\ -\frac{2}{9} & \frac{2}{9} & -\frac{2}{9} & \frac{2}{9} & -\frac{2}{9} & \frac{2}{9} & -\frac{2}{9} \\ \frac{1}{90} & \frac{1}{45} & \frac{2}{45} & \frac{4}{45} & \frac{8}{45} & \frac{16}{45} & \frac{32}{45} \\ \frac{1}{90} & -\frac{1}{45} & \frac{2}{45} & -\frac{4}{45} & \frac{8}{45} & -\frac{16}{45} & \frac{32}{45} \\ \frac{32}{45} & \frac{16}{45} & \frac{8}{45} & \frac{4}{45} & \frac{2}{45} & \frac{1}{45} & \frac{1}{90} \\ \frac{32}{45} & -\frac{16}{45} & \frac{8}{45} & -\frac{4}{45} & \frac{2}{45} & -\frac{1}{45} & \frac{1}{90} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}; \quad X = \begin{bmatrix} 1 & 0 & -\frac{21}{4} & 0 & \frac{21}{4} & 0 & -1 & 0 \\ 0 & 1 & 1 & -\frac{17}{4} & -\frac{17}{4} & 1 & 1 & 0 \\ 0 & -1 & 1 & \frac{17}{4} & -\frac{17}{4} & -1 & 1 & 0 \\ 0 & \frac{1}{2} & \frac{1}{4} & -\frac{5}{2} & -\frac{5}{4} & 2 & 1 & 0 \\ 0 & -\frac{1}{2} & \frac{1}{4} & \frac{5}{2} & -\frac{5}{4} & -2 & 1 & 0 \\ 0 & 2 & 4 & -\frac{5}{2} & -5 & \frac{1}{2} & 1 & 0 \\ 0 & -2 & 4 & \frac{5}{2} & -5 & -\frac{1}{2} & 1 & 0 \\ 0 & -1 & 0 & \frac{21}{4} & 0 & -\frac{21}{4} & 0 & 1 \end{bmatrix}; \quad (4.47)$$

$$S = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & \frac{1}{2} & -\frac{1}{2} & 1 \end{bmatrix} \quad (4.48)$$

## 4.6 Combining lossy and lossless schemes

In Chapter 3, I demonstrated how ADaPT, a low-rank approximation and layer-restructuring scheme, can help to reduce inherent redundancies in ConvNets. I also showed in such a lossy compression scheme how accuracy can be traded for performance. However, this scheme suits the case where training data is available in addition to the trained model. In this chapter, I introduced the modified Cook-Toom algorithm, which reduces the number of multiplications needed to compute convolution. The advantage of this scheme is that it reduces overall arithmetic complexity without needing to change the model architecture. An interesting question arises, whether these two schemes can be combined to leverage the benefits from both, provided the training data is available to fine-tune any loss in accuracy due to the approximation stage.

In this section, I demonstrate that these two lossy and lossless schemes can indeed be combined in a single pipeline of optimisation. This scheme is called 1D-FALCON, and Figure 4.2 shows a high-level functional diagram of the overall optimisation pipeline. The upper half of the diagram shows stage 1 that consists of the approximation scheme that is lossy. The lower half of the diagram shows stage 2 that consists of the modified Cook-Toom fast convolutional scheme. In stage 1, pre-trained ConvNets are analysed, approximated, and restructured to reduce redundancy in the model. This stage can be done offline. The

model definition and the related parameters (filter weights) are then passed to stage 2. The filters can be transformed offline and stored as they don't change during inference. Input images are then streamed and transformed using the Cook-Toom transformation matrices. The transformed images and the filters are then multiplied element-wise (a.k.a. Hadamard product) to produce a partial result. The final result is obtained after the application of the inverse or output transformation.

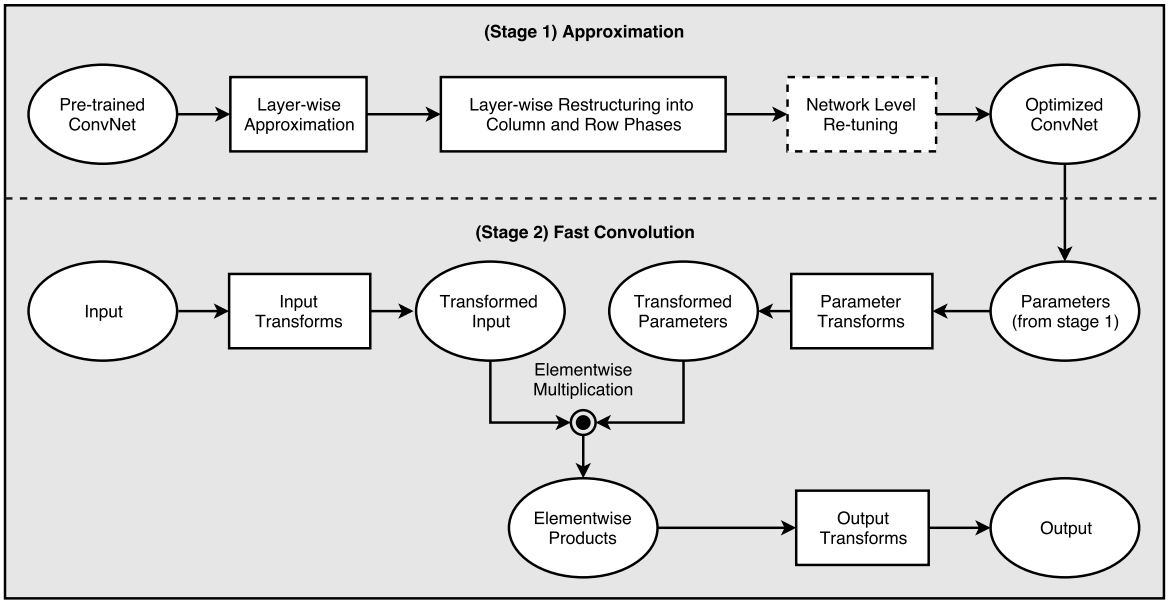



Fig. 4.2 1D-FALCON: The high-level optimisation pipeline consists of two main stages - (1) a lossy approximation stage, (2) a lossless fast convolution stage

## 4.7 Evaluation

In order to evaluate the effectiveness of the 1D-FALCON scheme I applied this two-stage pipeline in AlexNet, VGG-16, Inception-v1 and ResNet-152 networks on a Nvidia Titan-X GPU. The details of the platform configuration can be found in Table 3.1 in Chapter 3. I used four main metrics to measure the effectiveness of the scheme:

- **MULs:** Total number of strong operations (i.e., multiplications) in the convolutional layers
- **Speedup:** Total speedup achieved as compared to baseline 2D convolution
- **Top-5 Error:** The top-5 error rate is the percentage of test examples for which the correct class was not in the top 5 predicted classes

- **Fine-Tuning Time:** Average fine-tuning time in number of epochs. Fine-tuning is the process of re-training a CNN after having trained it once and then having reduced its complexity. An epoch is a complete pass through the training set.

Table 4.1 shows these four metrics for selected ConvNets as described earlier. As can be seen from the results, the 1D-FALCON scheme helps to speed up convolutions significantly without sacrificing in prediction accuracy.

Table 4.1 Speedup achieved using the 1D-FALCON scheme

| CNNs | #MULs (Millions) | Speed-Up | Top-5 Error (%) | Fine-Tuning Time |
|---|---|---|---|---|
| AlexNet [56] | 692 | 12.1× | 19.8 | 1 epoch |
| VGG-16 [82] | 15,300 | 11.4× | 9.5 | 1–2 epochs |
| Inception-v1 [88] | 1,428 | 7.2× | 10.7 | 3 epochs |
| ResNet-152 [44] | 11,300 | 6.2× | 5.3 | 2–3 epochs |

From the previous experiment, we learned that 1D-FALCON is an effective pipeline scheme that helps to reduce the overall computation required for convolution. To demonstrate the effectiveness of this scheme, I also compare the 1D-FALCON scheme with a wide variety of speedup techniques. I chose a direct 2D convolutional scheme [82], a low-rank scheme based on Tucker decomposition [55], two popular pruning techniques [41, 73], a sparsification scheme [60], and the 2D Winograd filtering scheme [58]. I chose the VGG-16 model for this experiment, not only because it was the winner of the ImageNet challenge in 2014 [82] but also because this model is used extensively in many model optimisation research works. The results from different schemes are easily available to be compared with this scheme. VGG-16 is a deep architecture and consists of 13 convolutional layers out of a total of 16 layers.

As can be seen from Table 4.2, the 1D-FALCON scheme achieves significant speedup compared to other schemes and does not require a long fine-tuning time. The overall speedup comes from combined application of both the low-rank approximation scheme and the fast 1D convolution technique using the modified Cook-Toom algorithm.

### 4.7.1 Comparison of multiplication intensity

A number of variants of the modified Cook-Toom algorithm can be designed and implemented. One may choose to use a different number of distinct points based on the target architecture (e.g. on chip storage size, layout). The number of distinct points also depends on the input and output tile size. The speedup will be different for different numbers of distinct points. Tables 4.3 and 4.4 show the theoretical reduction in multiplication intensity using

Table 4.2 Comparison of speedup of the VGG-16 network using different schemes, *Optimisations schemes that are part of this PhD

| Optimization Scheme | #MULs | Speedup | Top-5 Error (%) | Fine-Tuning Time |
|---|---|---|---|---|
| 2D Convolution [82] | 15.3G | 1.0× | 9.4 | None |
| Group-Wise Sparsification [60] | 7.6G | 2.0× | 10.1 | >10 epochs |
| Winograd's Filtering [58] | 6.8G | 2.26× | 9.4 | None |
| Iterative Pruning [73] | 4.5G | 3.4× | 13.0 | 60 epochs |
| Modified Cook-Toom Algorithm* [70] | 4.3G | 3.6× | 9.4 | None |
| Pruning+Retraining [41] | 3.0G | 5.0× | 10.88 | 20–40 epochs |
| ADaPT* [68] | 2.7G | 5.7× | 9.5 | 1-2 epochs |
| Tucker Decomposition [55] | 3.0G | 5.0× | 11.60 | 5–10 epochs |
| 1D FALCON* [This scheme] | 1.3G | 11.4× | 9.5 | 1–2 epochs |

the modified Cook-Toom implementation compared to a direct implementation for varying numbers of points and filter sizes 3 (1D) and 5 (1D) respectively.

### 4.7.2 Floating-point errors in the Cook-Toom algorithm

The Cook-Toom-based scheme is only lossless when infinite precision is used. Therefore, in a real implementation, all variations of the Cook-Toom algorithm are prone to some level of floating-point error. The magnitude of the error depends on a number of factors that includes choice of points, order of computations, tile size, and number of channels in a layer. More generally, floating-point errors occur because real numbers are approximated using a limited-precision number system. Each real number is mapped to its nearest floating-point equivalent and this rounding process leads to errors. Additionally, if the absolute value of the real number is smaller or greater than the largest allowed representable floating-point number, then this causes catastrophic loss of accuracy due to underflow or overflow. In ConvNets, underflow and overflow are rare when sufficient precision and range are used. Floating-point error also can occur during polynomial interpolation using the Lagrange method. If the set of points on which the polynomial is interpolated are not constant then error can arise while computing matrix inversion. In contrast, the modified Cook-Toom algorithm is an exact scheme of computing convolution. We always choose the distinct points for interpolation at design time. Therefore, in the Cook-Toom convolution scheme no error occurs due to polynomial interpolation.

The three main components of error in the Cook-Toom convolution are contributed by the transformation matrices, the input signal and the filters, and channel-accumulation error. As more points are used to compute a larger block of input, the floating-point approximation error grows. Figures 4.3 and 4.4 show that with an increased number of points, although the

Table 4.3 Comparison of multiplication intensity (i.e. number of strong operations per output point) between direct and the Cook-Toom convolution for varying output dimensions for filter size 3 (1D) and 3x3 (2D)

| number of points | output size for w = 3 (1D) | multiplication intensity | output size for w = 3×3 (2D) | multiplication intensity |
|---|---|---|---|---|
| 0 | 1 | 3 | 1×1 | 9 |
| 4 | 2 | 2 | 2×2 | 4 |
| 5 | 3 | 1.67 | 3×3 | 2.78 |
| 6 | 4 | 1.5 | 4×4 | 2.25 |
| 7 | 5 | 1.4 | 5×5 | 1.96 |
| 8 | 6 | 1.34 | 6×6 | 1.78 |
| 9 | 7 | 1.29 | 7×7 | 1.65 |
| 10 | 8 | 1.25 | 8×8 | 1.56 |

Table 4.4 Comparison of multiplication intensity (i.e. number of strong operations per output point) between direct and the Cook-Toom convolution for varying output dimensions for filter size 5 (1D) and 5×5 (2D)

| number of points | output size for w = 5 (1D) | multiplication intensity | output size for w = 5x5 (2D) | multiplication intensity |
|---|---|---|---|---|
| 0 | 1 | 5 | 1×1 | 25 |
| 6 | 2 | 3 | 2×2 | 9 |
| 7 | 3 | 2.33 | 3×3 | 5.44 |
| 8 | 4 | 2 | 4×4 | 4 |
| 9 | 5 | 1.8 | 5×5 | 3.24 |
| 10 | 6 | 1.67 | 6×6 | 2.78 |
| 11 | 7 | 1.57 | 7×7 | 2.47 |
| 12 | 8 | 1.5 | 8×8 | 2.25 |

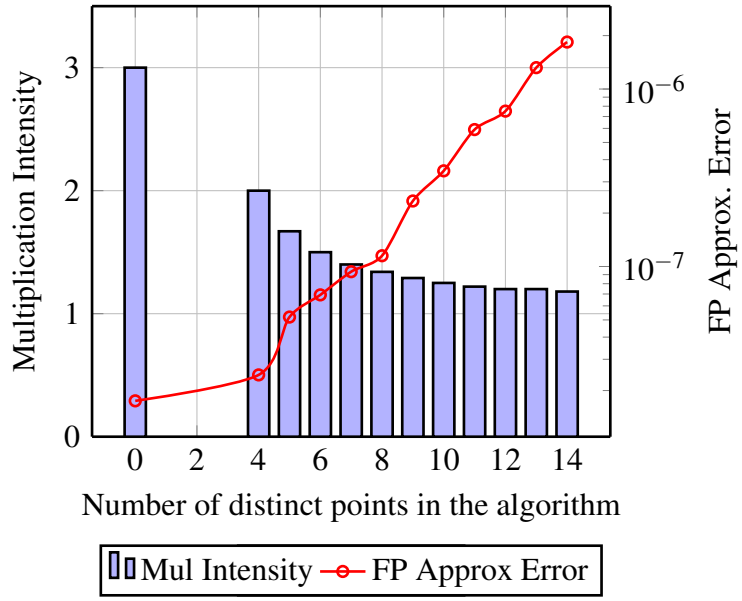



Fig. 4.3 As the number of points increases multiplication intensity (i.e. number of multiplications per output point) drops while floating-point error grows (example: 1D convolution with filter size of 3)

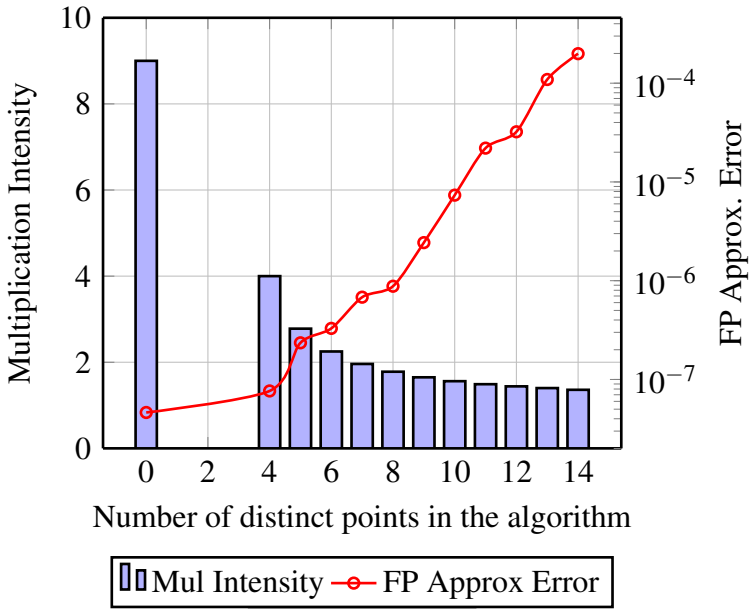



Fig. 4.4 As the number of points increases multiplication intensity (i.e. number of multiplications per output point) drops while floating-point error grows (example: 2D convolution with filter size of $3 \times 3$)

compute intensity drops, the floating-point error grows very fast. In ConvNets, a number of channels/feature maps are used in each layer to capture different patterns in the inputs. While computing convolutions, the intermediate results are summed from many channels and error arises during the accumulation process. Figure 4.5 shows that as more channels are accumulated the channel-accumulation error grows.

#### 4.7.2.1 Floating-point error measurement

To understand the contribution of floating-point error in the Cook-Toom algorithm I make empirical measurements. I first obtain the $X$, $W$, and $S$ Cook-Toom transformation matrices using the chosen block size $n = N + L - 1$ in infinite precision fractions. Since these transformation matrices are constant matrices we can compute them accurately before using them in the convolution. I then convert these matrices to the nearest representable floating-point number. I model the filter and input-pixel values as a uniform random distribution in the range $(-1, 1)$. I compute the error as the difference between the result of the convolution and an approximation of the numerically correct result. I calculate the reference result of the convolution using the direct convolution algorithm using 64-bit double-precision floating-point numbers and the approximation using 32-bit floating-point numbers. Since the input pixels and the filters are taken from the uniform-random distribution, I compute the average error arising from 1000 tests. Figures 4.3 and 4.4 show the magnitude of the floating-point error for different variations of the Cook-Toom algorithm using different block sizes. We can see that for both the one-dimensional and two-dimensional cases, the floating-point errors grow with the block size, which is proportional to the number of distinct points used in the Cook-Toom algorithm. Figure 4.5 shows the floating-point error due to channel accumulation. The error grows with number of channels or feature maps being accumulated. In addition, the error is higher for any variation of the algorithm that uses larger block sizes.

## 4.8 Discussion

Before I conclude this chapter, in this section I highlight a few important points related to practical implementations. The first and foremost topic of interest is how we can reduce floating-point error in the Cook-Toom algorithm. Then I compare the modified Cook-Toom algorithm with FFT and Winograd-based solutions. Finally, I mention a couple of limitations of the Cook-Toom algorithm in real-life scenarios.

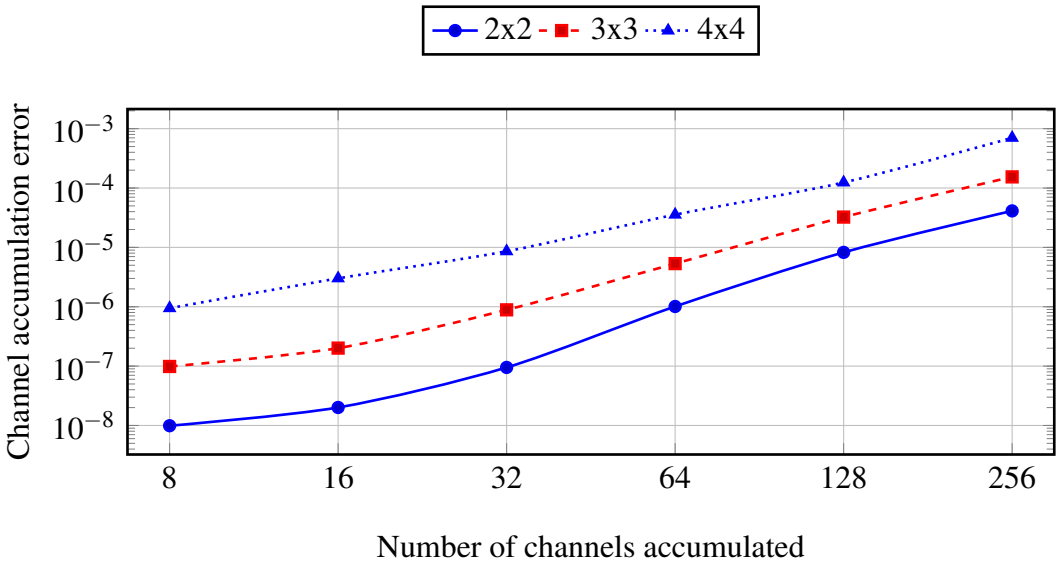



Fig. 4.5 Channel-accumulation error for different numbers of feature maps

### 4.8.1 Design strategies to reduce floating point error

In the previous section I highlighted the main error components of the Cook-Toom algorithm. In this section I focus on strategies that address the floating-point error. While constructing transformation matrices for the algorithm one may use the following list of strategies as a design guide.

Firstly, use the modified version of the Cook-Toom algorithm instead of the original Cook-Toom algorithm. As fewer distinct points are used in the modified version, the error of the modified version is significantly smaller.

Secondly, the selection of distinct points is crucial in reducing the floating-point error. The points $\{0,-1,1\}$ are generally good candidates that reduce the cost of implementation. Often these multiplications can be implemented by addition, which is very efficient. For multiplication by zero we can simply allow both scaling and addition to be skipped altogether. For large problems, small integers or simple fractions are good choices for reducing the required number of scalings and additions. The points $\{2,-2,\frac{1}{2},-\frac{1}{2}\}$ are a few points that yield simple transformation matrices and thus reduce floating-point error. For even larger problems, a set of potentially good points for the numerator in the fractions are $\{-3,-2,-1,0,1,2,3\}$. For the denominator the positive part of the same selection results in a reduction in floating-point error. Using the same tile size, one can arrive at different transformation matrices and thus different variants of the algorithm. For example, I show two different variations of the algorithm F($2\times1$, $7\times1$, $\{8\times1\}$) for tile size 8 and filter size

7. The algorithm in Section 4.5.2.3 is less prone to floating-point error than the one in Section 4.5.2.2 due to careful selection of points.

Thirdly, in floating-point computation the ordering of evaluations gives rise to different amounts of floating-point error. During the pre- or post-transformation the floating-point error varies based on the order of evaluation. Theoretically, it is difficult to suggest which order is the best for reducing error. A simple principle to follow is to sum smaller values first before moving on to larger numbers. Unfortunately, this is not always possible in a hardware implementation as we have further restrictions on the ordering of evaluation based an on-chip and off-chip storage layout.

### 4.8.2 Modified Cook-Toom versus the FFT algorithm

It is a common practice in digital signal processing to use FFT-based convolution. Similar to the Cook-Toom convolution, a tiled convolutional algorithm can also be constructed using the FFT algorithm. The main difference is that the transformation matrices $X$, $W$, and $S$ are replaced with the FFT and inverse FFT.

In an FFT-based convolution-layer implementation the transform overhead is much larger than that of the Cook-Toom based implementation. For example, to match the complexity of the Cook-Toom algorithm with a tile size of $4 \times 4$, an FFT-based implementation requires a tile size of $64 \times 64$. Also, in feature maps with sizes that are not close to a multiple of $64 \times 64$, many pixels will waste computations. Even with moderate-sized feature maps, there will be too few tiles to compute the FFT efficiently. The memory overhead is also significantly larger for FFT-based implementations. Using the same $64 \times 64$ FFT example, a $3 \times 3$ filter must first be expanded to $64 \times 64$. Effectively $3 \times 3$, or 9 values are expanded to $64 \times 64$, or 4096 values which is prohibitively large. FFT-based convolutional-layer implementations must have a large on-chip memory to hold the transformed data. For small filter sizes, the overhead of multiple kernel launches, streaming data in and out of memory multiple times, and zero padding to the input size outweigh the algorithmic advantage of FFT-based convolution. With larger filter sizes (e.g. $13 \times 13$), FFT tends to become advantageous. As modern convolutional neural nets are dominated by small filters, the Cook-Toom-based implementation is the preferred choice for acceleration.

### 4.8.3 Modified Cook-Toom versus the Winograd algorithm

Both the Winograd and the Cook-Toom schemes belong to a class of strength-reduction algorithm, where the number of strong operations is reduced at the expense of an increase in the number of weaker operations. Nevertheless, the underlying theory behind both

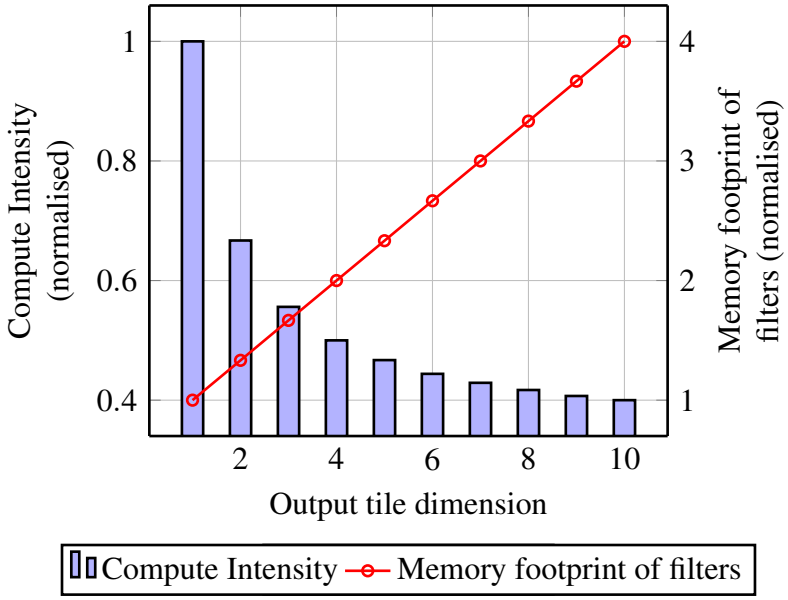



Fig. 4.6 With larger tile/block size, the cost of intermediate filter-memory footprint grows (Example: 1D convolution with filter size of 3)

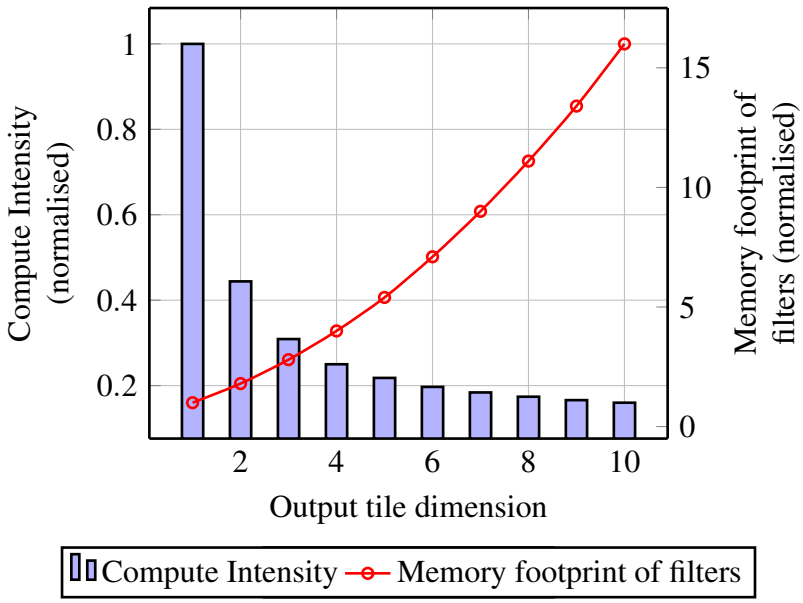



Fig. 4.7 With larger tile/block size, the cost of intermediate filter-memory footprint grows (Example: 2D convolution with filter size of $3 \times 3$)

these algorithms is different. The Winograd convolution algorithm is based on the Chinese remainder theorem over an integer ring. The central idea behind this theorem is that a non-negative integer can be uniquely determined only from its remainders with respect to pre-selected moduli, provided that the moduli are relatively prime. Hence, the design of Winograd's convolution algorithm heavily depends on modulus or remainder arithmetic, which are complex to derive. In contrast, the Cook-Toom, or the modified version of the algorithm, are based on the Lagrange polynomial interpolation theorem. Given pre-selected distinct points and the function values at those points, it is possible to reconstruct the polynomial using Lagrange interpolation. Hence, the convolution can be expressed in terms of multiplication of two polynomials. For convolution involving small filters, the design of transformation matrices using the Cook-Toom algorithm is less cumbersome and more straight forward. A number of variants can be easily designed to suit the target architecture using alternative distinct points.

In convolutional neural networks, each input feature map is convolved with many filters and generates new intermediate feature maps. As a result, the cost of pre-processing the input, filter transformations, and post-processing the output transformations, is amortised across many uses. Furthermore, the filters can be pre-processed at design time and pre-loaded in to the on-chip memory. Post-processing the output results needs to be done only once per layer. Thus, while applying strength reduction algorithm in ConvNet, we are primarily interested in the reduction of complexity of the general multiplications in the inner loop. Winograd's convolution algorithm offers trade-offs between post/pre-processing operations and general multiplications in the inner loop. In contrast, Cook-Toom and the modified version of the algorithm help us to construct solutions that minimise the number of general multiplications, potentially at the cost of more pre-/post-processing operations. Therefore, convolutions in modern ConvNets can benefit from variants of the Cook-Toom algorithm rather than the Winograd algorithm. For larger problems involving slightly larger filters, one can iterate on small variants of the Cook-Toom convolution algorithm by using nesting convolution techniques and achieve a significant reduction in general multiplication costs in the inner loop.

### 4.8.4 Limitations of the modified Cook-Toom algorithm

Figures 4.6 and 4.7 show that as we implement algorithm variants with larger tile sizes the computational intensity continues to drop significantly. But, there is also a side-effect of selecting an algorithm with a larger tile. In the intermediate stage, the filter dimension must match the larger tile dimension. Thus, the memory footprint of the parameters becomes significant for the underlying hardware, as shown in the right-hand axis of the figure. In

the most realistic implementation of the algorithm, an optimum choice between speed and storage costs must be made. Additionally, as I discussed earlier, floating-point error also grows with larger tile sizes. Therefore, the main limitation of the modified Cook-Toom algorithm is that although theoretically larger tile sizes lead to further compute reduction, in practice we cannot choose tile sizes beyond an optimum point.

### 4.8.5 Summary

The performance of modern convolutional neural networks can be significantly improved by modifying the low-level linear-algebra arithmetic. A fast arithmetic scheme using a modified Cook-Toom algorithm is lossless and can be applied on the majority of networks without changing the model architecture. Furthermore, in this chapter, I demonstrated that we can achieve further reduction in compute by combining a lossy and lossless scheme in a pipeline. I started with a theoretical foundation of Cook-Toom and a modified version of the algorithm. I then explained a detailed step-by-step design of such an algorithm from first principles. A number of variants of the modified Cook-Toom algorithm were introduced that are efficient when aiming for a low-power implementation. In the evaluation section, I demonstrated the benefit of the 1D-FALCON scheme on a number of modern ConvNets. I also compared the scheme with other prominent compute-reduction schemes. The results from these experiments show that the 1D-FALCON scheme helps to reduce compute complexity of modern pre-trained ConvNets without sacrificing accuracy by a significant amount.

# Chapter 5

# Implementation of the Cook-Toom class of fast convolution on a widely used mobile processor

In the previous chapter, I showed how a lossless fast-arithmetic scheme can reduce the overall compute complexity of the convolution layer in modern ConvNets. ARM based SoCs are ubiquitous in today's mobile computing. While many smartphone chip manufacturers are adding dedicated machine-learning acceleration units, the CPU is still the default processor for much of the AI workloads [5]. Furthermore, there are many scenarios, especially in low-power IoT (Internet of Things) solutions, where Arm-based CPUs are the only computing core that are available for performing all the on-device processing. Can the fast algorithm developed in the previous chapter be used to implement fast inference directly on an Arm processor? Unlike direct convolution, computing convolutions using the modified Cook-Toom algorithm requires a different data-processing pipeline as it involves pre- and post-transformations of the intermediate activations. This chapter solely focuses on an efficient implementation of the Cook-Toom class of fast convolution on the Arm architecture. I start this chapter with a multi-channel extension to the already existing algorithm to suit the underlying implementation scheme. I then briefly introduce the target Armv8-A architecture and necessary data layout details for SIMD computation. In the following section, I present details of the multi-channel multi-region (MCMR) implementation of the Cook-Toom algorithm. In the evaluation section, I compare the performance for both the individual convolutional layers and for the end-to-end models. After the evaluation section, I conclude the chapter with an in-depth discussion on limitations of implementing the Cook-Toom type of fast algorithm on an Arm processor. The majority of this research work was done in collaboration with Arm's machine-learning (ML) research group. I developed the

necessary Cook-Toom algorithm and MCMR implementations. Andrew Mundy helped to run the benchmark using the Arm compute library. The remaining authors helped us to review the draft of the paper which was later published in the IEEE HPCA-EMC2 workshop [69].

## 5.1 Multi-channel implementation of the modified Cook-Toom algorithm

The Cook-Toom convolution algorithm described in the last chapter can be adapted to accelerate the layers within convolutional neural networks (CNNs). Until this point I have shown how to apply the Cook-Toom convolution on individual feature maps or channels. But modern ConvNets consist of multiple feature maps per layer, which is an important property. Most modern deep ConvNets have something between 3 and 512 feature maps or channels in each layer. When performing convolution, separate convolution is performed at each feature map and then the results of each separate convolution are summed with the corresponding values in the other feature maps. The straight forward way to implement this summation across feature maps is to perform the complete convolution separately on each feature map and accumulate the result after the inverse transform as shown in Figure 5.1. However, this would require application of inverse-transformation on intermediate results with respect to each feature map, which is a relatively expensive operation. For example, a layer with 512 channels would require computing 512 inverse transformations. A better way to process these convolutions per layer would be to sum the intermediate results from all feature maps and then apply the inverse transformation operation only on the final sum. In this manner the required number of output transformations can be reduced significantly. The proposed flow of computing the Cook-Toom convolution by accumulating feature maps is shown in Figure 5.2.

To be more concrete mathematically, let us assume in a convolutional neural network each layer consists of a total of $C$ input channels each with dimension $H \times W$. We can split each input channel into many $i \times i$ tiles and apply the algorithm repeatedly with respective $w \times w$ filters:

$$f = \sum_{c=0}^{C} S^T[(WwW^T)_c \odot (X^T xX)_c]S \tag{5.1}$$

The straightforward implementation using Equation 5.1 will incur very high cost during the repeated inverse transformation for each input channel. We can altogether avoid doing the

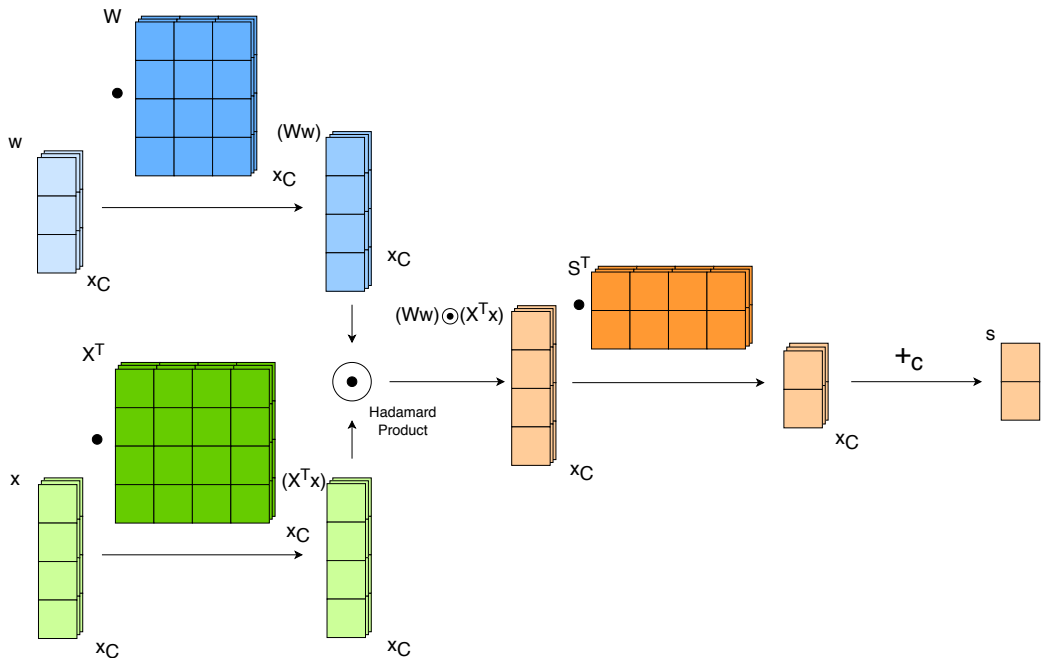



Fig. 5.1 Channel-by-channel implementation of the Cook-Toom convolution

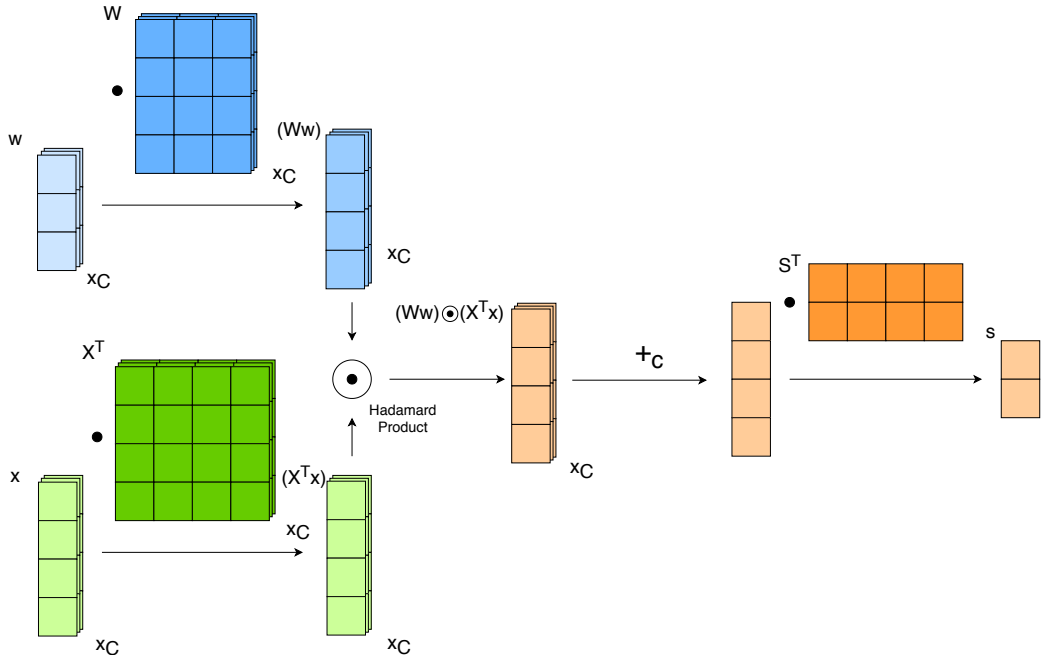



Fig. 5.2 Multi-channel implementation of the Cook-Toom convolution

repeated inverse transforms by using the following

$$f = S^T(\sum_{c=0}^{C}[(WwW^T)_c \odot (X^T xX)_c])S \tag{5.2}$$

Using Equation 5.2, the partial Hadamard products can be accumulated until all the input channels are ready. After the final sum is available, the inverse transform can be computed once to obtain the final result. This would altogether amortise the cost of inverse transform computation over the number of input channels. Later, I show how the MCMR (multi-channel multi-region) algorithm for a SIMD machine is built up on this basic idea.

## 5.2 An overview of the Armv8-A SIMD architecture

In the previous section, I proposed a multi-channel implementation of the modified Cook-Toom algorithm that helps to reduce computation in the post-processing phase. We are now ready to dive into the details of implementation on a target hardware. To this end I have selected the Arm-v8 architecture, as Arm-based platforms are standard in the low-power embedded and mobile industry. Firstly, I highlight the features of Arm's SIMD architecture and then I show how to combine the techniques developed previously to build an efficient Cook-Toom convolution implementation on an Arm CPU.

ARMv8-A.x is ARM's latest architecture, which supports 64-bit operations and introduces two execution modes: AArch32 and AArch64. NEON instructions are specifically used for SIMD (Single Instruction Multiple data) data processing. In this regime, instructions operate on vectors of elements of the same data type. The data types may be floating point or integer. In a typical SIMD operation, the operand registers are treated as vectors of individual elements. The operations are performed simultaneously on a number of lanes. Figure 5.3 shows an example of SIMD operation. In this example each lane contains a pair of 32-bit elements, one from each of the operand's 128-bit vector registers.

### 5.2.1 Data layout for SIMD computation

There are a variety of ways in which the tensors in deep convolutional neural networks can be arranged in memory. Two common options are called *NCHW* and *NHWC* [77] – where $N$ stands for the number of batches (or concurrent inferences), $C$ for the number of channels, and $H$ and $W$ stand for height and width, respectively (see Figure 5.5). In *NCHW* each plane of the tensor is stored contiguously in memory – i.e., pixel $(n,c,i,j)$ is followed by $(n,c,i,j+1)$ – whereas in *NHWC* all of the channels of a given pixel are stored

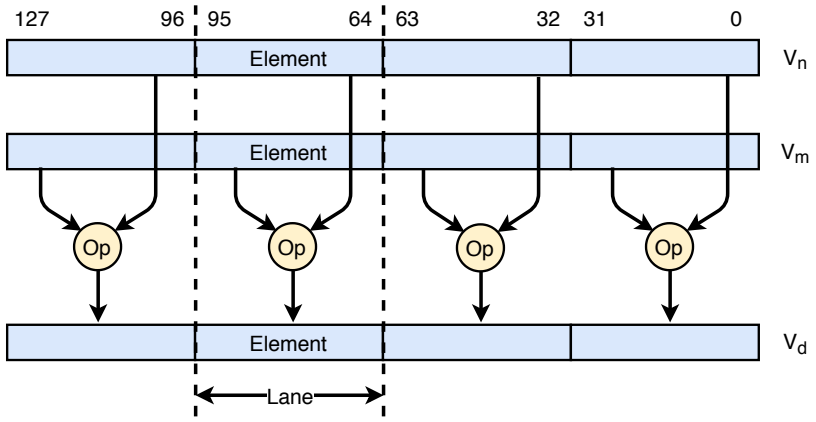



Fig. 5.3 Armv8-A SIMD processing

contiguously (i.e., value $(n,i,j,c)$ is followed by $(n,i,j,c+1)$). For scalar code, ignoring cache locality, the choice of tensor ordering is largely immaterial. However, when writing vectorised (SIMD) code, tensor ordering is crucial to achieving performance.

In the ARMv8-A architecture, there are thirty-two 128-bit SIMD registers. Each SIMD register can, therefore, store four 32-bit single-precision values. Hence, under *NCHW* a single SIMD register will store, after a 128-bit load, a row of four pixels, whereas under *NHWC* the same register would store four channels of data for a single pixel.

## 5.3 MCMR: Multi-channel multi-region Cook-Toom algorithm

First, I note that the algorithms derived from Equation 5.1 applied a $w\times w$ filter to only a small input region of size $x\times x$ to produce an output region of size $z\times z$. To perform larger convolutions, therefore, the input tensor must be divided into multiple regions of size $x\times x$. The output tensor must also be divided into an equivalent number of regions, each of which is computed as the element-wise (Hadamard) multiplication and accumulation of the corresponding input regions (representing $C$ input channels) with their respective weight tiles.

For example, a $6\times 6$ input tensor would be divded into a $2\times 2$ array of $4\times 4$ regions. Separate arrays of regions are produced for each channel in a tensor. Hence, a $3\times 6\times 6$ tensor would produce a $3\times 2\times 2$ array of regions. Convolving this input tensor with $M$ filters would result in the construction of an $M\times 2\times 2$ array of regions. Output regions are computed as described above: for example, the upper-leftmost output region in the $m^{\text{th}}$ channel results

from the summation of $C$ products – each of the $C$ channels of the upper-leftmost input region with the respective tiles from the $m^{\text{th}}$ filter. This algorithm is illustrated in Listing 5.1.

```
// For each output channel
for (unsigned int m = 0; m < M; m++)
     // For each output region
     for (unsigned int r = 0; r < R; r++)
          // Summation across the input channels
          for (unsigned int c = 0; c < C; c++)
               output_region[m, r] += HadamardProduct(
                    input_region[c, r], weight_region[m,c]
               );
```

Listing 5.1 The Cook-Toom convolution algorithm

The MCMR algorithm is divided into four stages:

1. **Input transform** Progresses over regions of the input tensor, pre-processes them using the input transforms and *scatters* the results into the 'A' matrices for the GEMMs.

2. **Filter transform** Progresses over regions of the weight tensor, pre-processes them using the filter transforms and *scatters* the results into the 'B' matrices for the GEMMs. Since parameters of a deep network are known after training, the filter transforms are typically performed prior to inference offline.

3. **GEMM** *Multiplies* the 'A' matrices generated in the *input transform* with 'B' (matrices generated when the weights were pre-processed using the weight transforms) to form the 'C' matrices.

4. **Output transform** Repeatedly *gathers* regions of values from the 'C' matrices, transforms them back from the spatial domain and writes the results into the output tensor.

### 5.3.1 Input transformation

In this section, I start by demonstrating the effect of tensor orderings through the example of implementing the input transform for $F(2\times2,3\times3,4\times4)$. The characteristic equation for this transform is:

$$X^{\top}xX = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} x \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 \\ -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

Under *NCHW* ordering, four registers are needed to store $x$, a $4\times4$ region of the input tensor. The transform matrices could be hard-coded as a series of row transformations, such that computing $X^\top x$ could be expressed as shown on Listing 5.2:

```
// Compute X^Tx using a series of row-transformations
{
        XTx[0] = vsubq_f32(x[0], x[1]);
        XTx[1] = vaddq_f32(x[1], x[2]);
        XTx[2] = vsubq_f32(x[2], x[1]);
        XTx[3] = vsubq_f32(x[1], x[3]);
}
```

Listing 5.2 Input transforms (1D)

By transposing the result, this code sequence can be repeated such that we compute $\left(X^\top\left((X^\top x)^\top\right)\right)^\top = X^\top xX$. Once this is completed we have 16 values (four registers containing four values each) which must be scattered, as described above, to 16 separate matrices.

In contrast, under *NHWC* ordering, sixteen SIMD registers are needed to represent four channels of a $4\times4$ region of the input tensor. The transformation can be hardcoded, but in this case it can operate on four channels of data simultaneously, as in Listing 5.3. Once the transformation is complete we are left with 16 registers, each containing four channels worth of data. These registers can be scattered directly into the input matrices for the GEMMs.

```
// Compute X^T x
for (int j = 0; j < 4; j++) {
        XTx[0][j] = vsubq_f32(x[0][j], x[2][j]);
        XTx[1][j] = vaddq_f32(x[1][j], x[2][j]);
        XTx[2][j] = vsubq_f32(x[2][j], x[1][j]);
        XTx[3][j] = vsubq_f32(x[1][j], x[3][j]);
}
// Compute U = (X^T x) X
for (int i = 0; i < 4; i++) {
        U[i][0] = vsubq_f32(XTx[i][0], XTx[i][2]);
        U[i][1] = vaddq_f32(XTx[i][1], XTx[i][2]);
        U[i][2] = vsubq_f32(XTx[i][2], XTx[i][1]);
        U[i][3] = vsubq_f32(XTx[i][1], XTx[i][3]);
}
```

Listing 5.3 Input transforms (2D)

### 5.3.2 Choice to vector orders: NCHW vs NHWC

For the specific instance of $F(2\times2,3\times3,4\times4)$ there are merits to both approaches. However, when either different data widths (such as half-precision floating point) or different variants

of the Cook-Toom algorithms are considered, the advantages of the *NHWC* ordering become more prominent.

For example, although I can use four SIMD registers to represent 16 values *in single-precision floating point* in *NCHW* – four values to a register – this breaks down when I move to half-precision and each register can contain eight values. The code sample from above for *NCHW* would need to be extensively rewritten to deal with this case, whereas the *NHWC* code could be simply modified to work on eight channels of data simultaneously.

Likewise, were I to implement the input transform for $F(4\times4, 3\times3, 6\times6)$, which requires use of $6\times6$ input regions, I could, in *NHWC* ordering use 36 values (and the stack) to represent each input region. However, in *NCHW* I would need to use one-and-a-half registers to represent each row of six values. Again, the code would be considerably more complicated. For these reasons, I prefer the use of *NHWC*-ordered data.

### 5.3.3 Filter or weight transformation

Typically, filter transforms follow a similar pattern to input transforms, except the filters are smaller in dimension. Each region of the input transform must match the filter dimension for maximum reuse of the on-chip data. Furthermore, parameters of a deep convolutional network are tuned during the training process and they are frozen once the training is complete. To reduce the cost of inference, the filter transforms are pre-processed and stored in memory offline. During inference, the transformed parameters are loaded from the on-chip memory and convolved with the input stream. Listings 5.5 and 5.6 show sample code for filter transformations used in the MCMR algorithm.

### 5.3.4 Using GEMM to compute Hadamard product

The fundamental operation in the basic convolution algorithm illustrated in Listing 5.1 is an element-wise multiply-accumulate (element-wise addition of Hadamard products). Secondly, there are two axes in which data is reused. Specifically,

1. Weight tile $(m, c)$ is used across all input regions in layer $c$.
2. Input region $(c, i, j)$ contributes to all $M$ output regions at $(i, j)$.

These observations – that the fundamental operation is multiply-accumulate and that there are two axes of reuse – suggest that one way of implementing a complete convolution is to leverage the GEMM (General Matrix Matrix Multiplication) algorithm. Use of GEMM is attractive since there exists a wide range of good GEMM implementations (e.g. libraries such

```
#ifdef __aarch64__
  for (; channels_remaining >= 4; channels_remaining -= 4) {
    // Matrices used/computed in this kernel.
    float32x4_t x[inner_tile_i][inner_tile_j];
    float32x4_t XTx[inner_tile_i][inner_tile_j];
    float32x4_t U[inner_tile_i][inner_tile_j];
    //initialize x, XTx
    // Load x
    for (int i = pad_top; i < cells_i; i++) {
      for (int j = pad_left; j < cells_j; j++) {
        x[i][j] = vld1q_f32(x_ptrs[i][j]);
        x_ptrs[i][j] += 4;
      }
    }
    // Compute XT . x
    for (int j = pad_left; j < cells_j; j++) {
      // XTx[0][j] = x[0][j] - x[2][j];
      XTx[0][j] = vsubq_f32(x[0][j], x[2][j]);
      // XTx[1][j] = x[1][j] + x[2][j];
      XTx[1][j] = vaddq_f32(x[1][j], x[2][j]);
      // XTx[2][j] = x[2][j] - x[1][j];
      XTx[2][j] = vsubq_f32(x[2][j], x[1][j]);
      // XTx[3][j] = x[1][j] - x[3][j];
      XTx[3][j] = vsubq_f32(x[1][j], x[3][j]);
    }
    // Compute U = XT . x . X
    for (int i = 0; i < inner_tile_i; i++) {
      // U[i][0] = XTx[i][0] - XTx[i][2];
      U[i][0] = vsubq_f32(XTx[i][0], XTx[i][2]);
      // U[i][1] = XTx[i][1] + XTx[i][2];
      U[i][1] = vaddq_f32(XTx[i][1], XTx[i][2]);
      // U[i][2] = XTx[i][2] - XTx[i][1];
      U[i][2] = vsubq_f32(XTx[i][2], XTx[i][1]);
      // U[i][3] = XTx[i][1] - XTx[i][3];
      U[i][3] = vsubq_f32(XTx[i][1], XTx[i][3]);
    }
    // Store the transformed matrix
    for (int i = 0, m = 0; i < inner_tile_i; i++) {
      for (int j = 0; j < inner_tile_j; j++, m++) {
        vst1q_f32(outptr + m*matrix_stride, U[i][j]);
      }
    }
    outptr += 4;
  }
#endif // __aarch64__
```

Listing 5.4 Input transforms (2D)

```
#ifdef __aarch64__

for (; channels_remaining >= 4; channels_remaining -= 4) {

    // Matrices used and computed in this kernel

    float32x4_t w[3][3], Ww[6][3], V[6][6];

    // Read weights

    for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 3; j++) {
        w[i][j] = vld1q_f32(inptrs[i][j]);
        inptrs[i][j] += 4;
      }
    }

    // Compute the matrix W w
    for (int j = 0; j < 3; j++) {
      // Ww[0][j] = 6*w[0][j];
      Ww[0][j] = vmulq_n_f32(w[0][j], 6.0);

      // Ww[1][j] = -4*w[0][j] + -4*w[1][j] + -4*w[2][j];
      Ww[1][j] = vmulq_n_f32(vaddq_f32(vaddq_f32(w[0][j], w[1][j]), w[2][j]),
          -4.0);

      // Ww[2][j] = -4*w[0][j] + 4*w[1][j] + -4*w[2][j];
      Ww[2][j] = vmulq_n_f32(vsubq_f32(vsubq_f32(w[1][j], w[0][j]), w[2][j]),
          4.0);

      // Ww[3][j] = 1*w[0][j] + 2*w[1][j] + 4*w[2][j];
      Ww[3][j] = vmlaq_n_f32(vmlaq_n_f32(w[0][j], w[1][j], 2.0f), w[2][j],
         4.0f);

      // Ww[4][j] = 1*w[0][j] + -2*w[1][j] + 4*w[2][j];
      Ww[4][j] = vmlaq_n_f32(vmlsq_n_f32(w[0][j], w[1][j], 2.0f), w[2][j],
         4.0f);

      // Ww[5][j] = 24*w[2][j];
      Ww[5][j] = vmulq_n_f32(w[2][j], 24.0f);
    }

    // Continued..

#endif // __aarch64__
```

Listing 5.5 Filter transforms (2D)

```
#ifdef __aarch64__

//... Continuation from previous page

    // Compute V = W w WT

    for (int i = 0; i < 6; i++) {
      const float recip576 = 1.0f / 576.0f;

      // V[i][0] = 6*Ww[i][0];
      V[i][0] = vmulq_n_f32(vmulq_n_f32(Ww[i][0], 6.0), recip576);

      // V[i][1] = -4*Ww[i][0] + -4*Ww[i][1] + -4*Ww[i][2];
      V[i][1] = vmulq_n_f32(vmulq_n_f32(vaddq_f32(vaddq_f32(Ww[i][0], Ww[i
          ][1]), Ww[i][2]), -4.0), recip576);

      // V[i][2] = -4*Ww[i][0] + 4*Ww[i][1] + -4*Ww[i][2];
      V[i][2] = vmulq_n_f32(vmulq_n_f32(vsubq_f32(vsubq_f32(Ww[i][1], Ww[i
          ][0]), Ww[i][2]), 4.0), recip576);

      // V[i][3] = 1*Ww[i][0] + 2*Ww[i][1] + 4*Ww[i][2];
      V[i][3] = vmulq_n_f32(vmlaq_n_f32(vmlaq_n_f32(Ww[i][0], Ww[i][1], 2.0f),
           Ww[i][2], 4.0f), recip576);

      // V[i][4] = 1*Ww[i][0] + -2*Ww[i][1] + 4*Ww[i][2];
      V[i][4] = vmulq_n_f32(vmlaq_n_f32(vmlsq_n_f32(Ww[i][0], Ww[i][1], 2.0f),
           Ww[i][2], 4.0f), recip576);

      // V[i][5] = 24*Ww[i][2];
      V[i][5] = vmulq_n_f32(vmulq_n_f32(Ww[i][2], 24.0f), recip576);
    }

    // Store the transformed weights

    for (int i = 0, m = 0; i < 6; i++) {
      for (int j = 0; j < 6; j++, m++) {
        vst1q_f32(outptr + m*matrix_stride, V[i][j]);
      }
    }
    outptr += 4;
}
#endif // __aarch64__
```

Listing 5.6 Filter transforms (2D)

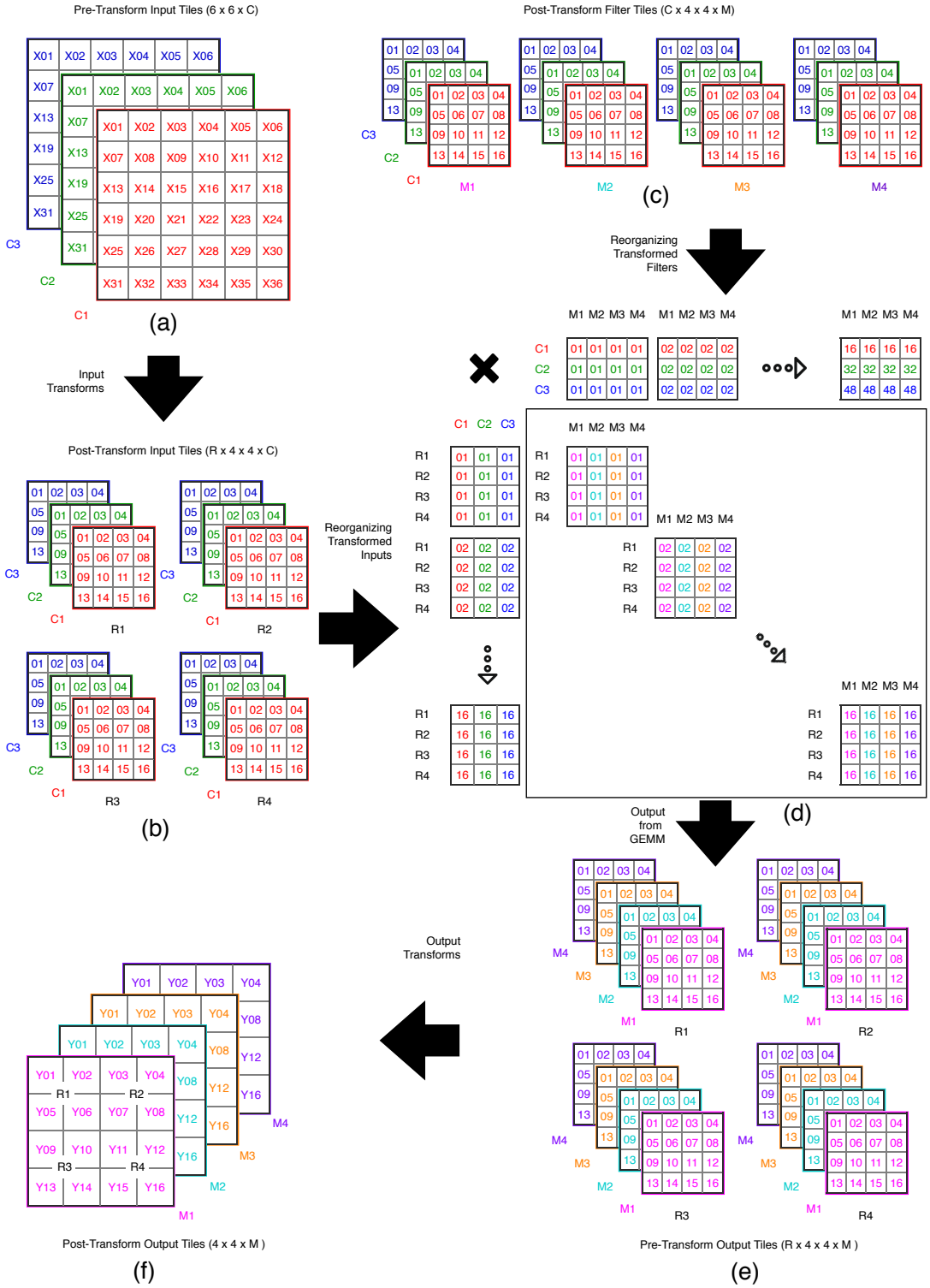



Fig. 5.4 High-level diagram of the MCMR Cook-Toom scheme

as BLASFEO [33]) capable of exploiting the SIMD instructions of the Armv8-A architecture. I now describe how the algorithm illustrated in Listing 5.1 can be mapped to GEMM.

Firstly, since the key data-processing operation is *element-wise* multiply-accumulate, each cell of input, weight and output regions is treated independently of the others. This suggests that for a Cook-Toom algorithm with a $x \times x$ input region size $x^2$ independent GEMMs must be performed.

Next, each output cell is the result of $C$ multiply-accumulates – this suggests that the inner dimension of our matrix multiplication must be $C$ (where $C$ is the number of channels in the input tensor).

Finally, except for the number of channels, the input and output region arrays are the same size – this suggests that one dimension of each GEMM should be the number of regions in a layer, $R$, and the final dimension of the GEMM the number of output channels, $M$. Consequently, the convolution of a tensor consisting of $C$ input layers and $R$ regions with a $M$ deep set of filters can be expressed as $x^2$ GEMMs of the form $[R \times C] \times [C \times M]$ or $[M \times C] \times [C \times R]$. For reasons which will be explained below, I choose the former. In this form the first matrix is a representation of the input tensor and must be computed for each inference and the latter represents the weights and can be computed once and reused. Figure 5.4 illustrates this mapping.

Shown in Figure 5.4(a) is the example from above: a $3 \times 6 \times 6$ tensor being convolved with four filters. In (b) this tensor is split into a $3 \times 2 \times 2$ array of regions, each of which is pre-processed using the input transforms. Meanwhile, in (c), a set of four filters (each, necessarily, of three channels) is shown pre-processed using the filter transforms. In (d), these weights are combined with the regions from (b) to form the GEMM structure described above. Specifically, an array of 16 GEMMs of size $[R \times C] \times [C \times M]$ is constructed, with the input tensor being represented by the first set of matrices and the weights by the latter. Performing the GEMMs results in the creation of 16 $[R \times M]$ matrices which are reordered, in (e), into regions of output. In (f) these regions are converted back to the spatial domain and so arranged as to form the result of the convolution. In practice, some of the steps described above can be combined.

### 5.3.5 Efficient tensor ordering for ARMv8 core

I noted earlier that the matrix multiplications used within the MCMR algorithm could be arranged as either $[R \times C] \times [C \times M]$ or $[M \times C] \times [C \times R]$, and that, of these, I preferred the former as shown in Figure 5.4. This selection follows directly from my choice of *NHWC* tensor ordering. Specifically, under *NHWC*, each SIMD register contains multiple channels of data and these values must be written into matrices of shape $R \times C$ or $C \times R$. Assuming row-

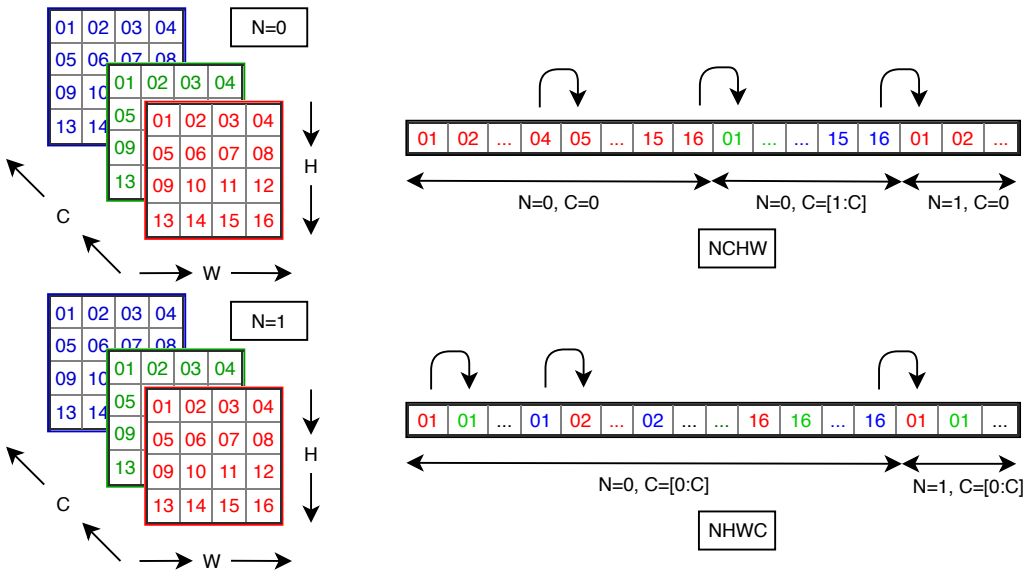



Fig. 5.5 NCHW vs NHWC layout

major ordered matrices, in the latter case, I could use multi-element structured stores (e.g., ST4 (single structure), [2]) to combine and store values from different registers. Alternatively, an unstructured store (STR [2]) could be used to write out a whole register into successive columns of an $R \times C$ matrix. Unlike structured stores, which are vector-store instructions for multiple elements, unstructured stores only store a single scalar-value at a time. Structured stores are slower than unstructured stores as multiple elements must be temporarily held before they are written to destinations. Since unstructured stores to have a higher throughput than their structured counterparts I chose to use the first matrix arrangement.

### 5.3.6 Output transformation

As shown in Figure 5.4, GEMM produces intermediate products that need to be converted back to the spacial domain. The key to the MCMR algorithm is the accumulation of multiple intermediate output maps before they are transformed using the inverse or output transforms. This alternative mode of flow reduces the computation requirement significantly. In Figure 5.4, for each $R(i)$ region, four intermediate outputs are accumulated for four output channels. Once the accumulation is complete for all the input channels in a typical layer, each region is converted back to the spacial domain using the inverse transforms. Listing 5.7 shows an example of output transformation where the output tile size is $2 \times 2$.

```
#ifdef __aarch64__
    for (; channels_remaining >= 4; channels_remaining -= 4) {
      // Matrices used and computed during this transform
      float32x4_t F[4][4], FZ[4][2], f[2][2], b;

      // Read a 4x4 tile in the Winograd domain
      for (int i = 0, m = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++, m++) {
          F[i][j] = vld1q_f32(inptr + m*matrix_stride);
        }
      }
      inptr += 4;

      // Compute the matrix F Z
      for (int i = 0; i < 4; i++) {
        // FZ[i][0] = F[i][0] + F[i][1] + F[i][2];
        FZ[i][0] = vaddq_f32(vaddq_f32(F[i][0], F[i][1]), F[i][2]);

        // FZ[i][1] = F[i][1] - F[i][2] - F[i][3];
        FZ[i][1] = vsubq_f32(vsubq_f32(F[i][1], F[i][2]), F[i][3]);
      }

      // Compute the output tile f = ZT F Z
      for (int j = 0; j < 2; j++) {
        // f[0][j] = FZ[0][j] + FZ[1][j] + FZ[2][j];
        f[0][j] = vaddq_f32(vaddq_f32(FZ[0][j], FZ[1][j]), FZ[2][j]);

        // f[1][j] = FZ[1][j] - FZ[2][j] - FZ[3][j];
        f[1][j] = vsubq_f32(vsubq_f32(FZ[1][j], FZ[2][j]), FZ[3][j]);
      }

      // Load the bias vector
      b = vld1q_f32(bptr);
      bptr += 4;

      // Write out the output tile
      for (int i = 0; i < cells_i; i++) {
        for (int j = 0; j < cells_j; j++) {
          vst1q_f32(outptrs[i][j], vaddq_f32(f[i][j], b));
          outptrs[i][j] += 4;
        }
      }
    }
#endif // __aarch64__
```

Listing 5.7 Output transforms (2D)

## 5.4 Evaluation

### 5.4.1 Evaluation setup - models and platform

I chose five widely used ConvNets of different sizes and complexities to validate the MCMR implementation of the Cook-Toom algorithm, namely, VGG19, VGG16, GoogleNet, Inception-v3, and SqueezeNet [87]. Out of these deep ConvNets, VGG19 is the largest and SqueezeNet is the smallest. In terms of complexity, VGG19 is a sequential model, whereas all the other models consist of parallel branches. Many of these networks also consist of filters with different dimension and types.

For benchmarking the accelerated MCMR implementation of the Cook-Toom algorithm I used the Huawei HiKey 960 development platform. I implemented the algorithm using the IEEE 754 32-bit floating-point standard. This development platform is based on the Kirin 960 system-on-chip, also found inside the latest generation Huawei Mate 9 smartphones. The Kirin 960 is an octacore Arm big.LITTLE system-on-chip, comprising four Arm Cortex-A73s ('big') running at 2.36 GHz and four Cortex-A53 ('LITTLE') cores running at 1.84 GHz with 3GB of LPDDR4 SDRAM memory, and 32GB of UFS 2.0 flash storage. I focus on the Cortex-A73 cluster, which supports the Armv8-A architecture and is optimised for a wide variety of mobile and embedded application workloads, and improves sustained performance and power efficiency by 30% over the previous generation processor [34].

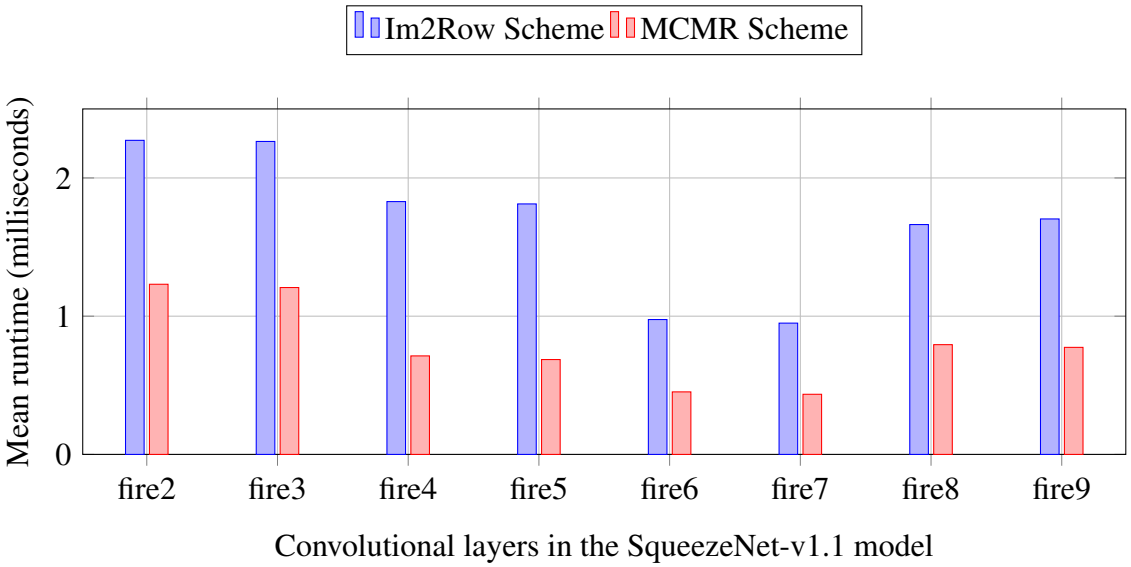



Fig. 5.6 Layerwise comparison of the runtime of the SqueezeNet-v1.1 model with a batch size of 1

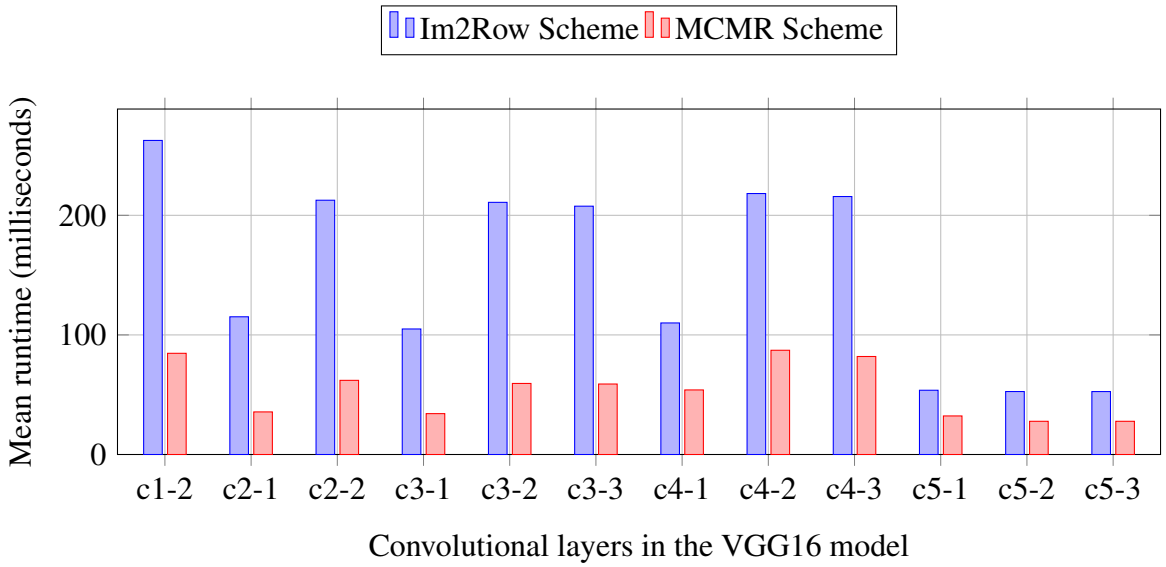



Fig. 5.7 Layerwise comparison of runtime of the VGG16 model with a batch size of 1

### 5.4.2 Results – per-layer speedup

To measure the performance of ConvNets using the MCMR algorithm I benchmarked variations of the Cook-Toom algorithm on individual layers of all the selected models. Section 4.5 shows many of these variations of the Cook-Toom algorithms used in these experiments. In each case I measured the number of cycles taken to perform all three stages of our algorithm (input transform, GEMMs and output transform) on the big cluster, which consists of four Cortex-A73 cores. As a baseline against which to compare I also benchmarked the GEMM calls which would result from application of the classic *im2row* technique to the same layers. I present the speedup achieved by the MCMR-based Cook-Toom technique over GEMM.

Figure 5.7 shows a comparison of the per-layer mean run-time between the MCMR scheme and the classical im2row scheme for a single batch inference of the VGG-16 model. For an output tile size of four, an average 2.7× speedup for individual layers, and in some cases a nearly 4× speedup over GEMM (im2row) is achieved. For all deep ConvNet models targeted on the ImageNet dataset, the first layer always operates on a tensor consisting of only three channels; this reduces the performance of the MCMR implementation since scalar, rather than SIMD, code is used for the input transform, and is a corner case for the MCMR implementation.

The SqueezeNet model consists of many fire modules within which many convolution layers are configured. The first phase of a fire module consists of a squeeze sub-module and this sub-module is followed by an expand sub-module. Figure 5.6 shows a comparsion of the

per-fire module mean run-time between the MCMR implementation and the classical im2row scheme for a single batch inference of the SqueezeNet-v1.1 model. As can be seen from the figure, SqueezeNet achieves an average 2× speedup on all relevant convolution layers.

The GoogLeNet model consists of combinations of many different filter sizes that depend on the receptive fields. The network was designed with computational efficiency and practicality in mind, so that inference can be run on individual devices including even those with limited computational resources, especially with low-memory footprint. The MCMR-based Cook-Toom convolution helps to accelerate the inference speed further. Figure 5.8 presents a comparison of the per-layer mean run-time between the MCMR scheme and the classical im2row scheme for a single batch inference of GoogLeNet model. As can be seen from the figure, on average 2.5× layerwise speedup is achievable. Furthermore, many of the layers achieve a peak speed up of 4×.

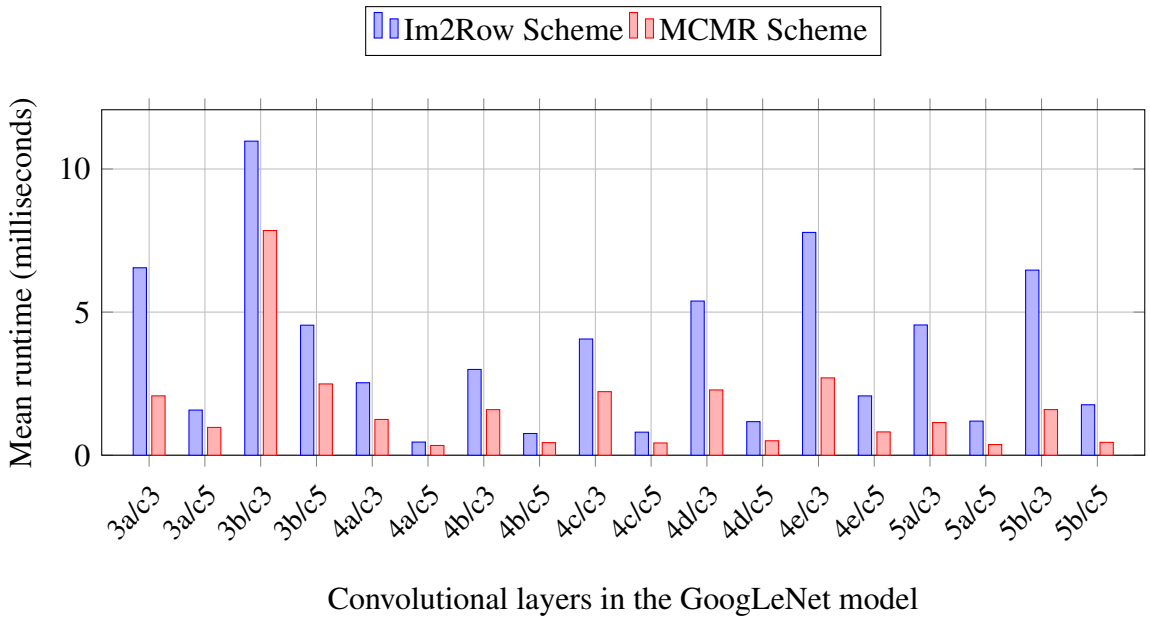



Fig. 5.8 Layerwise comparison of runtime of the GoogLeNet model with a batch size of 1

The Inception-v3 model evolved from the GoogLeNet model and it consists of many factorised $7 \times 7$ layers in addition to standard layers with filter of sizes of 3×3 and 5×5. The model is the culmination of many ideas developed by multiple researchers over the years. The model itself is made up of symmetric and asymmetric building blocks, including convolutions, average pooling, max pooling, concats, dropouts, and fully connected layers. Batchnorm is used extensively throughout the model and applied to activation inputs. Figures 5.9 and 5.10 present a comparison of the per-layer mean run-time between the MCMR scheme and the classical im2row scheme for the newly added $1 \times 7$ and $7 \times 1$ factorised layers. Using a

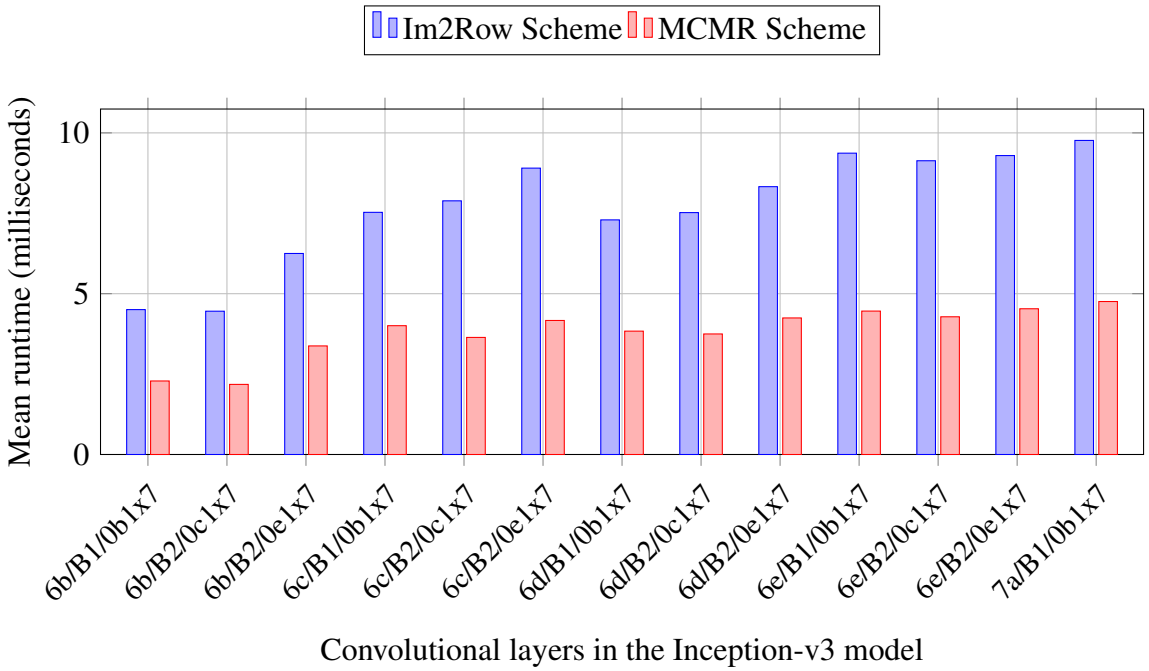



Fig. 5.9 Comparison of runtime of 1x7 layers of the Inception-v3 with a batch size of 1

variant of the modified Cook-Toom algorithm that uses an input tile size of eight, most layers achieve a speedup of 2×. Figure 5.11 shows the layerwise speedup of the other standard layers. As in the GoogLeNet, these layers also achieve an average layerwise speedup of 2.5× and a peak speedup of 4×. I have plotted the comparison of a few selected layers of the Inception-v3 model in Figure 5.12 as the absolute runtime of those layers are significantly higher than the rest of the layers. These layers are compute heavy and still manage to achieve an average 2.5× speedup using the modified Cook-Toom algorithm.

### 5.4.3 Results – whole-network speedup

To measure the effectiveness of the modified Cook-Toom-algorithm-based acceleration on end-to-end deep models, I also benchmarked the runtime of the entire ConvNets. As in the layerwise case, I also implemented the MCMR-based convolution kernels using the IEEE 754 standard. For comparison, I used the Arm Compute Library [3] to evaluate single-batch (batch size of 1) inferences of these networks on multi-threaded (4×) Cortex-A73. Two sets of benchmarks were run: in one, layers suitable for the Cook-Toom-based acceleration use the MCMR scheme, and the rest use the baseline *im2row* scheme; in the other all layers use *im2row*. Figure 5.13 presents the summary of speedup achieved in the Cook-Toom-suitable convolutional layers as a fraction of the entire model. Each bar in the figure is

divided into three segments, namely, fast convolution, direct convolution and other. Fast convolution relates to the runtime of those convolutional layers where the modified Cook-Toom acceleration is applicable. The rest of the convolutions are implemented without the modified Cook-Toom algorithm and shown under the direct convolution segment. The runtime of all the other types of operations in a model is accumulated and shown under the other segment. A typical ConvNet model consists of polling, batch norm and fully connected layers other than convolutional layers.

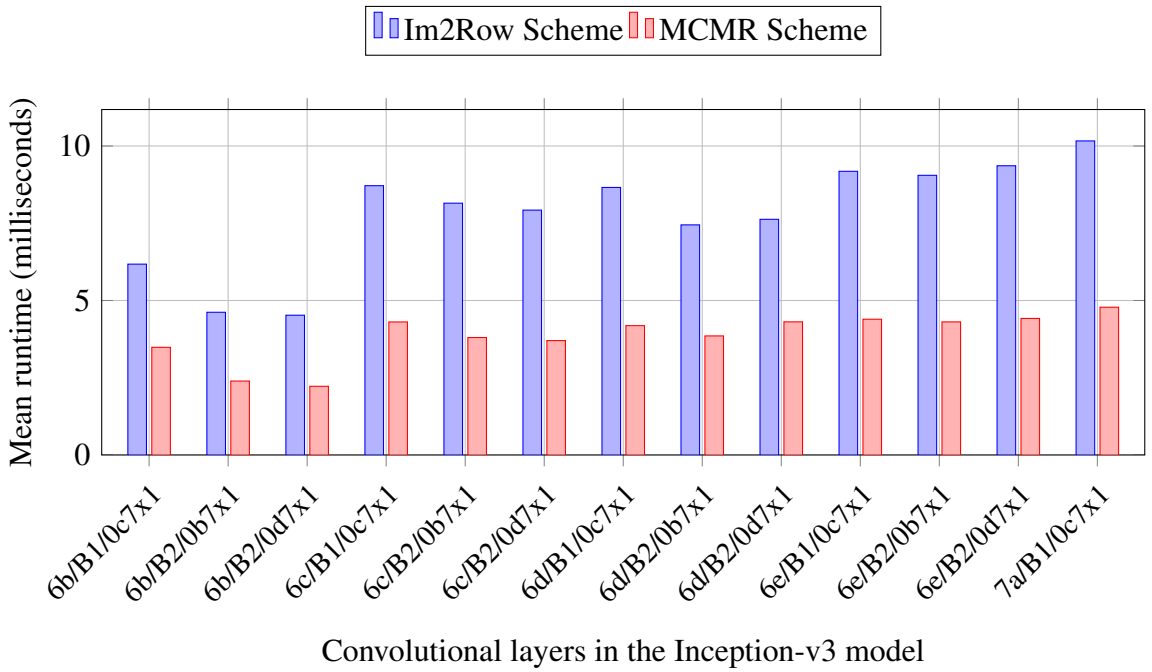



Fig. 5.10 Comparison of runtime of 7x1 layers of the Inception-v3 with a batch size of 1

Since whole-network performance is a combination of the time spent performing convolution and that spent computing other functions (such as pooling and activation) we expect to see lower speedups than those benchmarked for individual layers. This is visible in Figure 5.13, which shows the normalised runtime of all five ConvNets. In each case the entire network was run 21 times and I show the mean; noise in the system contributes to slight variations in the time spent in each portion of the network.

Tables 5.1 and 5.2 present the absolute end-to-end runtime (in milliseconds) of all the models under investigation. The first column under each model represents the runtime for the full network, whereas the second column represents the runtime only related to the layers where the MCMR-based scheme is applied. The first and second rows present the runtime of each model while using the Im2Row and the MCMR scheme, respectively. The third row presents the mean absolute runtime differences between Im2Row and MCMR schemes. The

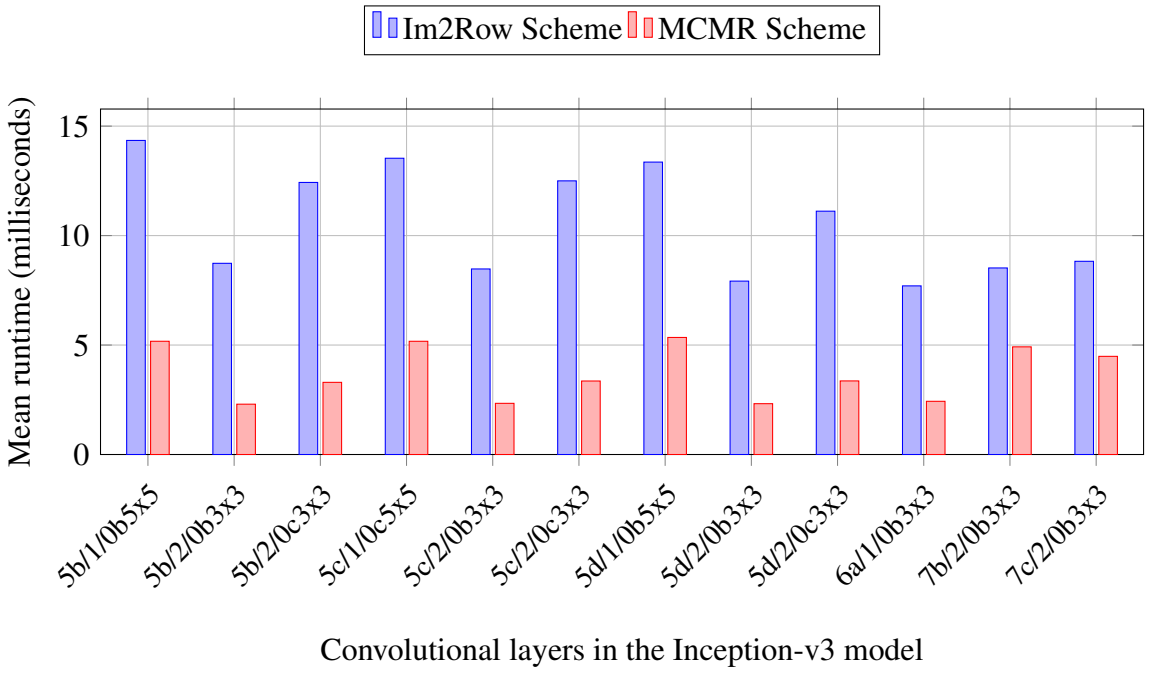



Fig. 5.11 Comparison of runtime of 3x3 and 5x5 layers of the Inception-v3 with a batch size of 1

final row shows the relative speedup achieved using MCMR scheme compared to Im2Row scheme. For VGG-16 and VGG-19, which are large models and dominated by the Cook-Toom-suitable convolution layers, we can see whole-network speedups approaching 60%. For Inception-v3 and GoogLeNet which are moderately medium-sized models, achieved a speedup of 41%. For SqueezeNet which is an extremely compact model achieves a speedup of 30%. Additionally, it can be deduced from the table that SqueezeNet and GoogleNet achieve 47 frames/sec and 10 frames/sec inference speed, respectively. The larger VGG-16 and Inception-v3 models achieve between 1 and 3 frames/sec. In all cases, the batch size was set to a single image and in all the models pre-trained accuracy was preserved.

Table 5.1 Summary of **mean absolute runtime** of the **whole network** in milliseconds (msec) for a batch size of 1 in VGG-19/-16 and GoogLeNet

| | VGG-19 | | VGG-16 | | GoogleNet | |
|---|---|---|---|---|---|---|
| | Full Network | Fast Layers | Full Network | Fast Layers | Full Network | Fast Layers |
| Using Im2Row Scheme | 2410.68 | 2311.01 | 1929.43 | 1829.10 | 173.13 | 91.42 |
| Using MCMR Scheme | 1035.58 | 936.05 | 758.05 | 670.79 | 101.04 | 38.38 |
| Speedup (msec) | 1375.10 | 1374.97 | 1171.38 | 1158.31 | 72.09 | 53.04 |
| Speedup (%) | 57.04% | 59.50% | 60.71% | 63.33% | 41.64% | 58.02% |

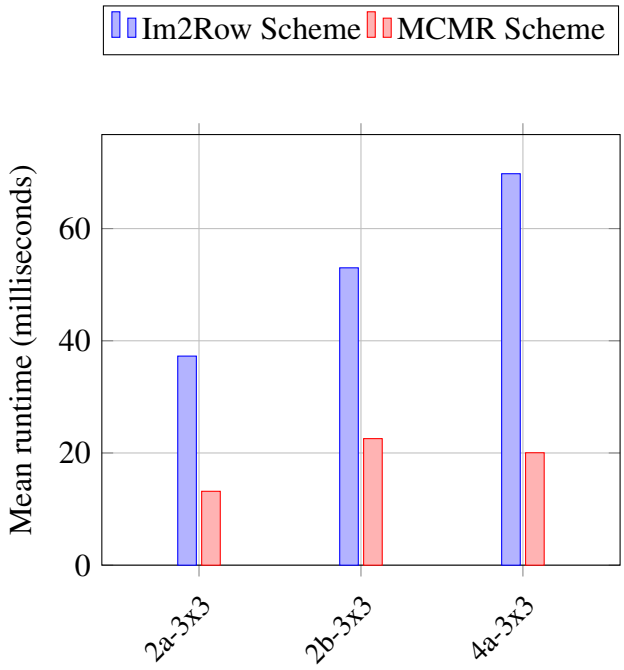



Fig. 5.12 Comparison of runtime of 3x3 layers of the Inception-v3 with a batch size of 1

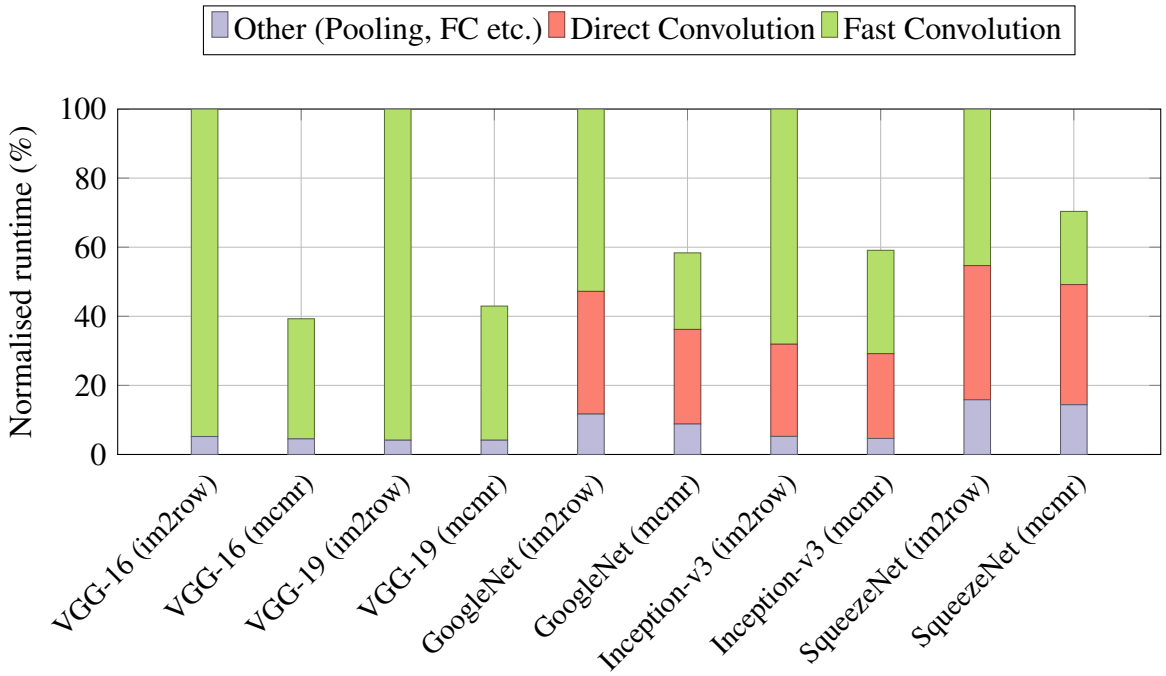



Fig. 5.13 Speed-up achieved in the Cook-Toom-suitable layers as a fraction of the entire model (batch size = 1)

Table 5.2 Summary of **mean absolute runtime** of the **whole network** in milliseconds (msec) for batch size of 1 in Inception-v3 and SqueezeNet-v1

| | Inception-v3 | | SqueezeNet | |
|---|---|---|---|---|
| | Full Network | Fast Layers | Full Network | Fast Layers |
| Using Im2Row Scheme | 750.37 | 510.92 | 29.72 | 13.47 |
| Using MCMR Scheme | 443.40 | 224.42 | 20.91 | 6.29 |
| Speedup (msec) | 306.98 | 286.51 | 8.81 | 7.18 |
| Speedup (%) | 40.91% | 56.08% | 29.64% | 53.28% |

## 5.5 Discussion

The MCMR implementation of the modified Cook-Toom algorithm on Arm's Cortex processors can dramatically reduce the compute time of individual convolution layers – by up to $4\times$. However, these speedup numbers are lower than the theoretical values. Partially, this is due to the challenges involved in implementing the algorithm in a real system but largely it is because the theoretical speed-up of any fast convolution algorithm disregards the cost of transforming to and from the alternative domain of computation.

### 5.5.1 Speedup gap due to the cost of transforms

To be more specific, the theoretical speed-up of an $F(s,w,x)$ algorithm can be computed as $\frac{sw}{s+w-1}$. The 2-dimensional version of the same algorithm can achieve a theoretical speedup of $(\frac{sw}{s+w-1})^2$. But, as can be seen from the layerwise speedup plots, none of the layers achieves the theoretical limit. As an example, the theoretical speed-up limit for an $F(4,3,6)$ variation of the Cook-Toom algorithm in 2 dimensions is $4\times$, whereas I have only benchmarked a $3.5\times$ speed-up on a single-threaded implementation. This gap between the theoretical and achieved speed-ups can be somewhat overcome by amortising the transform costs over those of the GEMMs. For example, since each input region is used to compute multiple output regions (as many as there are output channels) the cost of the input transformation is amortised over the number of output channels. Hence, as the number of output channels increases, the speed-up asymptotically approaches the maximum achievable.

### 5.5.2 Speedup gap due to the limitations of hardware architecture

The cost of the input transforms is also affected by their use of architectural registers. For example, the input transform for the $F(2,3,4)$ variation of the Cook-Toom algorithm can be applied on a $4\times4$ region of the feature map. Using the NHWC tensor ordering, sixteen SIMD registers are used to store a single one of these regions – consequently two can be stored in

the thirty-two SIMD registers available in the Armv8-A architecture. However, the input transform for the $F(4,3,6)$ variation of the Cook-Toom algorithm for a 2-dimensional feature map must work on regions of size $6\times6$, requiring four more registers than are available. Consequently, the intermediate values cannot be held in registers and must be spilled and read from the stack – consuming more memory bandwidth. As a result, although the theoretical speed-up increases with the tile size, achievable speed-up rapidly becomes limited by the cost of performing the input and output transforms. Depending on how much on-chip storage is available on a particular architecture, after an optimum input tile size the law of diminishing returns sets in. Furthermore, the cost of computing all the transforms also increases with those algorithms that use a larger tile size, resulting a lower effective speed-up.

### 5.5.3 Challenges in implementing the Cook-Toom convolution using lower-precision arithmetic

Over the past few years, reduced-precision techniques have proven exceptionally effective in accelerating deep-learning inference applications [38]. However, the Cook-Toom class of fast convolution algorithms are very sensitive to precision. In particular, computing the input and the output transformations requires high precision. Without a higher precision in the data path, the algorithms become unstable. To this end, I have explored the different ways in which the instability in the computation can be somewhat avoided. One possible solution could be to perform the input transform from the following layer in higher precision as a post-processing step after the element-wise multiplications and before the intermediate activation is sent back to the off-chip memory. However, this is challenging as there is not enough on-chip storage to keep processing activations further. Due to multiple memory read/writes in the resource-constrained processor, the benefit of the Cook-Toom scheme disappears. A custom mixed-precision fused MAC may help in this scenario, which I explore in the design of the Cook-Toom accelerator in Section 6.2.2 of the next chapter.

### 5.5.4 Summary

The modified Cook-Toom algorithm helps to reduce the complexity of the dense convolutional layers in modern ConvNets. But, naive and straight forward implementation of such algorithm using off-the-self software libraries results in poor utilisation of the underlying hardware. In this chapter, I showed how a multi-channel multi-region-based scheme on top of GEMM linear-algebra subroutines can help in efficient implementation of such a class of fast algorithms. Through experiments on a real development board I showed how individual layers benefit from the MCMR scheme. Additionally, I also showed benchmarked

performance for end-to-end models comparing with existing libraries. Although the results show significant performance improvement in all cases, none of the results matched the theoretical speedup limit. I concluded this chapter with a discussion on plausible causes behind this gap in results.

# Chapter 6

# Scalable architecture to enable fast inference using the Cook-Toom class of convolutions

In the previous chapter, I discussed the limitations of implementing the Cook-Toom class of convolutions on traditional resource-constrained Arm processors. In this chapter, I show how a more efficient and scalable architecture can be built for fast inference that addresses the limitations of a traditional general-purpose processor. In this chapter, I start by describing a list of guiding principles for designing efficient accelerator architectures that support fast inference. I then present the detailed architecture of the Cook-Toom accelerator. In the evaluation section, I show results from various performance evaluation experiments. I conclude the chapter with an in-depth discussion on a number of additional features that can be added to further improve performance.

## 6.1 Design principles

In this section, I highlight four principles that generally guide the design of the Cook-Toom accelerator discussed in this chapter. Not only do these guidelines lead to increased performance and energy efficiency, they also provide two valuable bonus effects. First, the architecture of this accelerator is highly scalable. The number of cores can be configured to support a wide range of applications with different targets for power, performance and area. Second, the architecture also supports a good level of configurability in order to support a broad range of operations in the hope of future proofing the design to some extent. For

example in Section 6.7.1, I briefly discuss how $1 \times 1$ convolutions can easily be implemented on this accelerator without any change to the hardware.

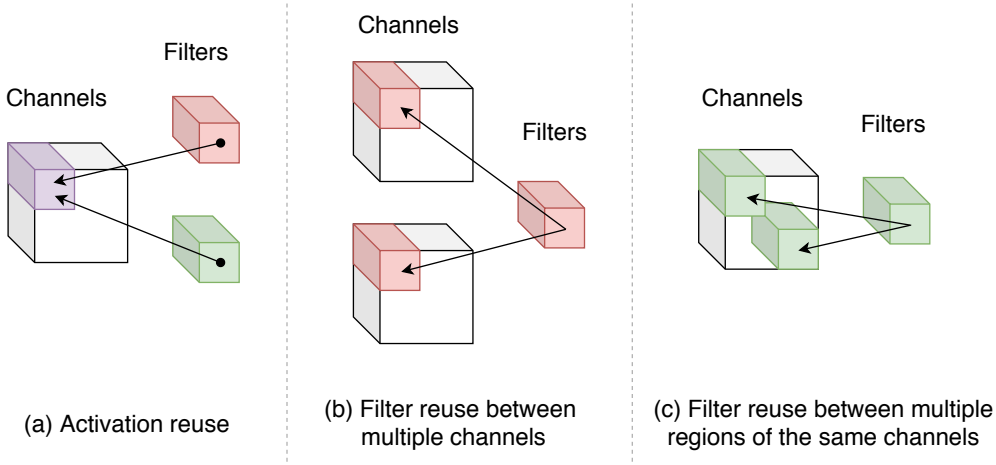



Fig. 6.1 Data-reuse patterns in modern ConvNets

- **Reuse patterns:** Modern ConvNets almost always have inherent parallelism. The key design decision for an efficient and scalable accelerator architecture is how to take advantage of that parallelism. The design decisions must follow the natural granularity of the parallelism and expose that parallelism simply in the programming model. For example, multiple feature maps or multiple regions of the same feature map can be exposed to same filter for parallel convolutions. Similarly, multiple filters can be used to convolve the same feature map to obtain different output features. All these three different types of reuse patterns are illustrated in Figure 6.1. A flexible accelerator must support some mechanism to support both these classes of parallelism for fast inference. A detailed discussion on this topic can be found in Section 2.6.

- **Staying local:** The energy cost of accessing DRAM is very high compared to the actual cost of arithmetic operations. Figure 6.2 illustrates the energy cost of an instruction in addition to the individual cost of arithmetic operations and various levels of memory access at a 45nm technology node [47]. As can be seen from the figure, it is not only DRAM accesses that are expensive, instruction-cache and register accesses account for almost one-third of the energy of an instruction. Therefore, to design an efficient accelerator the aim should be to load as much data as possible in one go and hold it in on-chip SRAM to maximise reuse. Various reuse patterns should be exploited here before the loaded data is removed from the on-chip SRAM. Furthermore, as the memory access pattern is well-known in ConvNets, instead of many levels of caches,

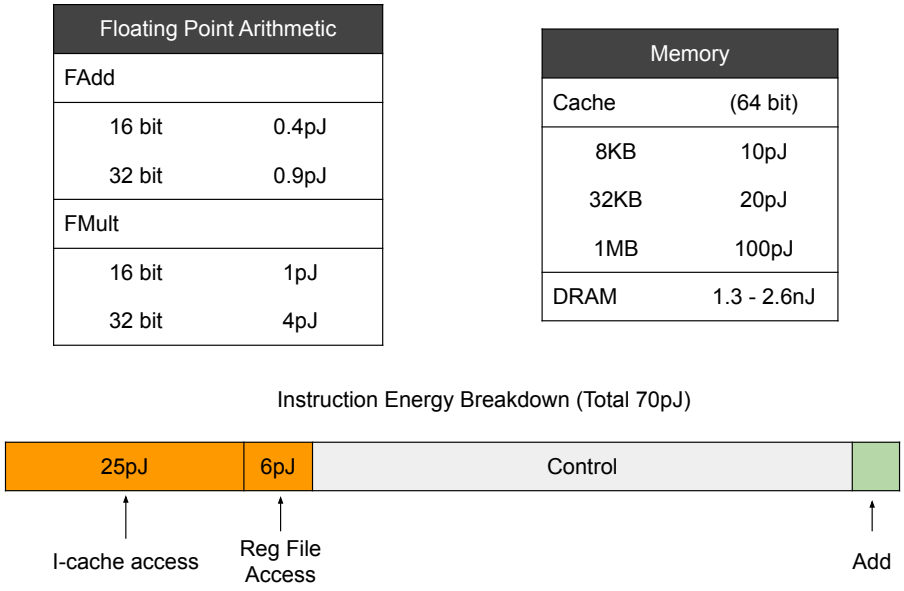



Fig. 6.2 Energy cost of DRAM access vs arithmetic operation at 45nm technology [47]

DMA-based memory access would help to reduce the energy cost significantly. In fact, many energy efficient domain-specific accelerators use software-controlled scratchpad memory as the access pattern is well-known [52, 4].

- **Reduction in data size:** As can be seen from Figure 6.2, reducing the data size also helps to lower the energy cost of computations. For example, at 45nm technology, 32-bit floating-point multiplication costs 4pJ. Reducing the multiplier to 16-bit lowers the energy cost by $4\times$. Thus, instead of few wider computation engines, more arithmetic units can be packed into the same chip area. This would also help indirectly in data reuse as more computations can be done on the same piece of data. In addition, the memory bound part of the computation can benefit from better memory-bandwidth utilisation.

- **Accelerate the common cases first:** The architectures of modern deep neural networks are always evolving. It is almost impossible to design one fixed architecture which supports all types of convolutions and other operations that are yet to become prominent. On the contrary, without any basic accelerator the field cannot make progress either. This dilemma between always evolving models and market demand for suitable hardware has given rise to numerous architectures that support some combination of features and operations from the collection of modern deep networks. To stay truly relevant to market demand and the need for acceleration for the future, one needs to analyse the workload and look for the most common component of the

workload that stays unchanged through evolution. If this fixed portion of the workload is significant, then accelerating this would always help. To this end, as discussed in the introductory chapter, a significant portion of the modern ConvNet is always dominated by dense spatial convolution. The aim of this specialised architecture is to accelerate this particular portion of the workload and also keep it flexible enough to support other types of data-reuse pattern.

## 6.2 High-level architecture of the Cook-Toom accelerator

The high-level flow of the Cook-Toom accelerator is presented in Figure 6.3. The Cook-Toom accelerator also follows the MCMR (multi-channel multi-region) implementation scheme discussed in Chapter 5. The accelerator consists of many fixed-function blocks which can be turned on or off as required. To start with, the DMA-in block accesses activation and filter data and stores them in on-chip SRAM memory. The DMA-in block supports both NHWC and NCHW data formats. The details of these data formats was introduced in Section 5.3.2. The on-chip SRAM is divided between the filter parameters and input activation. The

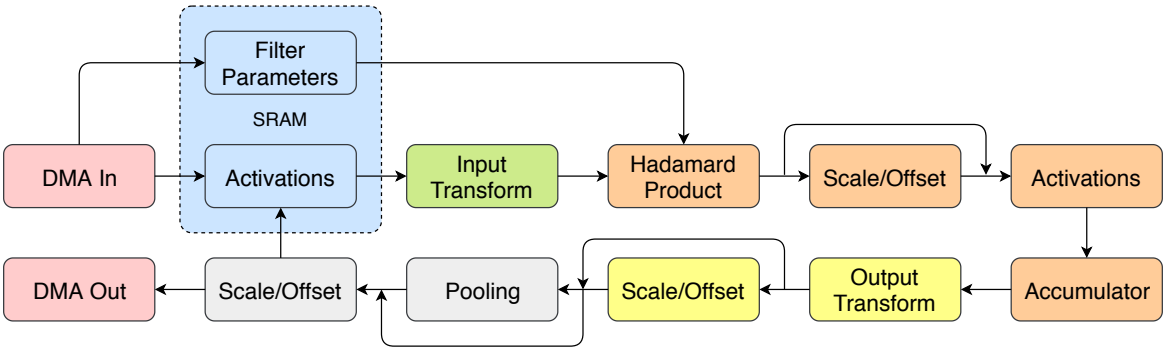



Fig. 6.3 High Level Functional Diagram of the Accelerator

activations (alternatively known as feature maps) are then pre-processed using the Cook-Toom input transforms. Once processed by the input transforms they are temporarily stored in ping-pong buffers for element-wise multiplication. The filter parameters are not required to pass through any input transform stages during inference as they are pre-processed offline. Once both the filters and activations are ready they are streamed into the Hadamard product engine. The Hadamard engine is the central component of the Cook-Toom accelerator. It can process multiple element-wise multiplications in the same cycle. The next section is dedicated to explain the details of this functional block. Once the output is ready from the Hadamard product engine it can be passed through the optional Scale/Offset block for making an adjustment either due to quantisation or batch normalisation, which is present

in many modern convolutional neural networks. The output from this block is temporarily stored in an activation buffer which is part of the on-chip SRAM. As described in the MCMR scheme in the previous chapter, activations are first accumulated in the Accumulator block for all the input channels before the inverse transform is applied only once per output channel. This saves a significant amount of computation in the output transform stage. The output Transform block handles the inverse transform operations which produce the activation in the spatial domain. The Output transform block is followed by an optional Scale/Offset block that can be used to adjust the data for quantisation. The activations from this stage either can be sent to the main memory using the DMA Out block or can be kept on-chip for further processing. For most of the layers, the activations are kept on SRAM for maximising re-use. I also show an optional Pooling block in the figure, which can be used before storing the activations on the SRAM or DRAM. The pooling block is only added in the diagram for completeness and I do not explore the functionality of the pooling block any further.

### 6.2.1 The architecture of the Hadamard Engine

The central component of the Cook-Toom accelerator is the Hadamard Product Engine (in short, Hadamard Engine). Unlike many earlier neural-network accelerators, the Hadamard engine makes use of all three types of reuse pattern illustrated in Figure 6.1, namely, activation reuse, filter reuse between channels and filter reuse between multiple regions. The Hadamard engine has four degrees of configurability and a particular configuration can be expressed by the metric $< f_i \times T \times f_o \times L >$. The parameters $f_i$ and $f_o$ are decided by the number of input and output channels processed per cycle, respectively. The configuration parameter $T$ is decided by the tile size of the Cook-Toom algorithm and $L$ is decided by the number of lanes or regions processed simultaneously in the Hadamard engine. The architecture of a typical Hadamard engine of configuration $< 8 \times 4 \times 8 \times 4 >$ is illustrated in Figures 6.4 and 6.5. In the next few sections the working principles of the engine are explained in detail.

#### 6.2.1.1 Energy saving output stationary data flow for MCMR scheme

The MCMR scheme, explained in the previous chapter, helps to reduce the number of inverse transforms drastically and thus lower the energy consumption. The use of this scheme also enforces the output-stationary data flow in the Hadamard engine. The guiding equation for the MCMR scheme is the following:

$$f = S^T(\sum_{c=0}^{C}[(WwW^T)_c \odot (X^T xX)_c])S \tag{6.1}$$

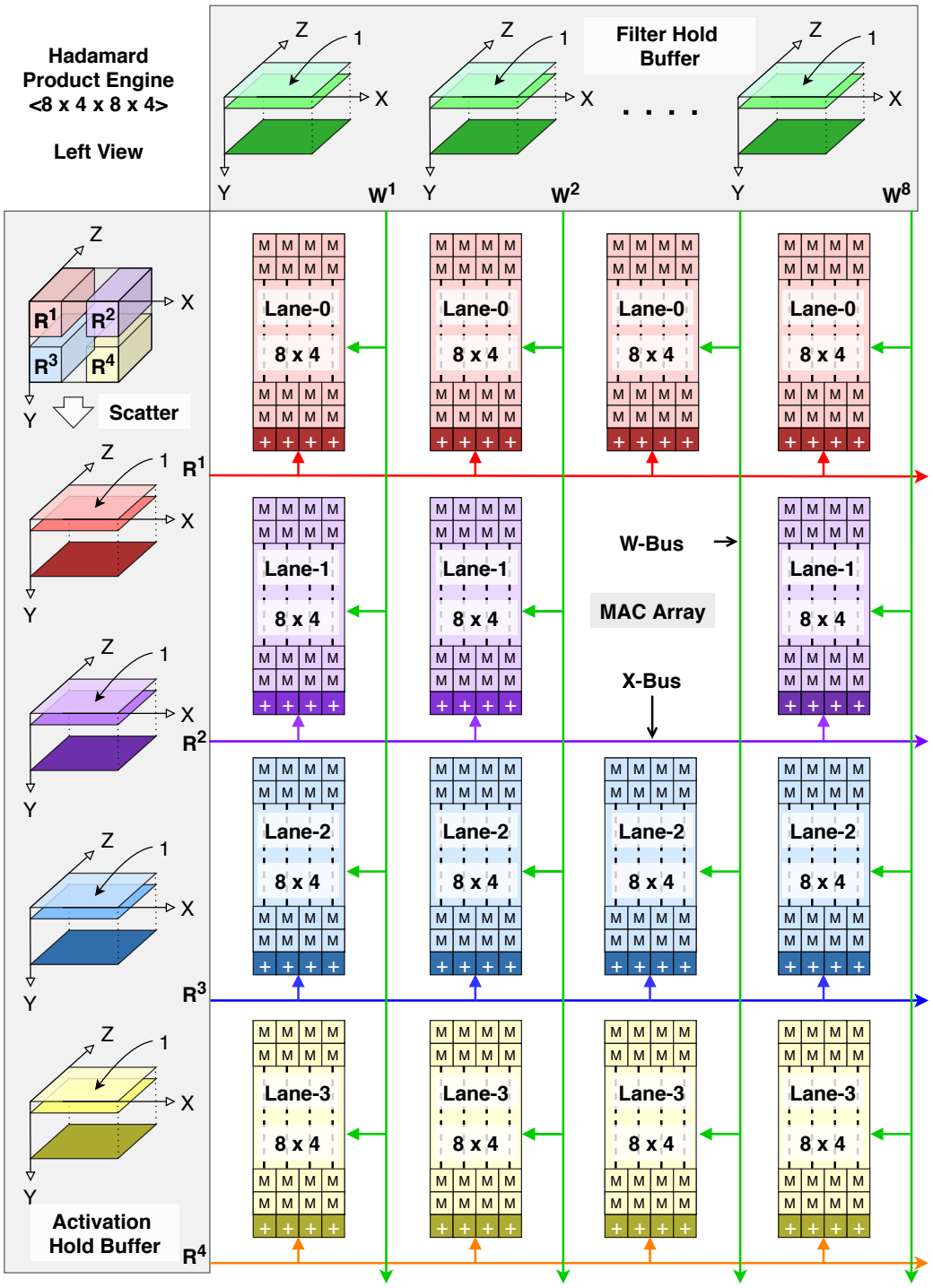



Fig. 6.4 The left view of the architecture of the central Hadamard Product Engine (Config: $< 8 \times 4 \times 8 \times 4 >$); 'M' is a MAC unit, '+' is an accumulator

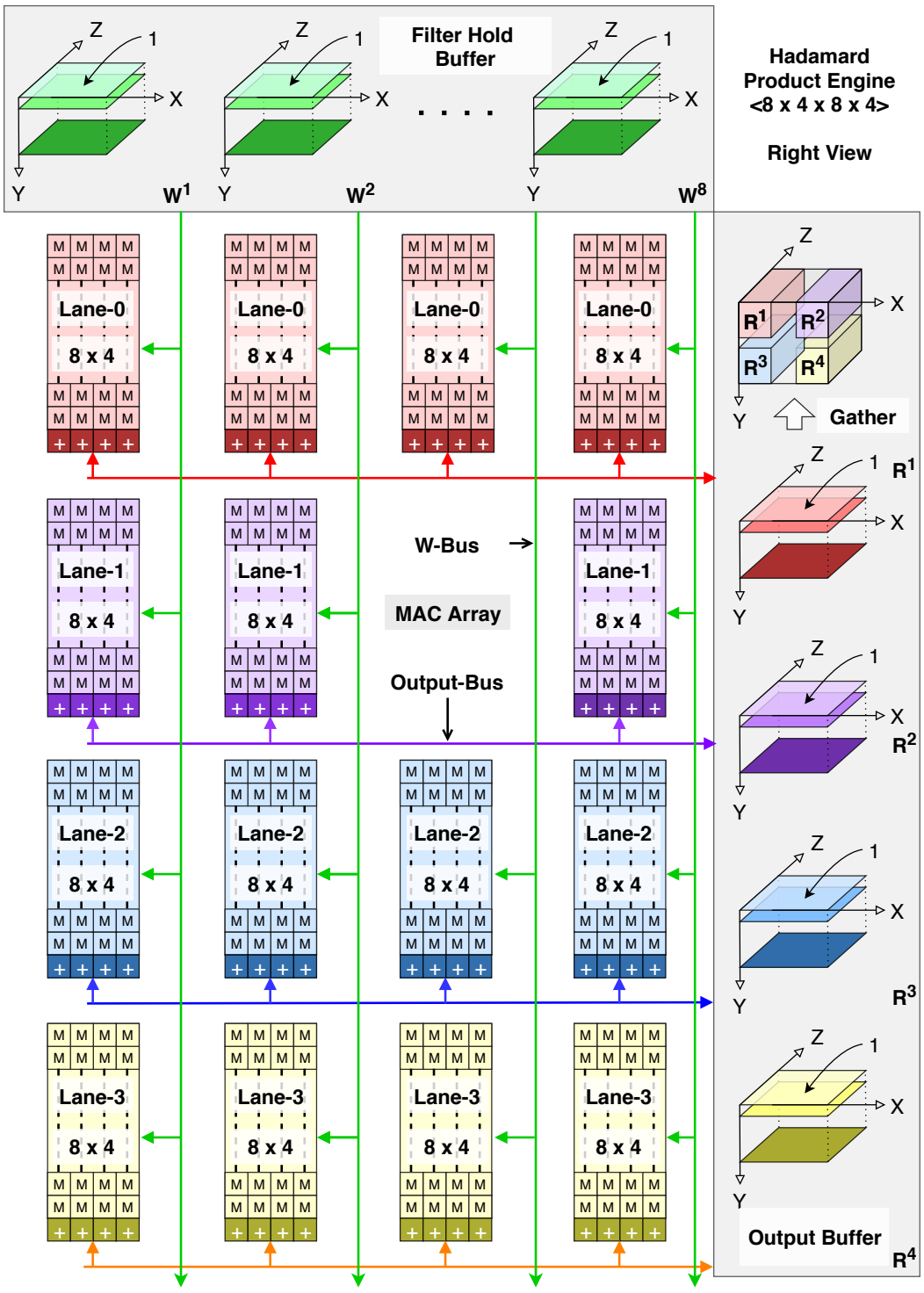



Fig. 6.5 The right view of the architecture of the central Hadamard Product Engine (Config: $< 8 \times 4 \times 8 \times 4 >$); 'M' is a MAC unit, '+' is an accumulator

As can be seen from Equation 6.1, the element-wise products are first accumulated for all input channels. The inverse transform is then applied on the accumulated sum. This naturally guides to the use of output-stationary data flow. For each output channel, the partially accumulated sum is kept on SRAM temporarily until all the input channels are considered. On the contrary, if we used the input-stationary or weight-stationary data flow, then multiple temporary results need to be stored on SRAM. This would require either a significant amount of on-chip storage or partial results must be sent back to main memory. Either of the techniques would increase the energy consumption significantly. Instead, the output-stationary data flow perfectly aligns with the guiding MCMR equation and helps to lower the energy consumption. Once all the element-wise products for a particular output channel are completed, control will move on to computing the elements of the next output channel.

#### 6.2.1.2 Support for different reuse patterns

To enable three different types of reuse pattern, the Hadamard engine uses two broadcast buses, namely X-bus and W-bus. In Figure 6.4, every input 3D tensor is represented within a xyz-frame. Each channel lies on a xy-plane, whereas the z-axis represents the number of channels. Although the Cook-Toom algorithm is applied on each channel at a time, the Hadamard engine processes many channels per cycle. To be specific, $f_i$ number of input channels are broadcast on the X-bus and all these activations are reused by multiple output filters. The number of filter sets depends on the configuration parameter $f_o$. Filter reuse is enabled by the W-broadcast bus. As shown in the figure, there are multiple X-buses depending on the number of lanes (L). Thus, each set of filter weights is reused by L sets of input regions or different groups of channels.

#### 6.2.1.3 Support for different variations of the Cook-Toom algorithms

In the previous chapter, I explained how different variations of the Cook-Toom algorithm may be needed to maximise the performance gain. In the Hadamard engine, this can be achieved by the configuration parameter T. As an example, if we are to use the algorithm based on a tile size of 4 and $f_i$ is set to 8 then in each cycle the Hadamard engine will read an (xz) horizontal slice of size $(8 \times 4)$. This enables simultaneous processing of many input pixels and improved throughput of the Hadamard engine.

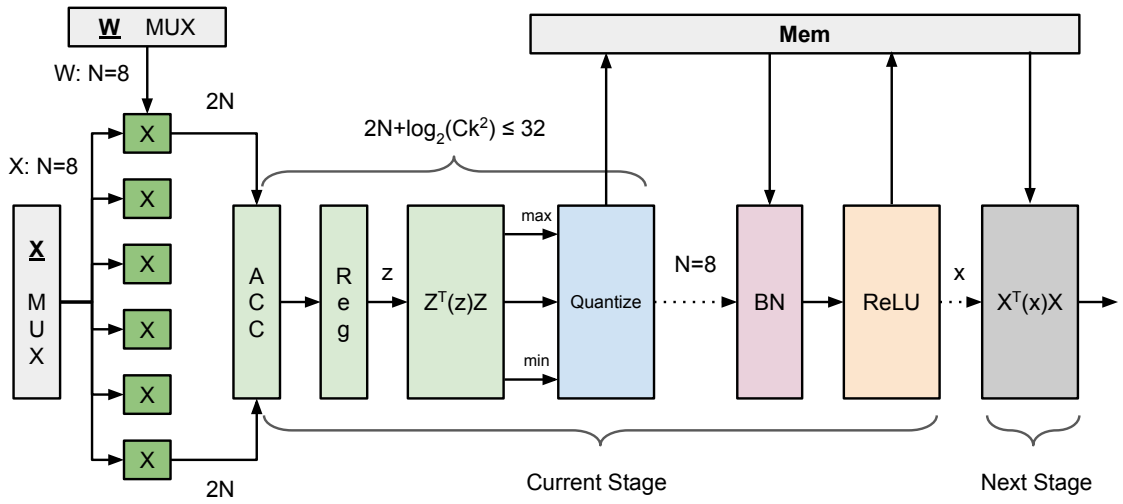



Fig. 6.6 The architecture of the fused MAC unit - generic

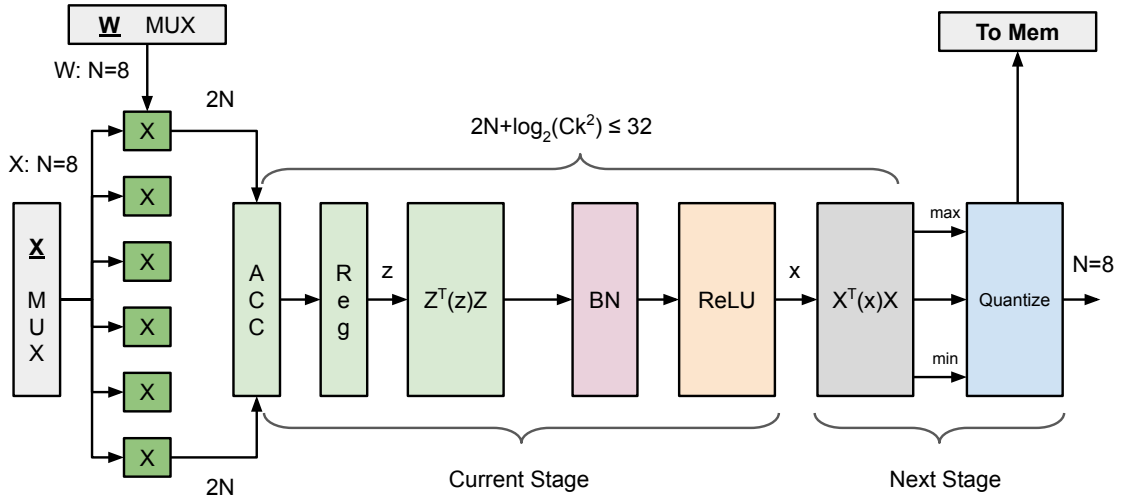



Fig. 6.7 The architecture of the fused MAC unit - optimised

### 6.2.2 Fused MAC architecture

At the beginning of this chapter, I mentioned that a reduction in data size helps to lower the energy cost and improves the operation density of the cores. However, the accuracy of the Cook-Toom algorithm is very sensitive to the precision of the data used during the computation. To mitigate the effect of low precision, the MAC (multiply and accumulate) unit uses mixed-precision components. To optimise the data path further, the MAC unit is fused with normalisation (BN), an activation unit (ReLU), and uses a tapered data path as shown in Figure 6.6. In this case, 8-bit quantised activations and filters weights are fed to the multiplication unit. The multiplication unit produces 16-bit products, which are accumulated inside a 32-bit accumulator for all the input channels. Since the accumulator needs to handle the sum of many input channels, a check must be done to make sure the program does not accumulate more than C input channels at a time. The value of the parameter C is given by the following equation:

$$2N + log_2(Ck^2) \leq 32 \tag{6.2}$$

where N is the width of the activation, and k is the filter dimension. But, since each xy pixel position is accumulated on its own, k is 1 here. The results from the accumulator first pass through the inverse transform unit and are then quantised. This reduces the memory bandwidth and requirement if the activations are sent back to the memory. The normalisation and activations are applied after the quantisation block.

The Cook-Toom algorithm becomes very unstable if the input and output transforms are applied on 8-bit precision. A straight forward fixed-precision implementation leads to drop in prediction accuracy. To avoid this instability, I fuse the current stage's inverse transform and next stage's input transforms with the MAC unit. The quantisation unit is only applied at the final stage of the pipeline, just before the activations are ready for the next element-wise operations. The fused MAC pipeline also allow efficient use of mixed-precision data to avoid any overflow during computation. This optimised version of the fused MAC unit helps to reduce the size of the MAC units significantly and also does not affect the accuracy of the Cook-Toom algorithm. Figure 6.7 presents the architecture of the optimised fused MAC unit used in the Hadamard engine.

### 6.2.3 Input-transformation engine

In the MCMR scheme, the filter transforms can be pre-processed offline as they don't change after training. The input transforms still need to be performed on-the-fly before the activations are fed to the Hadamard engine. Figure 6.8 illustrates an efficient and light-weight pipeline

architecture of the input-transform engine. Inside this engine, the input goes through a matrix multiplier and a transpose block repeatedly to produce the final result. The intermediate results are stored in the registers. The final results are stored in a ping-pong buffer-set, namely, buffer 0 and buffer 1, to avoid starvation in the Hadamard engine due to any memory bottleneck.

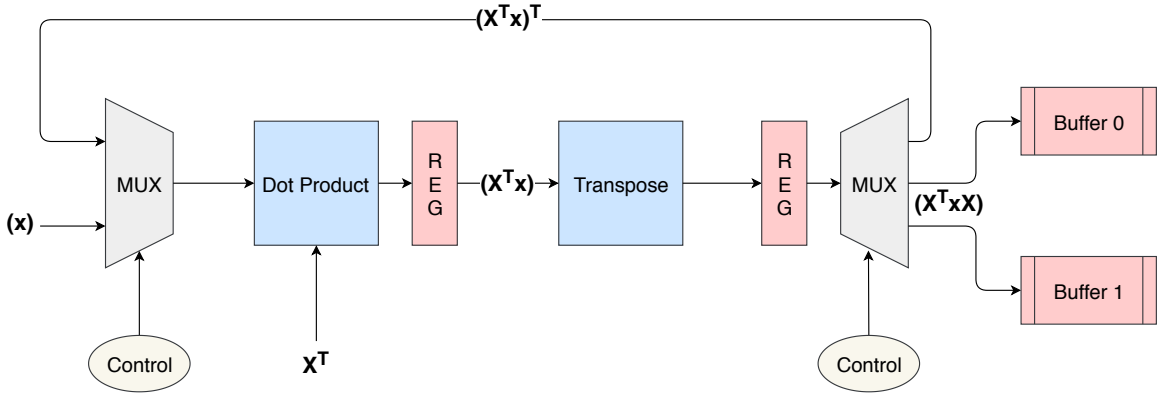



Fig. 6.8 The architecture of the input transform Engine

## 6.3 On-chip SRAM, hold buffers and data layout

The Cook-Toom accelerator has 1MB on-chip SRAM which can be shared between the activation and filter parameters. The amount of on-chip storage is decided based on maximum memory-footprint requirement of a convolutional layer of most common ConvNet models. The objective here is to reduce duplicate DRAM reads. In a typical scenario half of the SRAM is allocated to activation memory and the other half for the parameters.

Apart from this on-chip SRAM, the Hadamard Engine also has dedicated activation and filter hold buffers as shown in Figure 6.4. These buffers are crucial to the performance of the accelerator as they hold the activations and weights temporarily to maximise data reuse. Therefore, determining the appropriate size is very important for proper functioning of the Hadamard engine. Typically, the size of these hold buffers can be derived from the configuration parameters of the Hadamard Engine. As an example, for a Hadamard Engine configuration $< 8 \times 4 \times 8 \times 2 >$, 1KB of buffer memory will be allocated to activations and another 1KB of buffer memory will be allocated to filter parameters. The buffer allocation can be set appropriately to suit the reuse budget of either of the four configuration parameters in a Hadamard engine. For example, if the engine is designed to process more feature map regions per cycle then the activations would require a larger share of the buffer memory. Table 6.1 illustrates buffer memory allocation for a few example configurations of the Hadamard

Engine. The best accelerator configuration can be chosen by analysing two main factors, namely, the tile size and the total memory-footprint of a convolutional layer. The rest of the configuration parameters can be decided from the power-performance-area budget of the target application. For example, if multi-lane input-processing is possible, the power budget will limit on how many such lanes can be processed simultaneously.

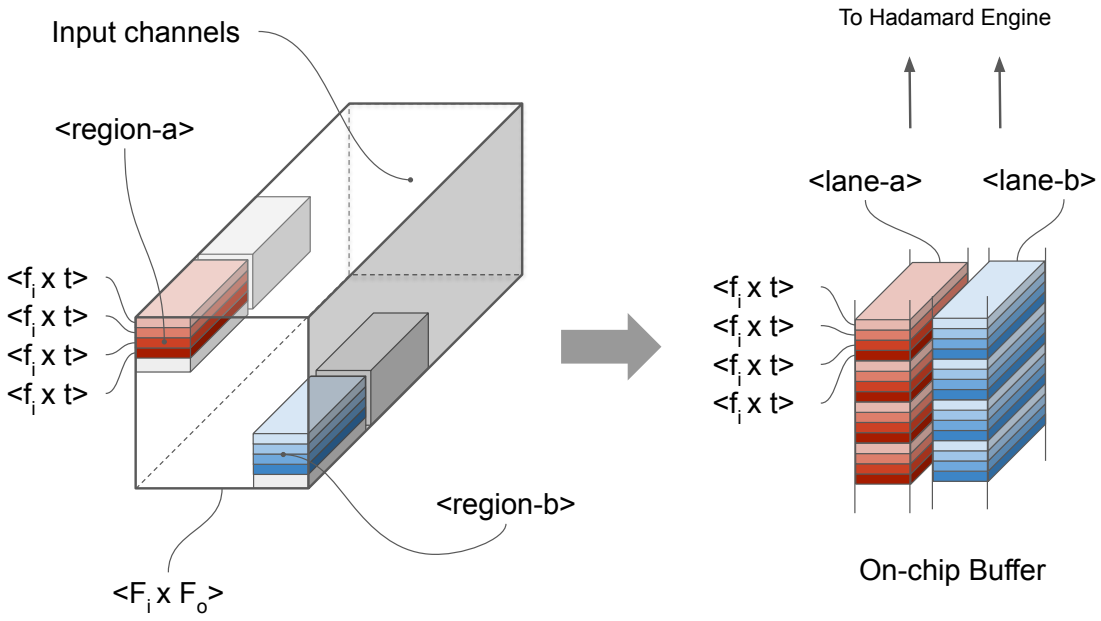



Fig. 6.9 Activation-buffer layout

Typically, activations are stored on the hold buffer to allow processing on the xy-plane. But, in the Hadamard engine, after the application of the Cook-Toom input transforms, the activations are stacked on the xz-plane. Depending on the engine configuration, multiple xz-slices from different regions or lanes are stacked on separate lanes. The alignment of the xyz-system with the activation tensor is shown in Figure 6.4. Figures 6.9 and 6.10 illustrate the storage layout of the activations and filters. The xz-plane-based layout helps in the next stage when the Hadamard Engine reads each xz-activation in every cycle. A similar layout is also applied on the filter memory. The filters are always pre-processed offline and each output set corresponding to each output channel is stacked on the xz-plane. Once the Hadamard Engine reads one xz-slice of activations and multiple xz-slice of filters, then it starts performing element-wise operations on them.

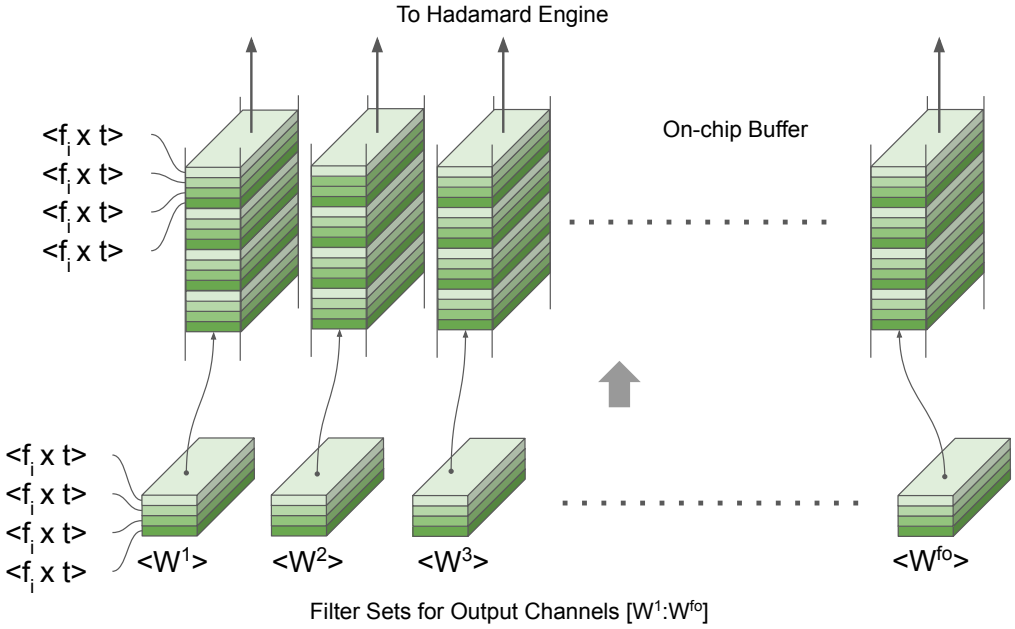



Fig. 6.10 Filter-buffer layout

Table 6.1 Activation and filter hold-buffer allocation table for various configurations of the Hadamard Engine $< f_i \times T \times f_o \times L >$

| Hadamard Engine Configuration | Activation Buffer (kB) | Filter Buffer (kB) | Total Buffer (kB) |
|---|---|---|---|
| $< 4 \times 4 \times 8 \times 4 >$ | 1 | 0.5 | 1.5 |
| $< 4 \times 4 \times 8 \times 8 >$ | 2 | 0.5 | 2.5 |
| $< 8 \times 4 \times 8 \times 2 >$ | 1 | 1 | 2 |
| $< 8 \times 4 \times 8 \times 4 >$ | 2 | 1 | 3 |
| $< 16 \times 4 \times 16 \times 2 >$ | 2 | 4 | 6 |
| $< 16 \times 4 \times 16 \times 4 >$ | 4 | 4 | 8 |
| $< 32 \times 4 \times 8 \times 1 >$ | 2 | 4 | 6 |
| $< 16 \times 4 \times 32 \times 2 >$ | 2 | 8 | 10 |

## 6.4 Mapping convolutions to the Hadamard Engine

The Hadamard Product engine works pass-by-pass. In each pass it loads a 3-dimensional tensor and processes all pixels for multiple output channels. The number of cycles it takes to complete one pass depends on the configuration of the Hadamard Engine. As an example, I show a single pass for a $(8 \times 8 \times 8)$ 3-d tensor running on a $< 8 \times 4 \times 8 \times 1 >$ Hadamard engine in Figure 6.11. For brevity, I show a mapping of the frontal $(8 \times 8)$ slice only in the figure. For each filter tensor, the Hadamard engine finishes a particular xy-patch in each

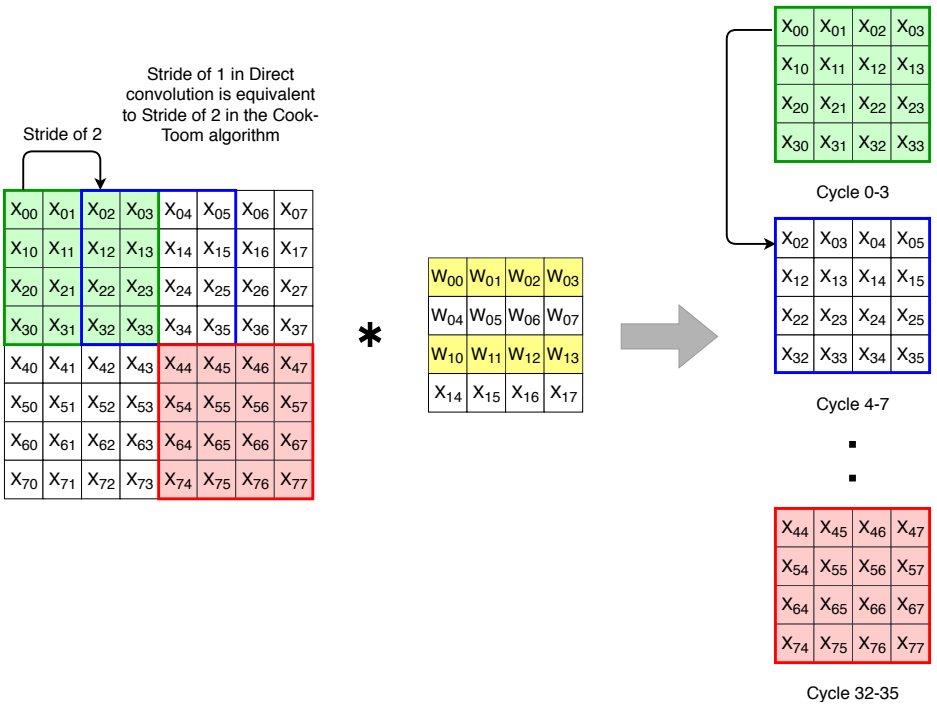



Fig. 6.11 Tiles mapped to the Hadamard Engine in each cycle

cycle and then moves on to the next xy-patch. This patch-by-patch scheme minimises the filter-parameter reads from SRAM. In the running example of $(8 \times 8 \times 8)$, there are a total of nine xy-patches to be processed. In the Cook-Toom algorithm, each $(4 \times 4)$ patch produces a $(2 \times 2)$ output patch. Thus, the stride of one in direct convolution is equivalent to a stride of two in the Cook-Toom convolution. For each patch, there are four rows and it takes four cycles (cycles 0-3) to complete. Since there are nine xy-patches, it will take 36 cycles to complete the pass. Listing 6.1 shows the pseudo-code program for the above example convolution.

```
// Example convolution for (8 X 8 X 8) input tensor and 4 X 4 tile
//Top Rows
// Patch 0
    Cycle 0: Accumulator_0 += {X_00,X_01,X_02,X_03} * {W_00,W_01,W_02,W_03}
    Cycle 1: Accumulator_1 += {X_10,X_11,X_12,X_13} * {W_04,W_05,W_06,W_07}
    Cycle 2: Accumulator_2 += {X_20,X_21,X_22,X_23} * {W_10,W_11,W_12,W_13}
    Cycle 3: Accumulator_3 += {X_30,X_31,X_32,X_33} * {W_14,W_15,W_16,W_17}

// Patch 1
    Cycle 4: Accumulator_0 += {X_02,X_03,X_04,X_05} * {W_00,W_01,W_02,W_03}
    Cycle 5: Accumulator_1 += {X_12,X_13,X_14,X_15} * {W_04,W_05,W_06,W_07}
    Cycle 6: Accumulator_2 += {X_22,X_23,X_24,X_25} * {W_10,W_11,W_12,W_13}
    Cycle 7: Accumulator_3 += {X_32,X_33,X_34,X_35} * {W_14,W_15,W_16,W_17}

// Patch 2
    Cycle 8: Accumulator_0 += {X_04,X_05,X_06,X_07} * {W_00,W_01,W_02,W_03}
    Cycle 9: Accumulator_1 += {X_14,X_15,X_16,X_17} * {W_04,W_05,W_06,W_07}
    Cycle 10: Accumulator_2 += {X_24,X_25,X_26,X_27} * {W_10,W_11,W_12,W_13}
    Cycle 11: Accumulator_3 += {X_34,X_35,X_36,X_37} * {W_14,W_15,W_16,W_17}

//Middle Rows
// Patch 4
    Cycle 12: Accumulator_0 += {X_20,X_21,X_22,X_23} * {W_00,W_01,W_02,W_03}
    Cycle 13: Accumulator_1 += {X_30,X_31,X_32,X_33} * {W_04,W_05,W_06,W_07}
    Cycle 14: Accumulator_2 += {X_40,X_41,X_42,X_43} * {W_10,W_11,W_12,W_13}
    Cycle 15: Accumulator_3 += {X_50,X_51,X_52,X_53} * {W_14,W_15,W_16,W_17}

    ... ... ... ... ... ...

//Bottom Rows
// Patch 7
    Cycle 24: Accumulator_0 += {X_40,X_41,X_42,X_43} * {W_00,W_01,W_02,W_03}
    Cycle 25: Accumulator_1 += {X_50,X_51,X_52,X_53} * {W_04,W_05,W_06,W_07}
    Cycle 26: Accumulator_2 += {X_60,X_61,X_62,X_63} * {W_10,W_11,W_12,W_13}
    Cycle 27: Accumulator_3 += {X_70,X_71,X_72,X_73} * {W_14,W_15,W_16,W_17}

    ... ... ... ... ... ...

// Patch 9
    Cycle 32: Accumulator_0 += {X_44,X_45,X_46,X_47} * {W_00,W_01,W_02,W_03}
    Cycle 33: Accumulator_1 += {X_54,X_55,X_56,X_57} * {W_04,W_05,W_06,W_07}
    Cycle 34: Accumulator_2 += {X_64,X_65,X_66,X_67} * {W_10,W_11,W_12,W_13}
    Cycle 35: Accumulator_3 += {X_74,X_75,X_76,X_77} * {W_14,W_15,W_16,W_17}

//End of a single pass
```

Listing 6.1 An example pass of the Cook-Toom accelerator

## 6.5 High-level flow: z- and xy-phase of computations

Until now I have focused only on the low-level flow of the MCMR scheme. In particular, I considered input that matches only the tile size of the Cook-Toom algorithm. But, the high-level flow of the computation is also very important. How does the Hadamard engine handle multiple regions and channels within the same convolution layer? This section explores the high-level flow of the MCMR scheme.

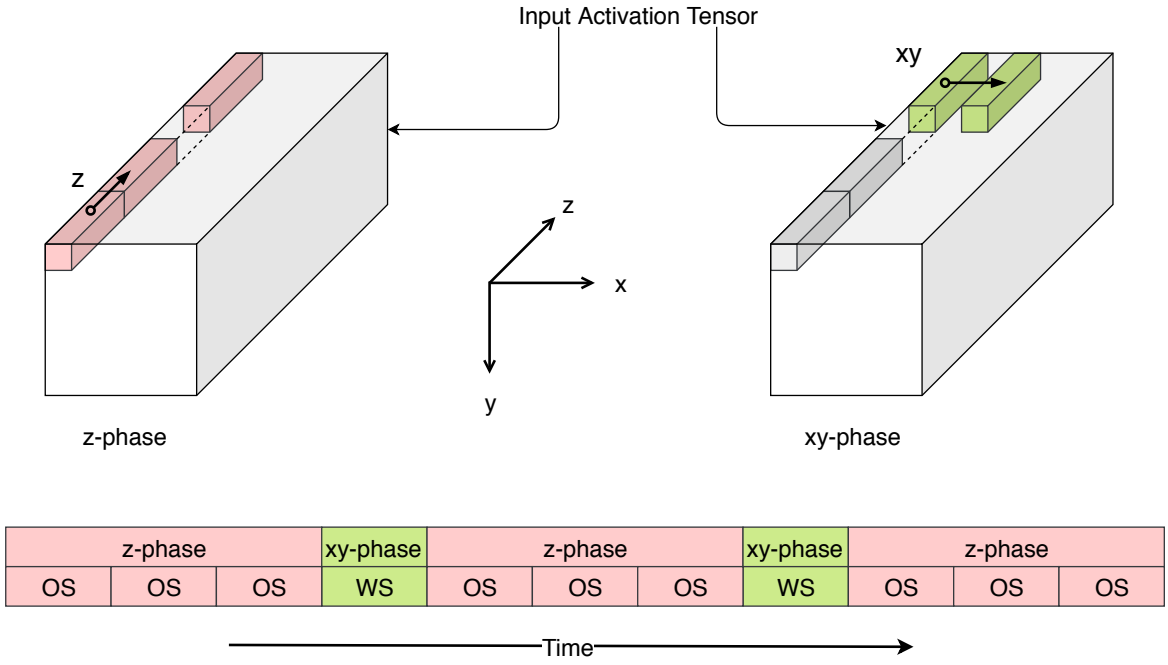



Fig. 6.12 High-level flow of the MCMR scheme

The Cook-Toom accelerator is flexible to handle both weight and output-stationary data-flow. Figure 6.12 illustrates the difference between the two different types of data flow. In the case of output-stationary data-flow, the weights are discarded after a while and a new set of weights for the same output channels but different input channels are loaded for further computation. Thus no weight reuse is possible. Although there is no weight reuse, in output-stationary data-flow, the cost of output activation storage stays fixed and minimal. On the contrary, in case of weight-stationary data-flow, although weights are reused, the partially computed output activations are temporarily saved half way through the MCMR scheme. A new set of output activations are loaded for different xy-pixels and once this is finished these sets of activations are saved again. If we want to avoid constant swapping between different output-activation maps, then the storage cost of output activation will rise significantly.

Therefore, for efficient implementation of the MCMR scheme, the data flow must alternate between output-stationary and weight-stationary data flow. Here, I introduce the term z-

phase and xy-phase computation. The terms are defined with respect to the axes of the 3-dimensional input activation tensor as shown in Figure 6.12. At the start, during the z-phase, the Cook-Toom accelerator loads a particular window of input feature maps, and continues loading new input feature maps towards the z-axis. Once all the feature maps corresponding to that particular window are finished, the accelerator moves temporarily to the xy-phase and chooses a new window for computation. After that the accelerator repeats the z- and xy-phase alternatively until all the input feature maps for a particular layer are completed.

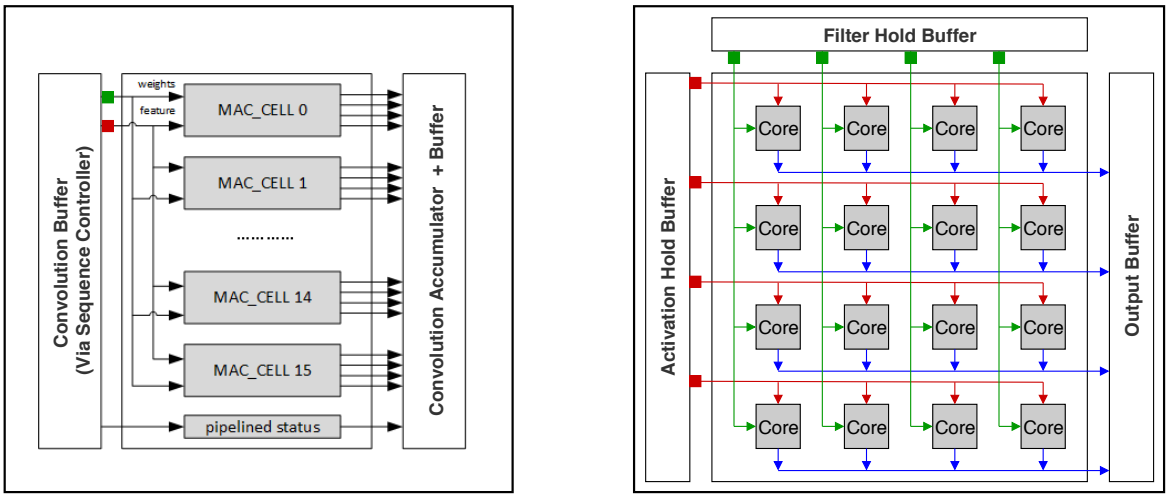



Fig. 6.13 On-chip interconnect in (a) NVDLA, (b) Cook-Toom accelerator

## 6.6 Evaluation

### 6.6.1 Evaluation setup and the baseline

To evaluate the performance of the Cook-Toom accelerator I measure the core MAC-array utilisation within the Hadamard Product engine. The core MAC-array utilisation provides us with an insight on how well the data is reused within the accelerator. The higher the core utilisation, the better it is. As a baseline comparison, I compare a mid-sized Cook-Toom accelerator (configuration: $< 16 \times 4 \times 16 \times 2 >$) with Nvidia's open source NVDLA embedded accelerator. Both the NVDLA accelerator and the mid-sized Cook-Toom accelerator consist of 512 KByte of convolutional buffer memory and 2,048 core MAC units which run at 1 GHz clock. As most of the die area of these accelerators are contributed by the cores and memory, they are very similar in size except the interconnect. Figure 6.13 compares

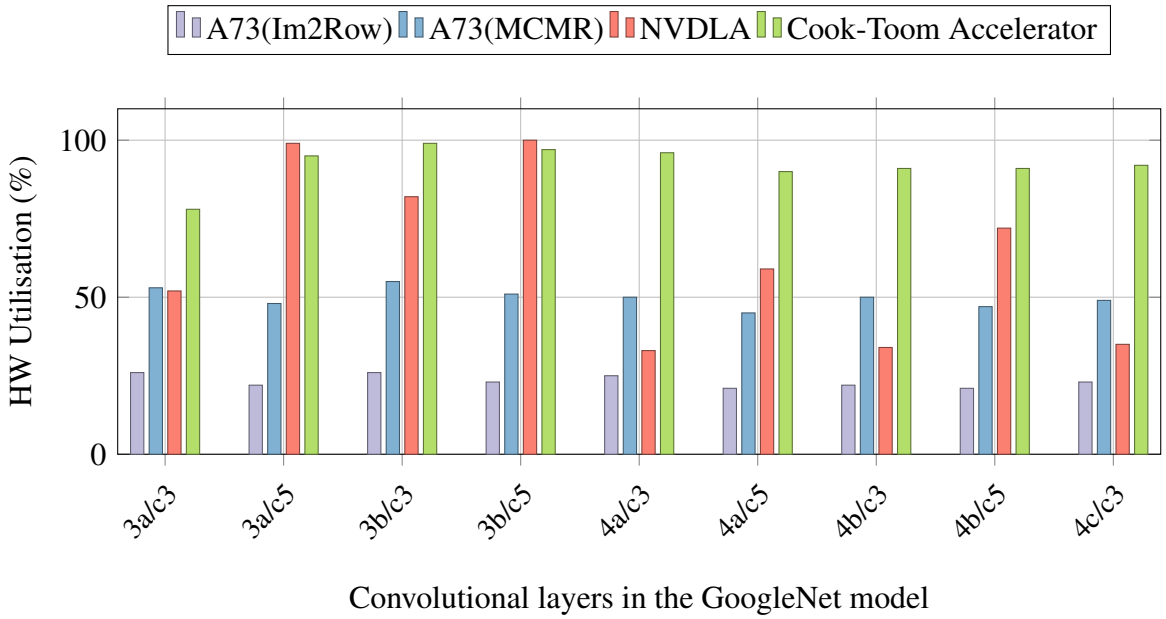



Fig. 6.14 Layerwise HW utilisation of the GoogLeNet with a batch size of 1 (Group-A)

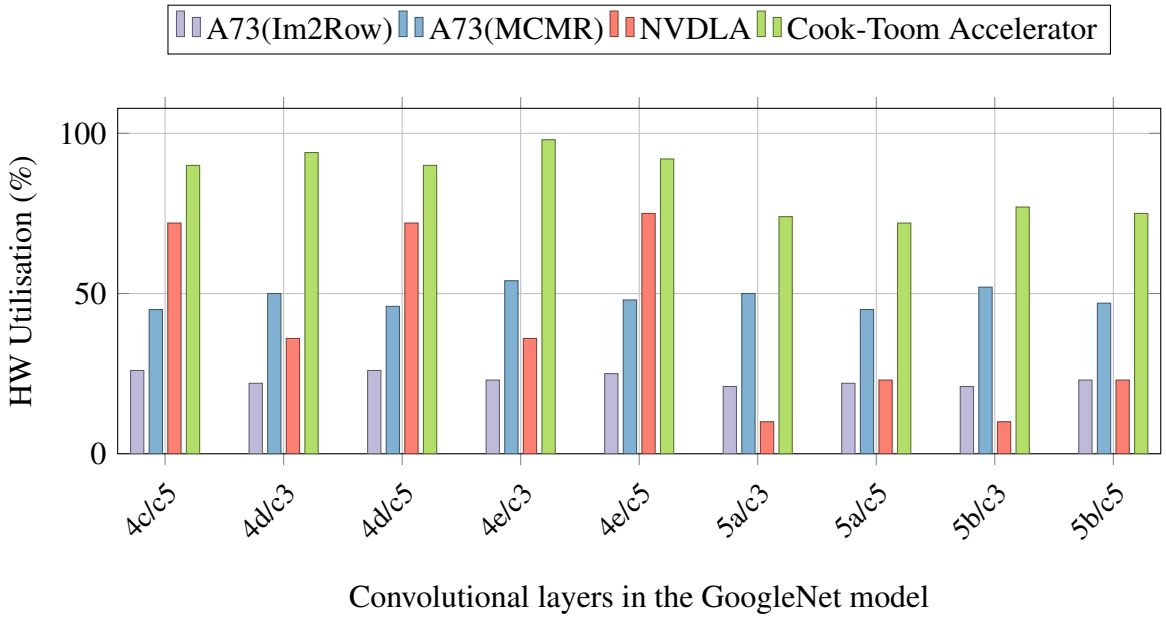



Fig. 6.15 Layerwise HW utilisation of the GoogLeNet with a batch size of 1 (Group-B)

the on-chip interconnect in the NVDLA and the Cook-Toom accelerator side-by-side. The on-chip network in the NVDLA accelerator consists of two broadcast networks dedicated for activations (or feature maps) and weights. Both the activations and weights are broadcast to each core (MAC_CELL in the figure). One of the challenges with broadcast networks is scalability when the number of cores grows. If the NVDLA accelerator is configured to contain a large number of cores, then the broadcast network will become slow. Furthermore, if a data-reuse pattern requires broadcasting to a cluster of cores, then multiple broadcasts are required, which is wasteful. In contrast, the on-chip interconnect in the Cook-Toom accelerator supports multiple 1D multicasts to different clusters of cores. This enhances the data-reuse capability of the accelerator. As an example, different clusters of cores now can work on different regions of the feature maps using the same set of filters. However, this organisation with a 1D multicast network adds some additional overheads to the design of the on-chip SRAM. To support multiple 1D multicasts, the SRAMs will require a large number of ports. Whereas the NVDLA requires one port for each type of data, the Cook-Toom accelerator requires $L$ ports for activation memory and $f_o$ ports for the filter memory. The value of $L$ depends on the configuration parameter lane, which defines the number of regions. The value of $f_o$ depends on the configuration parameter number of output channels that can be processed simultaneously. In Figure 6.13, the Cook-Toom accelerator consists of four ports in the activation SRAM memory which are marked in red, and four ports in the weight SRAM memory, which are marked in green. I also compare the results with the previously established ARM A73 SIMD benchmark. I considered three widely used ConvNets of different sizes and complexity, namely AlexNet, GoogLeNet, and ResNet50. Not only are these workloads good representatives of modern ConvNets, but their implementations are also available on the baseline NVIDIA's accelerator. Out of these three models, AlexNet is the largest (217MB), GoogLeNet is the smallest (7MB) and ResNet50 is between the two (98MB). But, in terms of required numbers of operations, GoogLeNet's compute complexity is higher than that of the other two. GoogLeNet is four times more computationally expensive than ResNet50 in spite of having a much smaller memory footprint. This is due to the branched nature of the GoogLeNet architecture. GoogLeNet also produces more activations (102 millions) due to its parallel architecture - almost five times that of AlexNet. Models that produce more intermediate activations are very challenging for an accelerator as this puts strain on on-chip memory. In fact how efficiently an accelerator can handle on-chip data often defines the performance of an accelerator.

There are two main ways a simulation model can be built for any accelerator - cycle-based simulation and event-based simulators. In traditional cycle-based simulators every component's operation method is called in every cycle which is wasteful. To this end, I built a

simulation environment based on events, where an event queue manages the processes in the simulator and a separate scheduler schedules the events. As an example, in the Cook-Toom simulator the core engines have compute functions that make calls and perform all the element-wise operations. At the end, an event is scheduled to call the next compute function in a clock period time only if the compute engine is ready with data. The hold buffer notifies the core engine when it is ready with the appropriate data that is requested by the core engine. Note that the simulator does not include a model of DRAM traffic. The simulator assumes that the on-chip hold buffers always can prefetch the necessary weights and activations. The Cook-Toom event-based simulator is built in the python language and borrows many ideas from the widely used GEM5 computer-system simulator [8].

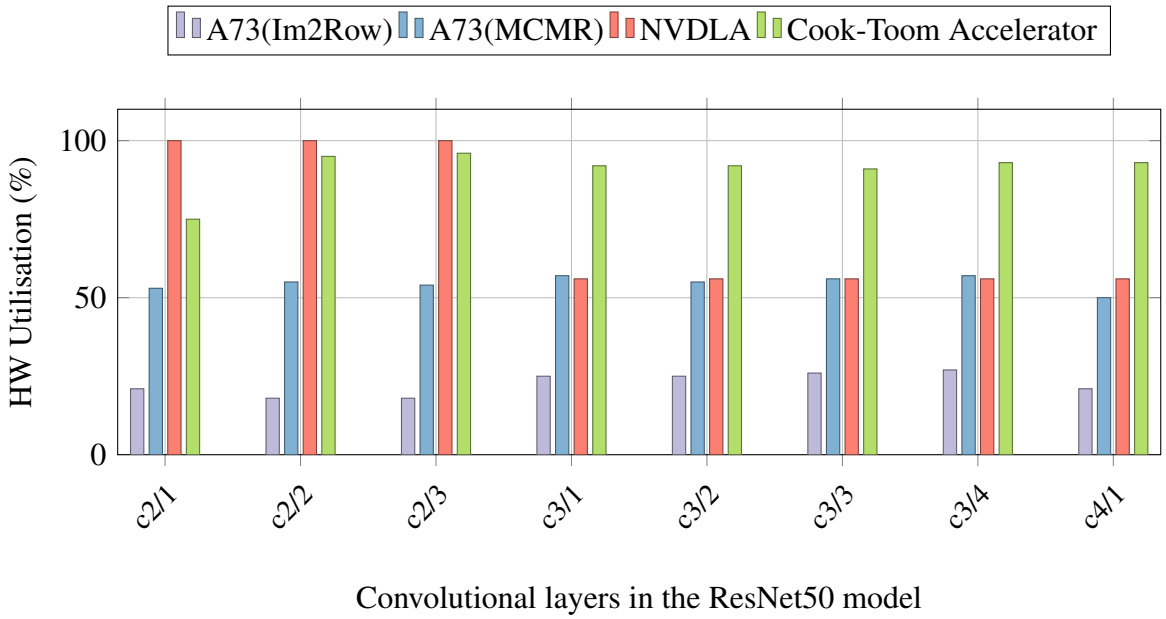



Fig. 6.16 Layerwise HW utilisation of the ResNet50 with a batch size of 1 (Group-A)

### 6.6.2 Performance analysis – per-layer

I now focus on core MAC-array utilisation of individual layers. I compare the mid-sized Cook-Toom accelerator with the similar sized NVDLA accelerator and the previously established benchmark from the Arm Cortex A73-SIMD implementation. Figures 6.14 and 6.15 present the per-layer core utilisation in all four solutions for the GoogLeNet model. As there are quite a few layers in this model, the graphs are divided into two groups, namely, group-A and group-B. As can be seen in the figure, except for the first few layers, the core utilisation in the Cook-Toom accelerator is consistently higher than that of NVDLA and others. In the

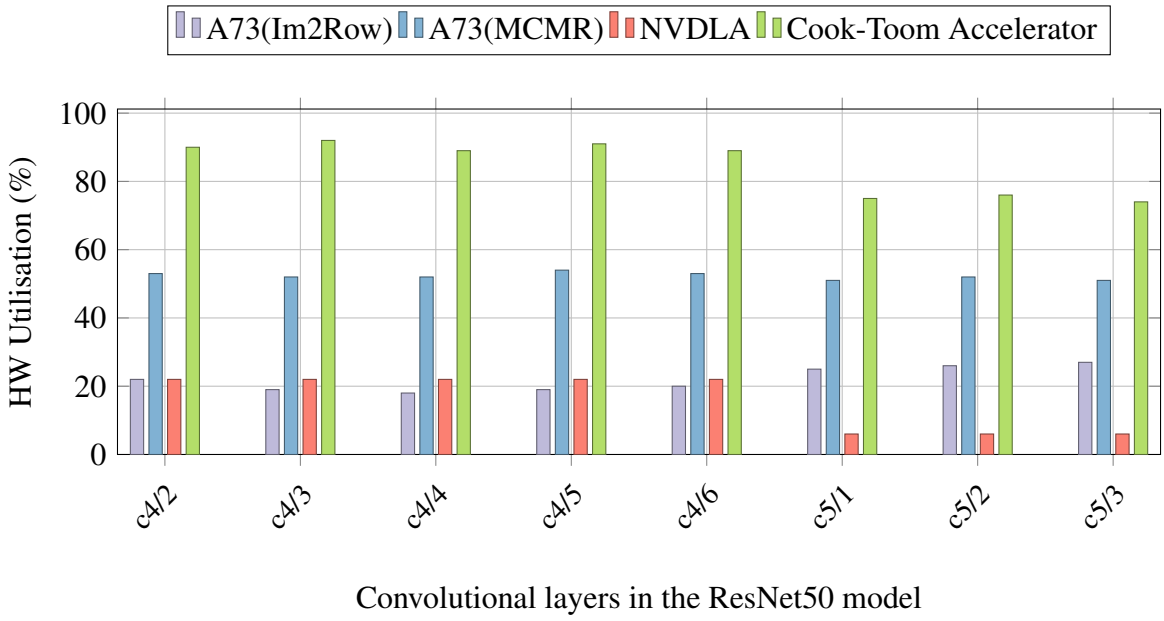



Fig. 6.17 Layerwise HW utilisation of the ResNet50 with a batch size of 1 (Group-B)

first few layers both the NVDLA and the Cook-Toom accelerator perform equally well. But due to the lack of enough data reuse the NVDLA performs poorly in the last few layers. As the Hadamard Engine can reuse data in four different dimensions using the hold buffer, the utilisation in the latter layers stays consistently high. Note that I am only comparing the convolutional layer here as the performance of these layers are our concern. Most of these convolutional layers consist of dense $3 \times 3$ and $5 \times 5$ layers. Another reason why NVDLA does not perform well is because it does not accelerate the $5 \times 5$ layers with Winograd convolution. Overall, both the NVDLA and the Cook-Toom accelerator outperform an A73-based SIMD implementation. The A73 processors are very resource efficient and thus lack sufficient on-chip storage to allow enough data reuse. In this case, after every few cycles of computation, the SIMD registers must be set up for a new chunk of input activations. Often the data does not fit the on-chip storage and therefore partially complete results must be swapped. This problem of low resource utilisation is resolved to some extent in the MCMR based implementation of the A73 solutions.

Figures 6.16 and 6.17 illustrates the per-layer core utilisation in all four platforms for the widely used ResNet50 model. Similar to the case of the GoogLeNet workload, utilisation is very high in the first few layers in the NVDLA. But, when it comes to the deeper layers the utilisation drops significantly in case of the NVDLA accelerator. In contrast, the Cook-Toom accelerator consistently maintains its utilisation in almost all the convolutional layers. Both the MCMR algorithm and the hold buffers help to maximise data reuse in the Hadamard

product engine. The core utilisation of the NVDLA accelerator is very low in the last few convolutional layers. From the public NVDLA resources, it is not clear why this is the case. Once reason could be that the feature-map dimensions at this stage become so small that these do not occupy the NVDLA cores enough. Another aspect to note is that in many layers the A73 SIMD-based solution achieves a comparable data reuse as in the NVDLA accelerator. Overall, NVDLA still outperforms the A73 SIMD-based solutions.

The Cook-Toom accelerator's configuration must be decided carefully at design time keeping in mind the workload. The mid-sized configuration that I use in the last two cases does not scale well with large and odd sizes of feature map. I demonstrate this problem with the AlexNet model. Figure 6.18 illustrates the layerwise core utilisation in all four platforms for AlexNet. As can be seen, the utilisation for the first layer in the Cook-Toom accelerator is poor. The first layer of AlexNet consists of large filters $11 \times 11$ which are applied on large feature maps. The overall utilisation in the Cook-Toom accelerator improves in the deeper layers. Again, in the deeper layers the core utilisation is low for the NVDLA accelerator. The Cook-Toom accelerator maintains high utilisation overall compared to other accelerators.

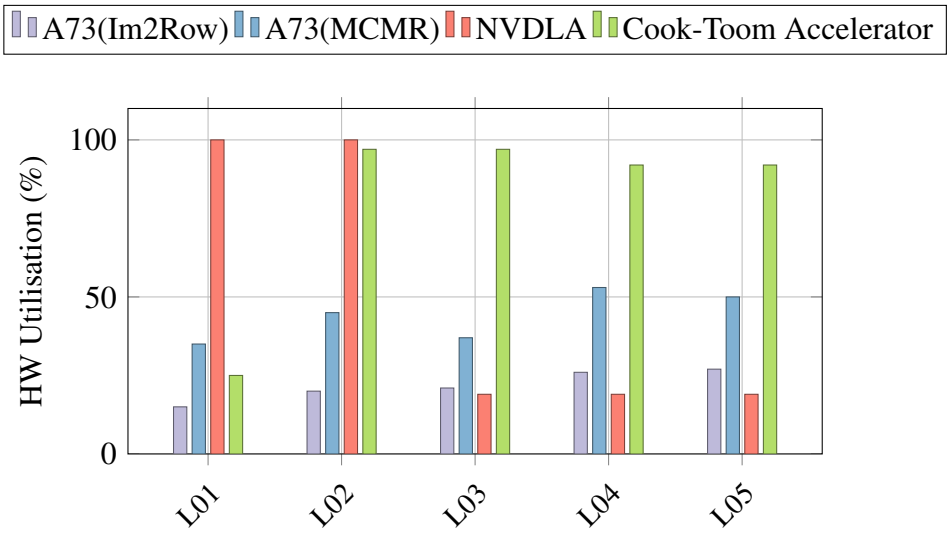



Fig. 6.18 Layerwise HW utilisation of the AlexNet with a batch size of 1

### 6.6.3 Performance analysis – whole network

To analyse the performance of the Cook-Toom accelerator, I also measured the average core utilisation of whole model. To this end, I collated the layer-wise utilisation of each

model under investigation and compared them for all the target platforms - A73 with the im2row scheme, A73 with the MCMR scheme, the NVDLA accelerator, and the Cook-Toom accelerator. Figure 6.19 illustrates the average utilisation of all the models. As can be seen from the figure, the average hardware utilisation in case of the Cook-Toom accelerator is much higher than the others in case of GoogLeNet and ResNet50. Also, note that the average utilisation is very much as per the A73 solution that uses the MCMR scheme. This is very significant as it shows that the architecture alone is not the deciding factor of how well the overall algorithm behaves. The MCMR scheme on the Cook-Toom algorithm helps to improve the performance of the SIMD machines. Furthermore, co-design of the MCMR scheme and the Cook-Toom accelerator helps to build a very efficient solution that allows maximising the resources in use. For AlexNet, the utilisation in the Cook-Toom accelerator is not that significant compared to the NVDLA accelerator. This is due to the mismatch between the model and the Cook-Toom accelerator configuration. Instead of a mid-sized Cook-Toom accelerator, a large Cook-Toom accelerator improves the utilisation of the AlexNet model. As discussed earlier, the configuration parameters of the Cook-Toom accelerator must be decided by keeping the workload in mind. Although a mid-sized Cook-Toom accelerator can handle most of the latest ConvNets, some of the old larger models would require a different configuration of the accelerator. Table 6.2 compares absolute run time of full networks. The Cook-Toom accelerator achieves an average 30% improvement over the NVDLA accelerator. Also, note the difference in run time of AlexNet between two different Cook-Toom accelerator configurations as explained earlier.

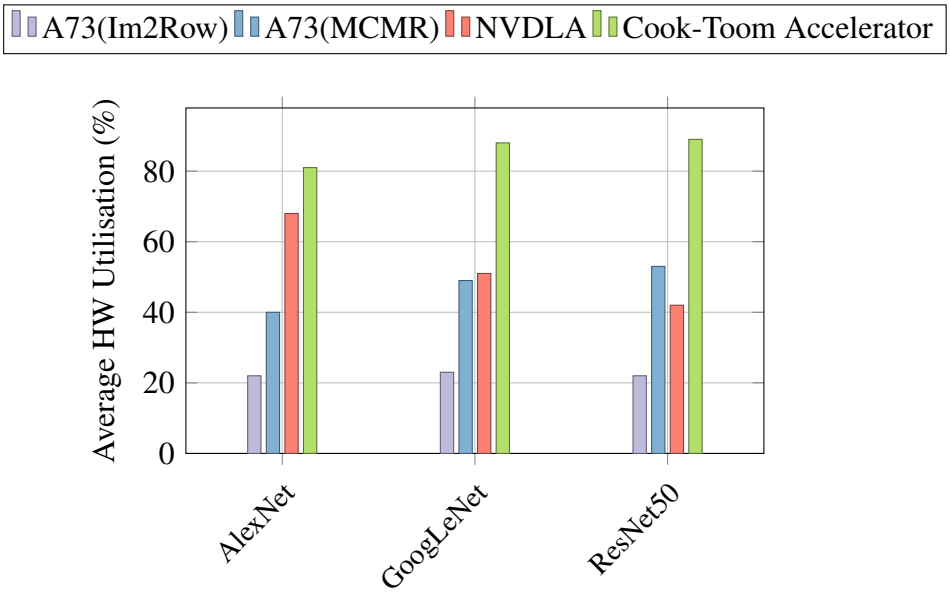



Fig. 6.19 Average HW utilisation of selected ConvNets with a batch size of 1

Table 6.2 Summary of **mean absolute runtime** of the **whole-network** in milliseconds (msec) for batch size of 1 in AlexNet, GoogLeNet and ResNet

| | AlexNet (Accl Size M) | AlexNet (Accl Size L) | GoogLeNet | ResNet50 |
|---|---|---|---|---|
| NVDLA (msec) | 0.9 | 0.9 | 1.22 | 7.11 |
| Cook-Toom (msec) | 0.8 | 0.65 | 0.96 | 5.41 |
| Absolute Speedup (msec) | 0.1 | 0.25 | 0.26 | 1.7 |
| Speedup (by %) | 12.5% | 38% | 27.1% | 31.4% |

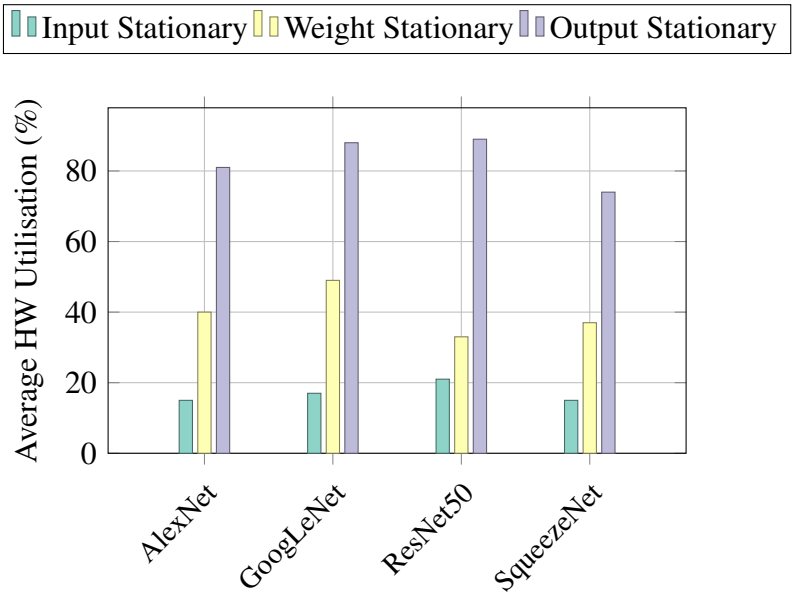



Fig. 6.20 Comparison between different data-reuse patterns - IS, WS, and OS

### 6.6.4 Comparison between different data-reuse patterns

The Hadamard product engine supports all three different types of data-reuse pattern, namely, input stationary (IS), weight stationary (WS) and output stationary (OS). This is achieved with the help of multiple broadcast buses and the hold buffers. Earlier in this chapter, I discussed that for an efficient implementation of the Cook-Toom algorithm an output-stationary reuse pattern is best. This was an argument from a theoretical stand point which needs practical validation. In this section, I compare the average utilisation of the MAC array inside the Hadamard product engine for four widely used ConvNet models of different sizes and complexities, namely, AlexNet, GoogLeNet, ResNet50, SqueezeNet. I implemented them using all the three data-reuse patterns and measured the average MAC utilisation in the Hadamard engine. Figure 6.20 illustrates the average hardware utilisation of the three

data-reuse patterns for all the four models under investigation. The results show us that the core utilisation is very poor in the case of the input-stationary reuse pattern. This is due to the fact that in the input-stationary reuse pattern input activations are held on the buffer and weights with respect to multiple output channels are applied on the input activations. This scheme produces many partial output-channel sums which overflow the on-chip storage. As a result, the partially computed maps must be swapped with new ones to carry on computing. The continuous map movement becomes very time consuming and power hungry. As a result, the MAC array often has to wait for data to be available for computation and overall resource utilisation drops significantly. In the case of weight-stationary, the problem shifts to the input activations. In this scheme, the input feature maps continuously need to be moved to maximise the weight reuse. This also results in poor MAC-array utilisation. The situation is slightly better than that of the input stationary as large feature maps can stay longer on the hold buffer. For the MCMR scheme, the best reuse pattern is output stationary. As can be seen in the figure, for all the models the output-stationary reuse pattern achieves the best core utilisation. This is possible as every output region is loaded only once to complete convolutions for all the corresponding input feature maps.

### 6.6.5 Scalability property of the Hadamard engine

One of the key properties of domain-specific accelerators is the ability to scale easily. Without scalability the same architecture cannot be extended for different power-performance-area and cost points. The architecture of the Hadamard engine is designed from the ground up by keeping scalability in mind. The Hadamard engine can be scaled up and down easily by choosing appropriate configuration parameters, namely, $f_i, f_o, T$, and $L$. Table 6.3 illustrates the three main configurations of the Hadamard engine targeting different performance requirements. The medium configuration can process images at 4TOP/sec. While evaluating various benchmark workloads, I used the medium configuration of the accelerator. The small engine can process 1TOP/sec and can be targeted for smaller or tiny networks (such as keyword detection). The large configuration can handle larger networks such as AlexNet, VGG16 and can process at 8TOP/sec.

## 6.7 Discussion

Co-design of the Cook-Toom accelerator and the MCMR algorithm helps to significantly speed up the dense portion of modern deep convolutional neural networks. As deep models keep evolving one of the common questions arises often on generalisation of the accelerator.

Table 6.3 Typical configurations of a Cook-Toom accelerator

| Cook-Toom Accelerator | Small | Medium | Large |
|---|---|---|---|
| Accelerator Configuration | $< 8 \times 4 \times 8 \times 2 >$ | $< 16 \times 4 \times 16 \times 2 >$ | $< 16 \times 4 \times 32 \times 2 >$ |
| Clock Frequency | 1 GHz | 1 GHz | 1 GHz |
| Performance (Peak Throughput) | 1 TOP/sec | 4 TOP/sec | 8 TOP/sec |
| Activation Hold Buffer Size | 1 KB | 2 KB | 2 KB |
| Weight Hold Buffer Size | 1 KB | 4 KB | 8 KB |
| On-chip SRAM | 512 KB | 1 MB | 2 MB |

In this section, I consider one of the emerging convolutional layer types - $1 \times 1$. A $1 \times 1$ convolutional layer is not only used on its own, it's also one of the phases of the widely used depthwise separable convolution. I discuss the possibility of accelerating such layers using the same Hadamard engine without any need for architectural change. I then conclude this chapter with a few more observations on domain-specific accelerator design.

### 6.7.1 Support for widely used 1x1 convolution in the Cook-Toom accelerator

The architecture of the modern convolutional neural network has been changing rapidly in recent years. In fact during my whole PhD research phase, I observed a gradual evolution of the deep network models from simple sequential ones to more complex ones that consist of branches. Researchers are continuously trying to come up with more efficient models that can be implemented in hardware. The types of convolutional layers have evolved a lot. Out of many types, $1 \times 1$ convolutional layers are becoming more common which helps to reduce the dimensions of the layers. While I built the Cook-Toom accelerator, one interesting question arose. Is it possible to implement $1 \times 1$ convolutional layers efficiently in the Cook-Toom accelerator? In this section, I answer this particular question. The $1 \times 1$ convolution is probably the simplest convolution that can be easily implemented in the existing Cook-Toom accelerator. In Listing 6.2, I show an example program for performing $1 \times 1$ convolution. In this case, I used a Cook-Toom accelerator configuration $< 8 \times 4 \times 8 \times 1 >$ which is simple enough to demonstrate. The Hadamard engine still reads a buffer of $8 \times 8 \times 8$. In this case, one pass phase takes 16 cycles as shown in the pseudo-code.

### 6.7.2 Optimisation using a mixed-precision data path

For inference using a convolutional neural network, 8-bit fixed-point or even lower precision has been proved to be sufficient to maintain high accuracy [38]. However, when it comes

```
// Example convolution program for (8 X 8 X 8) input tensor and 1 X 1 tile
// Accelerator Configuration <8, 4, 8, 1>

          Cycle 0: Accumulator_0 += {X_00,X_01,X_02,X_03} * w
          Cycle 1: Accumulator_1 += {X_04,X_05,X_06,X_07} * w
          Cycle 2: Accumulator_2 += {X_10,X_11,X_12,X_13} * w
          Cycle 3: Accumulator_3 += {X_14,X_15,X_16,X_17} * w
          Cycle 4: Accumulator_4 += {X_20,X_21,X_22,X_23} * w
          Cycle 5: Accumulator_5 += {X_24,X_25,X_26,X_27} * w
          Cycle 6: Accumulator_6 += {X_30,X_31,X_32,X_33} * w
          Cycle 7: Accumulator_7 += {X_34,X_35,X_36,X_37} * w
          Cycle 8: Accumulator_8 += {X_40,X_41,X_42,X_43} * w
          Cycle 9: Accumulator_9 += {X_44,X_45,X_46,X_47} * w
          Cycle 10: Accumulator_10 += {X_50,X_51,X_52,X_53} * w
          Cycle 11: Accumulator_11 += {X_54,X_55,X_56,X_57} * w
          Cycle 12: Accumulator_12 += {X_60,X_61,X_62,X_63} * w
          Cycle 13: Accumulator_13 += {X_64,X_65,X_66,X_67} * w
          Cycle 14: Accumulator_14 += {X_70,X_71,X_72,X_72} * w
          Cycle 15: Accumulator_15 += {X_74,X_75,X_76,X_77} * w

//End of a single pass
```

Listing 6.2 An example pass of the Cook-Toom accelerator

to the Cook-Toom class of algorithm, the accuracy of the model is significantly impacted by the straightforward application of quantisation. To tackle this, a mixed-precision data path can be used. The fused-MAC implementation used in the Hadamard engine is one such example. In this case the data path is widened slowly at every step. Initially, both the activation and filter weights are fed using an 8-bit representation. The multiplier produces 16-bit results which are then accumulated in 32-bit. The gradual widening of the data path helps to prevent any overflow during the computation and at the same time reduces the need for a constant end-to-end 32-bit wide data path. In the case of the Cook-Toom algorithm, the input transforms are very sensitive to quantisation. To avoid this, input transforms of the following stage are always computed in the previous stage before the activations are ready for further element-wise operations.

### 6.7.3 Overhead of on-chip interconnect in the accelerator

The Cook-Toom accelerator supports a wide range of data-reuse patterns. This helps in an efficient implementation of the MCMR scheme and enables support for many emerging layer types. However, this comes at the cost of increased complexity of the on-chip interconnect and SRAM memory interface. The weight bus (or the W-bus) needs to multicast the parameters

to multiple cores within a cluster. The activation bus (or the X-bus) also needs to multicast the activations to multiple compute clusters that correspond to multiple output channels. To support multiple 1D multicasts, the SRAM interface design becomes complex. To enable multi-lane data reuse, multiple ports must be opened at the SRAM interface. However, the complexity of the SRAM interface can be reduced by building a hierarchical on-chip interconnect that shares the same port at the SRAM interface. In summary, although there is an area overhead associated with the complex on-chip interconnect, compared to that of the cores and the memory it is not significant.

### 6.7.4 Summary

The Cook-Toom custom accelerator illustrates that co-designing algorithm and underlying hardware architecture helps to achieve significant performance improvement. It requires more collaborative effort between machine-learning researchers and computer architects. In this chapter, I illustrated how this could be achieved by first establishing design guidelines. Any domain-specific architecture, including the Cook-Toom accelerator, must be flexible, configurable and scalable. Through a variety of benchmarking exercises, I showed that the novel Cook-Toom accelerator is easily configurable and scalable. Additionally, the Cook-Toom accelerator can be configured to support any of three main data-reuse patterns. For efficient implementation of the MCMR algorithm, the data-reuse pattern must be altered while switching regions within the input feature maps as explained earlier. Additionally, I illustrated a few mapping examples of spatial convolutional layers onto the core Hadamard engine. Finally, I concluded this chapter with a brief discussion on possible support for emerging convolutional layers and the use of mixed-precision data paths in the design of the core unit.

# Chapter 7

# Conclusion

In order for deep learning to fulfil its promise in many industries, it is necessary to be able to deploy deep neural-network-based inference on low-power embedded system. Implementing a compute-heavy deep model on resource-constrained devices has its own unique set of challenges and requirements. Furthermore, the continuous evolution of the architecture of deep models make the situation even more challenging by making it necessary to balance flexibility against specialisation to avoid the inability to adapt in the near-to-long term. Embedded systems have several factors to consider that are related to the system's physical limitations. There are frequently inflexible requirements regarding size, memory, power, thermal limit, longevity and of course, cost. In the midst of balancing all of the requirements and concerns for an embedded application, there are a few important factors to consider when implementing solutions to execute deep neural-network-based inference for the edge:

- **Consider generalisation over specialisation:** In order for solutions to be applicable for a broader class of models, generalisation should be preferred over specialisation. Although some degree of specialisation helps, over specialisation may hurt generalisation. For example, excessive quantisation (say 4-bits) may be effective in one particular case. However, any hardware which is just specialised to run a network capable of such a level of quantisation will not be able run other models where higher degrees of quantisation hurt accuracy. In a similar manner, the use of a specialised activation function, uncommon layer structures, insertion of masks in the network to reduce computations, restricting higher-magnitude activations at the post-training stage may reduce the overall scope of such solutions. While developing ADaPT, 1D-FALCON and MCMR schemes, I always aimed for generalised solutions and avoided any specialisation that may restrict the applicability to only a few cases. The evaluation using a wide variety of benchmark supports this hypothesis.

- **Choose co-optimisation over isolated solutions:** There has been a lot of research in recent years on optimisation and implementation of deep neural networks. Many of these works were undertaken in isolation without considering the implications for the complete hardware/software stack. Machine-learning researchers came up with interesting model-optimisation techniques without considering their effects on hardware implementations. Similarly, computer architects implemented models provided by machine-learning communities without much feedback. To achieve optimum performance, design of models, algorithm and underlying hardware architecture must be done together. The Cook-Toom acceleration scheme introduced in this dissertation is a solid example of such co-design. On one hand the base Cook-Toom model has been modified to suit the underlying hardware and memory layouts. On the other hand, innovative low-level schemes were developed to map the high-level algorithm to the core computation structure. Furthermore, the Cook-Toom custom accelerator was developed to address limitations in existing low-power processors. The results from various experiments show model-algorithm-architecture co-design can bring significant improvement in performance of deep neural-network-based inference solutions.

- **Accelerate the most commonly occurring structures:** Despite the interest and the volume and velocity of work that has been produced recently, the fundamental understanding of deep-learning architecture is still in a nascent state. Thus, the research community continues to explore and propose new efficient model architectures. It is almost impossible to address the performance implications of the newer model structures immediately. Fortunately, much of the baseline architecture of the still evolving deep neural networks, especially convolutional neural networks, stayed the same. This involves spatial convolution with small filters, which is illustrated in Table 2.3 in Chapter 2. While implementing models for any specific target hardware, researchers should address the performance of those common computations first. This would not only benefit a wider class of architectures but also in the long run the optimisation would stay relevant. The solutions I developed during my PhD mainly aim at the most commonly occurring small spatial filters. The evaluation using deep models of different complexities and sizes shows that the impact from such careful optimisation is significantly higher and still relevant today.

- **Ensure flexibility to support emerging models:** Designing a custom convolutional neural-network inference accelerator that meets the demands of a constantly changing architecture space can be very challenging. In order to stay relevant in the longer term the custom architecture must be flexible and configurable. Overspecialised and

fine-tuned architectures may be very efficient in the short term but may not be an economical solution. As the deep models evolve over time, newer custom architecture specialisation will win over the previous ones. To meet the needs of industry, the architecture must be able to support the majority of the workloads. The Cook-Toom custom accelerator I designed from the ground up follows this basic principle. It does the necessary job efficiently and is not overspecialised to support anything specific, which is uncommon. The Cook-Toom accelerator is very efficient in computing small spatial convolutional layers using the fast arithmetic algorithm. It does so using the multi-channel multi-region scheme, which is described in detail in Chapter 5. For future extension, it also supports all three major data-reuse patterns. It is scalable and configurable to support different sizes and shapes of convolution. The results from the evaluation show that due to its simplified but versatile design, it wins over an NVDLA accelerator that is designed by industry. Due to its flexibility and configurability it can support all types of one and two-dimensional small filters. In contrast, Nvidia's NVDLA accelerator only can support one specific type of square filter that is common in modern ConvNets.

- **Aim for the right performance point:** Designing a novel custom architecture for emerging neural-network workloads can be overwhelming as performance targets vary a lot between applications. One general custom architecture may not be suitable for every application. Once the scope of the deep-learning application is known, it becomes important to understand how much processing performance is necessary to satisfy the application's needs. This could be difficult to understand when it comes to deep learning because so much of the performance is application specific. As an example, the performance of a ConvNet that classifies objects on a video stream depends on which type of layers are used in the model, how many layers there are in the model, the resolution of the video, the frames per second requirement, how many bits are used for processing - to name just a few. In an embedded system, however, it is very important to start with an initial estimate on performance needed. Adopting an excessively high-performance processor will naturally come at the cost of increased power, area and cost. Although a custom solution may be capable of running a ResNet50 model at 60fps for HD (720p/1080p) resolution in a centralised server, it's likely overkill for an application that will run a more embedded-friendly model on a stamp-sized region of interest. The Cook-Toom accelerator designed in this research is targeted purely at the low-power embedded space running inference between 1 to 8TOP/sec.

## 7.1 Future research scope

### 7.1.1 Extension of the ADaPT scheme using tensor decomposition

The novel ADaPT scheme helps to reduce the compute complexities of the convolutional layers by decomposing filters. This is done using singular value decomposition of matrices in two dimensions. Tensor arithmetic deals with higher dimensions. A two dimensional convolutional layer is a 4 dimensional tensor - $col \times rows \times input \times output$. The idea behind tensor decomposition is to decompose the high-dimensional tensor into several smaller tensors. The convolutional layers can then be approximated by several smaller convolutional layers. There has been some research done on the CP (canonical polyadic) and Tucker decomposition as I mentioned in the evaluation section under Chapter 3. But, the field remains almost untouched primarily due to the fact that tensor algebra is not a popular tool yet except in some areas of physical science. But tensor decomposition or sometimes called higher-order SVD, still remains an attractive direction for the ADaPT research work.

### 7.1.2 Extension of the Cook-Toom algorithm for efficient training

The Cook-Toom class of fast arithmetic helps to reduce compute complexity of convolutional layers during inference. This lossless scheme is very effective for small filters which are widely used in modern convolutional neural networks. The same technique can also be extended to train deep networks from scratch. To enable training using the Cook-Toom class of arithmetic a specialised Cook-Toom layer may be required. But, fundamentally designing a Cook-Toom layer is challenging. The linear transform that maps parameters from spatial domain to the Cook-Toom's alternative domain is non-invertible. Thus, any training method that needs to make use of both sets of parameters in conjunction will cause inconsistency. In recent years, there were a few efforts made to establish similar training methodologies. But, there was no significant progress made in this direction. Cook-Toom-based training still remains an attractive future research direction and an extension from my work is very much possible.

### 7.1.3 Mixed-precision implementation on a low-power Arm processor

In the past many researchers have proposed low- and mixed-precision alternatives to perform deep-learning inference. Some of the new custom accelerators from industry also support 8- or 16-bit implementation of many widely used models. Since Arm's resource-constrained processors have optimised support for 8- or 16-bit arithmetic, it would be practical to

implement inference using less precision wherever possible. To this end, I have conducted a preliminary investigation on how this could be done on an existing Arm processor. The main challenges behind this is the instability due to loss of precision during computation which I mentioned earlier. The Cook-Toom class of fast convolutions are very sensitive to precision, especially the input and output transform stages. One way to tackle this would be to derive alternative input and output transforms that are less sensitive to low precision and compromise the total number of multiplication required per convolution. Mixed-precision implementation of the Cook-Toom class of convolution using alternative transforms could significantly improve the overall performance of inference further.

### 7.1.4 Extension of the custom accelerator to support other emerging types of convolutional layers

The custom Cook-Toom accelerator I proposed and designed is very efficient in processing small spatial filters using the MCMR scheme. A study of efficiency of other types of convolutional layers on similar architecture could be an interesting direction of research. As an initial effort, I have already investigated the next widely used convolutional layer pattern of $1 \times 1$. It turns out the emerging $1 \times 1$ convolutional layers are also very efficient on the custom architecture and a brief discussion on this can be found in Chapter 6. There are a few more other types of convolutional layers which are also occasionally found in various emerging ConvNet architectures. As mentioned in Chapter 2 that many of those emerging layer architectures have very low arithmetic intensity. Thus accelerating such layers could be very challenging even on a custom accelerator. Is a flexible architecture like the Cook-Toom accelerator efficient for such layers or do we need a completely different type of architecture? Finding answers to such question could be an interesting area of further research.

# References


[1] Arm (2018). Hc30_arm_2018_08_17. https://www.hotchips.org/hc30/2conf/2.07_ARM_ML_Processor_HC30_ARM_2018_08_17.pdf. (Accessed on 07/02/2019).

[2] Arm-Ltd. (2017a). Arm architecture reference manual armv8, for armv8-a architecture profile.

[3] Arm-Ltd. (2017b). Compute library arm developer. https://developer.arm.com/technologies/compute-library. (Accessed on 03/28/2018).

[4] Arm-Ltd. (2019a). Ip products | arm ml processor – arm developer. https://developer.arm.com/ip-products/processors/machine-learning/arm-ml-processor. (Accessed on 06/18/2019).

[5] Arm-Ltd. (2019b). What's powering artificial intelligence. (Accessed on 06/07/2019).

[6] Astrid, M. and Lee, S. (2017). Cp-decomposition with tensor power method for convolutional neural networks compression. *CoRR*, abs/1701.07148.

[7] Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305.

[8] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. (2011). The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7.

[9] Cai, H., Zhu, L., and Han, S. (2018). Proxylessnas: Direct neural architecture search on target task and hardware. *CoRR*, abs/1812.00332.

[10] Cavigelli, L., Gschwend, D., Mayer, C., Willi, S., Muheim, B., and Benini, L. (2015). Origami: A convolutional network accelerator. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, GLSVLSI '15, pages 199–204, New York, NY, USA. ACM.

[11] Chen, L.-C., Collins, M., Zhu, Y., Papandreou, G., Zoph, B., Schroff, F., Adam, H., and Shlens, J. (2018). Searching for efficient multi-scale architectures for dense image prediction. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 8699–8710. Curran Associates, Inc.

[12] Chen, W., Wilson, J. T., Tyree, S., Weinberger, K. Q., and Chen, Y. (2015). Compressing neural networks with the hashing trick. *CoRR*, abs/1504.04788.

[13] Chen, Y., Emer, J., and Sze, V. (2017). Using dataflow to optimize energy efficiency of deep neural network accelerators. *IEEE Micro*, 37(3):12–21.

[14] Chen, Y., Krishna, T., Emer, J. S., and Sze, V. (2016a). 14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *2016 IEEE International Solid-State Circuits Conference, ISSCC 2016, San Francisco, CA, USA, January 31 - February 4, 2016*, pages 262–263.

[15] Chen, Y. H., Emer, J., and Sze, V. (2016b). Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture, ISCA*, pages 367–379.

[16] Chintala, S. (2016). soumith/convnet-benchmarks: Easy benchmarking of all publicly accessible implementations of convnets. https://github.com/soumith/convnet-benchmarks. (Accessed on 09/21/2016).

[17] Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1800–1807.

[18] Cong, J. and Xiao, B. (2014). *Minimizing Computation in Convolutional Neural Networks*, pages 281–290. Springer International Publishing, Cham.

[19] Courbariaux, M. and Bengio, Y. (2016). Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830.

[20] Courbariaux, M., Bengio, Y., and David, J. (2014). Low precision arithmetic for deep learning. *CoRR*, abs/1412.7024.

[21] Cun, Y. L., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In *Advances in Neural Information Processing Systems*, pages 598–605. Morgan Kaufmann.

[22] Dally, B. (2011). Power, programmability, and granularity: The challenges of exascale computing. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 878–, Washington, DC, USA. IEEE Computer Society.

[23] Denil, M., Shakibi, B., Dinh, L., Ranzato, M., and de Freitas, N. (2013). Predicting parameters in deep learning. In *Proceedings of the 26th International Conference on Neural Information Processing Systems*, NIPS'13, pages 2148–2156, USA. Curran Assoc. Inc.

[24] Denton, E., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R. (2014). Exploiting linear structure within convolutional networks for efficient evaluation. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'14, pages 1269–1277, Cambridge, MA, USA. MIT Press.

[25] Dieleman, S., Fauw, J. D., and Kavukcuoglu, K. (2016). Exploiting cyclic symmetry in convolutional neural networks. *CoRR*, abs/1602.02660.

[26] Dos Santos, C. N. and Gatti, M. (2014). Deep convolutional neural networks for sentiment analysis of short texts. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 69–78, Dublin, Ireland. Dublin City University and Association for Computational Linguistics.

[27] Dos Santos, C. N. and Zadrozny, B. (2014). Learning character-level representations for part-of-speech tagging. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages II–1818–II–1826. JMLR.org.

[28] Du, Z., Fasthuber, R., Chen, T., Ienne, P., Li, L., Luo, T., Feng, X., Chen, Y., and Temam, O. (2015). Shidiannao: Shifting vision processing closer to the sensor. *SIGARCH Comput. Archit. News*, 43(3):92–104.

[29] Eckart, C. and Young, G. (1936). The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218.

[30] Farabet, C., Martini, B., Corda, B., Akselrod, P., Culurciello, E., and LeCun, Y. (2011). Neuflow: A runtime reconfigurable dataflow processor for vision. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR Workshops 2011, Colorado Springs, CO, USA, 20-25 June, 2011*, pages 109–116.

[31] Feurer, M. and Hutter, F. (2019). *Hyperparameter Optimization*, pages 3–33. Springer International Publishing, Cham.

[32] Flagel, L., Brandvain, Y., and R Schrider, D. (2019). The unreasonable effectiveness of convolutional neural networks in population genetic inference. *Molecular biology and evolution*, 36:220–238.

[33] Frison, G., Kouzoupis, D., Zanelli, A., and Diehl, M. (2017). BLASFEO: basic linear algebra subroutines for embedded optimization. *CoRR*, abs/1704.02457.

[34] Frumusanu, A. (2016). Performance, power, area & closing thoughts - the Arm cortex A73 - artemis unveiled. https://www.anandtech.com/show/10347/arm-cortex-a73-artemis-unveiled/3. (Accessed on 03/21/2018).

[35] Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A. C., and Bengio, Y. (2013). Maxout networks. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1319–1327.

[36] Google (2016). How to quantize neural networks with tensorflow. https://www.tensorflow.org/versions/master/how_tos/quantization/index.html. (Accessed on 09/09/2016).

[37] Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. (2015). Deep learning with limited numerical precision. *CoRR*, abs/1502.02551.

[38] Gysel, P., Motamedi, M., and Ghiasi, S. (2016). Hardware-oriented approximation of convolutional neural networks. *CoRR*, abs/1604.03168.

[39] Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. (2016). EIE: efficient inference engine on compressed deep neural network. *CoRR*, abs/1602.01528.

[40] Han, S., Mao, H., and Dally, W. J. (2015a). Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149.

[41] Han, S., Pool, J., Tran, J., and Dally, W. J. (2015b). Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS'15, pages 1135–1143, Cambridge, MA, USA. MIT Press.

[42] Han, S., Pool, J., Tran, J., and Dally, W. J. (2015c). Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626.

[43] Hassibi, B. and Stork, D. G. (1993). Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems 5, [NIPS Conference]*, pages 164–171, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[44] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, abs/1512.03385.

[45] Hershey, S., Chaudhuri, S., Ellis, D. P. W., Gemmeke, J. F., Jansen, A., Moore, C., Plakal, M., Platt, D., Saurous, R. A., Seybold, B., Slaney, M., Weiss, R., and Wilson, K. (2017). Cnn architectures for large-scale audio classification. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*.

[46] Hinton, G. E. (2012). A practical guide to training restricted Boltzmann machines. In Montavon, G., Orr, G. B., and Müller, K.-R., editors, *Neural Networks: Tricks of the Trade (2nd ed.)*, volume 7700 of *Lecture Notes in Computer Science*, pages 599–619. Springer.

[47] Horowitz, M. (2014). Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers, ISSCC*, pages 10–14.

[48] Hummel, R., Lowe, D., and of Mathematical Sciences. Computer Science Department, C. I. (1987). *Computing Large-kernel Convolutions of Images*. Number nos. 253-264 in Computing Large-kernel Convolutions of Images. New York University.

[49] Hutter, F., Kotthoff, L., and Vanschoren, J., editors (2018). *Automated Machine Learning: Methods, Systems, Challenges*. Springer. In press, available at http://automl.org/book.

[50] Iandola, F. N., Moskewicz, M. W., Ashraf, K., Han, S., Dally, W. J., and Keutzer, K. (2016). Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360.

[51] Jaderberg, M., Vedaldi, A., and Zisserman, A. (2014). Speeding up convolutional neural networks with low rank expansions. *CoRR*, abs/1405.3866.

[52] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA. ACM.

[53] Kalchbrenner, N., Grefenstette, E., and Blunsom, P. (2014). A convolutional neural network for modelling sentences. *CoRR*, abs/1404.2188.

[54] Kim, Y. (2014). Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882.

[55] Kim, Y., Park, E., Yoo, S., Choi, T., Yang, L., and Shin, D. (2015). Compression of deep convolutional neural networks for fast and low power mobile applications. *CoRR*, abs/1511.06530.

[56] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012a). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[57] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012b). Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA. Curran Associates Inc.

[58] Lavin, A. and Gray, S. (2016). Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021.

[59] Le Cun, Y., Jackel, L. D., Boser, B., Denker, J. S., Graf, H. P., Guyon, I., Henderson, D., Howard, R. E., and Hubbard, W. (1990). Handwritten digit recognition: Applications of neural net chips and automatic learning. In Soulié, F. F. and Hérault, J., editors, *Neurocomputing*, pages 303–318, Berlin, Heidelberg. Springer Berlin Heidelberg.

[60] Lebedev, V. and Lempitsky, V. (2016). Fast convnets using group-wise brain damage. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2554–2564.

[61] Lecun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

[62] LeCun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. Morgan-Kaufmann.

[63] Lin, M., Chen, Q., and Yan, S. (2013). Network in network. *CoRR*, abs/1312.4400.

[64] Liu, B., Wang, M., Foroosh, H., Tappen, M., and Pensky, M. (2015). Sparse convolutional neural networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[65] Liu, H., Simonyan, K., and Yang, Y. (2018). DARTS: differentiable architecture search. *CoRR*, abs/1806.09055.

[66] Long, J., Shelhamer, E., and Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In *2015 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440.

[67] Maclaurin, D., Duvenaud, D. K., and Adams, R. P. (2015). Gradient-based hyperparameter optimization through reversible learning. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 2113–2122.

[68] Maji, P., Bates, D., Chadwick, A., and Mullins, R. (2017). Adapt: Optimizing cnn inference on iot and mobile devices using approximately separable 1-d kernels. In *Proceedings of the 1st International Conference on Internet of Things and Machine Learning*, IML '17, pages 43:1–43:12, New York, NY, USA. ACM.

[69] Maji, P. and Mullins, R. (2017). 1D-FALCON: Accelerating deep convolutional neural network inference by co-optimization of models and underlying arithmetic implementation. In Lintas, A., Rovetta, S., Verschure, P. F. M. J., and Villa, A. E. P., editors, *Artificial Neural Networks and Machine Learning - ICANN 2017 - 26th International Conference on Artificial Neural Networks, Alghero, Italy, September 11-14, 2017, Proceedings, Part II*, volume 10614 of *Lecture Notes in Computer Science*, pages 21–29. Springer.

[70] Maji, P. and Mullins, R. (2018). On the reduction of computational complexity of deep convolutional neural networks. *Entropy*, 20(4).

[71] Maji, P., Mundy, A., Dasika, G., Beu, J., Mattina, M., and Mullins, R. (2019). Efficient winograd or cook-toom convolution kernel implementation on widely used mobile CPUs. *arXiv preprint arXiv:1903.01521*.

[72] Miyashita, D., Lee, E. H., and Murmann, B. (2016). Convolutional neural networks using logarithmic data representation. *CoRR*, abs/1603.01025.

[73] Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. (2016). Pruning convolutional neural networks for resource efficient transfer learning. *CoRR*, abs/1611.06440.

[74] Nguyen, T. H. and Grishman, R. (2015). Relation extraction: Perspective from convolutional neural networks. In *Proceedings of the 1st Workshop on Vector Space Modeling for Natural Language Processing*, pages 39–48, Denver, Colorado. Association for Computational Linguistics.

[75] NVidia (2016). jetson_tx1_whitepaper.pdf. https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf. (Accessed on 09/20/2016).

[76] Nvidia (2017). Nvdla primer — nvdla documentation. http://nvdla.org/primer.html. (Accessed on 06/18/2019).

[77] Nvidia (2019). Deep learning performance guide :: Deep learning sdk documentation. https://docs.nvidia.com/deeplearning/sdk/dl-performance-guide/index.html#tensor-layout. (Accessed on 07/10/2019).

[78] Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., Song, S., Wang, Y., and Yang, H. (2016). Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 26–35, New York, NY, USA. ACM.

[79] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. (2016). Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279.

[80] Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2018). Regularized evolution for image classifier architecture search. *CoRR*, abs/1802.01548.

[81] Rigamonti, R., Sironi, A., Lepetit, V., and Fua, P. (2013). Learning separable filters. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2754–2761.

[82] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556.

[83] Society, R. (2017). The internet of things: Opportunities and threats | royal society. https://royalsociety.org/science-events-and-lectures/2017/10/tof-internet-of-things/. (Accessed on 07/01/2019).

[84] Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. A. (2014). Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806.

[85] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958.

[86] Sze, V. (2019). Balancing efficiency and flexibility for dnn acceleration. https://www.emc2-workshop.com/assets/docs/cvpr-19/sze-talk.pdf. (Accessed on 07/14/2019).

[87] Sze, V., Chen, Y., Yang, T., and Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. *CoRR*, abs/1703.09039.

[88] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. *CoRR*, abs/1409.4842.

[89] Temam, O. (2012). A defect-tolerant accelerator for emerging high-performance applications. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 356–367.

[90] Valenti, M., Squartini, S., Diment, A., Parascandolo, G., and Virtanen, T. (2017). A convolutional neural network approach for acoustic scene classification. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 1547–1554.

[91] van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A. W., and Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio. *CoRR*, abs/1609.03499.

[92] Vanschoren, J. (2018). Meta-learning: A survey. *CoRR*, abs/1810.03548.

[93] Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S., and LeCun, Y. (2014). Fast convolutional nets with fbfft: A GPU performance evaluation. *CoRR*, abs/1412.7580.

[94] Verbancsics, P. and Harguess, J. (2013). Generative neuroevolution for deep learning. *CoRR*, abs/1312.5355.

[95] Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In Dasgupta, S. and Mcallester, D., editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 1058–1066. JMLR Workshop and Conference Proceedings.

[96] Wang, Y. and Parhi, K. (2000). Explicit Cook-Toom algorithm for linear convolution. In *2000 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.00CH37100)*, volume 6, pages 3279–3282 vol.6.

[97] Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76.

[98] Wu, C., Buyya, R., and Ramamohanarao, K. (2016). Big data analytics = machine learning + cloud computing. *CoRR*, abs/1601.03115.

[99] Xu, X., Dehghani, A., Corrigan, D., Caulfield, S., and Moloney, D. (2016). Convolutional neural network for 3d object recognition using volumetric representation. In *2016 First International Workshop on Sensing, Processing and Learning for Intelligent Machines (SPLINE)*, pages 1–5.

[100] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2017). Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012.

[101] Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society, Series B*, 67:301–320.

# Appendix A

# Lagrange polynomial interpolation

Let $\beta_0$, $\beta_1$,...,$\beta_n$ be a set of $n+1$ distinct points, and let $p(\beta_k)$ for $k = 0,\ldots,n$ be given. There is exactly one polynomial $p(x)$ of degree $n$ or less that has value $p(\beta_k)$ when evaluated at $\beta_k$ for $k = 0,\ldots,n$. It is given by

$$p(x) = \sum_{i=0}^{n} p(\beta_i) \frac{\prod_{j \neq i}(x - \beta_j)}{\prod_{j \neq i}(\beta_i - \beta_j)} \tag{A.1}$$

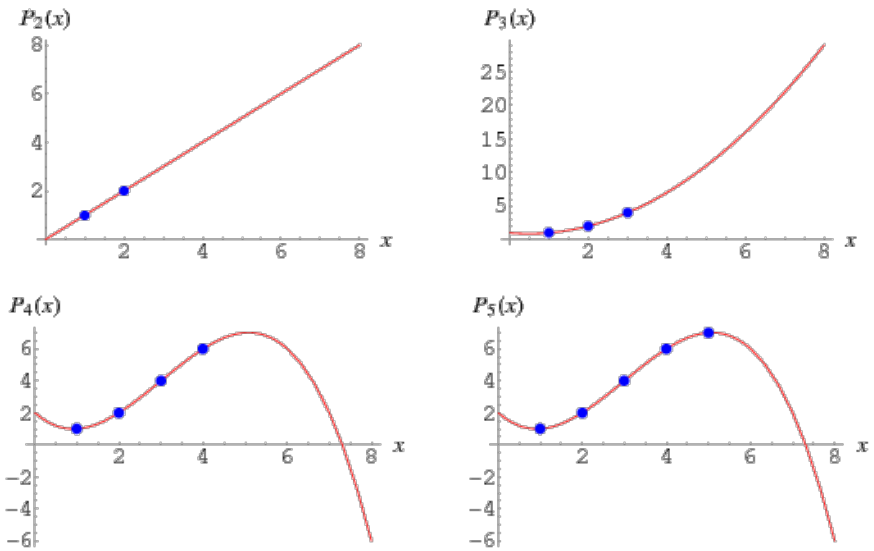



Fig. A.1 Lagrange interpolation polynomials

**Proof** The state polynomial $p(x)$ passes through the given points, as can be verified by substituting $\beta_k$ for $x$. Uniqueness follows because if $p'(x)$ and $p''(x)$ both satisfy the requirements and $P(x) = p'(x) - p''(x)$, then $P(x)$ has degree at most $n$ and has $n+1$ zeros at $beta_k$ for $k = 0,\ldots,n$. Hence $P(x)$ equals the zero polynomial.