Optimization of Compiler-Generated OpenCL CNN Kernels and Runtime for FPGAs

by

Seung-Hun Chung

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Optimization of Compiler-Generated OpenCL

CNN Kernels and Runtime for FPGAs

Seung-Hun Chung

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2021

This work explores the viability of end-to-end convolutional neural network inference using OpenCL HLS kernels generated from TVM on Intel FPGAs. We explore layer-pipelined execution for small networks and time-multiplexed kernels for larger CNNs. Naively generated kernels do not produce efficient hardware. We propose a set of optimizations to increase parallelism, resource utilization, and more efficiently use memory bandwidth. They include loop unrolling, tiling, fusion, invariant code motion, cached writes, CL channels, autorun kernels, concurrent execution, and parameterized kernels. These optimizations improve performance up to a factor of 1150x over the naive baseline implementation generated by TVM. Compared to Keras/Tensorflow on a 56-core Xeon 8280, we observe performance improvements up to 4.57x and 1.4x over LeNet and MobileNet but has a slowdown at 0.43x for ResNet-18/34.

Acknowledgements

I would like to thank my advisor, Tarek Abdelrahman, for his patient guidance in my research. I have greatly benefited from his insight and experience, and his support has made the completion of this work possible. I also thank Professors Paul Chow, Andreas Moshovos, and Wei Yu for their feedback on my thesis and presentation. Their feedback has helped me refine this work.

I extend my sincere thanks to my colleagues Lifu Zhang, Francis O'Brien, and Matthew Agostini for their feedback and company in the past several years.

Lastly, I would like to thank my family and friends. I am especially grateful to my parents, whose support and sacrifices have made it possible for me to pursue higher education, and to Nunu, for her kind patience and encouragement.

Contents

Chapter 1

Introduction

Deep learning is a machine learning method based on artificial neural networks with multiple layers that are used to extract higher-level features from the input data [44]. It is difficult to overstate the impact of deep learning on a number of disciplines, ranging from computer vision [54], speech/audio processing [24], natural language processing [17], and bioinformatics [59]. An abundance of data paired with significant leaps in computational power have caused the number of successful, practical applications for deep learning to increase dramatically.

In particular, *convolutional neural networks* (CNNs) have seen great success in image classification, with the ImageNet classification challenge [54] being a driving factor in advancing the state-of-the-art [9]. These advances have been substantial, with work published as early as 2015 [30] showing CNNs to surpass human performance at classifying ImageNet images.

However, to achieve such success, CNNs have become deeper with more layers and increasingly complex, incurring high computational costs. One study [61] argues that progress in all prominent deep learning applications heavily relies on advances in computational power, with graphical processing units (GPUs) and specialized neural processing accelerators[1] providing relief in the computational burden.

Field Programmable Gate Arrays (FPGAs) offer the unique opportunity to realize highly customizable accelerators that can be network-specific. Today, CNN accelerators on FPGAs are realized as either a *systolic array* of processing elements (PEs) that is time-multiplexed to perform tensor operations layer-by-layer [55], or as a *streaming architecture* made of hand-designed building blocks [63] (also known as *templates*). These accelerators are often implemented with RTL components [22, 52, 55, 65], with some using a higher level of abstraction that allows vendor tools to implement designs based on algorithmic descriptions [3, 27, 63, 66].

[1] See [58] for a list of ICs and IPs for AI/DL processing.

Systolic array accelerators rely on fixed data dimensions (i.e., input tensor dimensions) for high parallelism and reuse. This may result in sub-optimal PE utilization [12]. Further, since the same array is used for all layers, optimizations cannot be implemented at a per-layer granularity (e.g., layer-specific bit precision). In contrast, streaming accelerators can be tailored for a target network, but if a network operation cannot be implemented by existing building blocks, an additional hardware component must be designed. This requires additional time in development, which is not ideal for exploring different models.

This thesis proposes a compilation flow that generates streaming architecture accelerators without dependence on components from hand-designed hardware templates. The approach deploys deep neural networks from high-level descriptions in machine learning frameworks by automatically generating CNN kernels that are synthesized for FPGA through the use of Intel OpenCL tools and TVM, an open-source deep learning compiler. However, it is unclear if this approach is feasible, nor if it can deliver competitive performance. Thus, the overall goal of this work is to explore the feasibility and performance of such an approach.

1.1 Research Overview

In this thesis, we explore a novel approach in deploying floating-point[2] CNN inference accelerators by generating OpenCL kernels using a deep learning compiler. Specifically, frozen graph-based representations of neural networks are imported into TVM, where each node in the computation graph is executed by an OpenCL kernel. These OpenCL kernels are automatically constructed by TVM's code generator. The kernels are then synthesized by the Intel OpenCL for FPGA compiler (AOC) for a target FPGA platform. Each kernel represents a different set of CNN layer(s), and inputs and activations are streamed through kernels during execution.

This approach offers an *end-to-end* deployment of CNNs[3]. That is, it starts from a CNN model and produces an accelerator that can be used for inference without hardware programming on the part of the user. More specifically, it offers the following advantages:

- It generates CNN kernels using a compiler, which allows for rapid deployment of CNNs without exposing the hardware design process to the user. Further, it generates kernels on a per-layer basis, allowing both network and layer-specific optimizations.

[2]We implement the accelerators in floating-point for generality and due to lack of support for reduced bit precision representation in TVM, although this is an area of active development. However, this approach is not fundamentally limited to floating-point representation.

[3]This approach is not limited exclusively to CNNs, although it is the focus of our study.

- It uses an open-source ML compiler, which permits support for a range of common neural network operators, simplifying support for new operators/activations.
- It does not rely on hand-designed templates or building blocks since kerels are generated using a compiler. This allows the support of arbitrary operations, an existing limitation of streaming accelerators.

The above approach is not void of challenges. Kernels generated from TVMs perform poorly due to a lack of parallelism in the generated hardware or poor utilization of memory bandwidth. Worse, the kernels may not synthesize at all for larger networks where the design exceeds the target FPGA resources. Thus, we utilize a set of optimizations to improve kernel performance and implement them in TVM. These optimizations include: loop tiling, strip mining, unrolling, fusion, and caching reads/writes. To address resource limitations, we use TVM to parameterize kernels and re-use them across layers with the same operation. We also utilize and hand-apply OpenCL-specific optimizations such as channels, autorun kernels, and concurrent execution. In spite of these optimizations and given that the accelerator is made from generated HLS, accelerators generated with this approach are unlikely to perform as well as hand-optimized designs. Nonetheless, the approach can accommodate pre-production environments that benefit from fast prototyping and increased performance.

We demonstrate the viability of our approach by using it to deploy three CNNs on three target FPGAs. Specifically, we deploy LeNet-5, MobileNetV1, and ResNet (18/34) on an Intel Stratix 10 SX, Stratix 10 MX, and Arria 10 GX.

Our experimental evaluation shows that with LeNet, the generated accelerator achieves a maximum speedup of 10.34× over the TVM-generated naive implementation. For MobileNet and ResNet, our approach achieves improvements of 183.8× and 1009× over the naive implementations, respectively. This demonstrates the efficacy of our optimizations.

In a comparison with Keras/TensorFlow (a highly optimized ML framework), the LeNet accelerator is 4.57× and 3.07× better than an Intel Xeon 8280 (4 threads) and NVIDIA GTX 1060 respectively. The resulting MobileNet accelerator is 1.4× better than TensorFlow with 112 threads. However, for ResNet-18 and 34 respectively, the accelerator suffers a slowdown compared to TensorFlow with 112 threads and delivers 0.40× and 0.37× the performance. The main reason for the slowdown is that the generated accelerators are limited by the area consumed for loading weights and activations, which prevents full utilization of compute resources.

Further, we compare this work to existing approaches. We preface this comparison with a disclaimer that comparing to other work in this area is difficult due to a number of factors. These can include

differences in benchmark models, platforms and technology, batching, and bit precision. These factors should be considered for a comprehensive evaluation. Compared to Caffeinated FPGAs [18], a convolution engine implemented with hand-written OpenCL, this work performs 1.41× better in GFLOPS for 3×3 convolutions, although there is a five year gap in technology between their work and this work. Compared Tensorflow to Cloud FPGAs [27], which compiles models to an HDL abstraction, this work demonstrates a speedup of 3.23× for LeNet and comparable ResNet performance. Finally, compared to DNNWeaver [55], an accelerator framework implemented with a hand-optimized hardware library, our MobileNet accelerator delivers 0.11× the GFLOPS observed in their AlexNet accelerator.[4]

The evaluation in general shows that the optimizations we apply to the kernels significantly improve performance to the extent where speedups over consumer GPUs and server-grade CPUs can be demonstrated. However, more optimizations are required to close the performance gap between this work and hand-optimized FPGA neural processing architectures, particularly in the deployment of large CNNs. Despite the current limitations, the results of this prototype show promise in deploying successfully across a variety of devices and networks.

1.2 Contributions

The contributions of this thesis are:

- A compilation flow for single-image inference acceleration that relies on industry-standard tools, including TVM for CNN kernel generation and Intel OpenCL SDK for FPGAs for synthesizing OpenCL into hardware.
- Optimized schedules and/or parameterized kernels for convolutions, fully-connected layers, and softmax layers implemented with TVM schedule primitives. These schedules result in significantly increased kernel performance.
- An evaluation of the end-to-end deployment of three CNNs of varying sizes on 3 target FPGAs, demonstrating the portability and utility of our flow.

1.3 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 gives background material on CNNs, FPGAs, OpenCL, the Intel OpenCL FPGA compiler, and TVM. Chapter 3 describes our proposed

[4] MobileNet and AlexNet have significant differences in architecture, and thus, this is not a complete comparison for network performance but the closest one that can be made with our evaluations.

compilation flow. Chapter 4 catalogs the kernel optimizations used to improve performance. Chapter 5 describes the implementation of the optimized schedules in the TVM compiler. Experimental evaluation is presented in Chapter 6. Finally, Chapters 7 and 8 respectively review related work and provide concluding remarks with directions for future work.

Chapter 2

Background

This chapter gives background material for our work. Deep neural networks (DNNs) and convolutional neural nets (CNNs) are described in Section 2.1. Brief descriptions of FPGAs and FPGA CNN accelerators are provided in Section 2.2. The OpenCL standard (version 1.0) is described in Section 2.3, and how it is implemented by Intel for the FPGA OpenCL SDK is summarized in Section 2.4. Finally, an overview of functions and capabilities provided by TVM, an open-source deep learning compiler, is provided in Section 2.5.

2.1 Deep Neural Networks

2.1.1 Overview

Deep neural networks (DNNs) are a type of artificial neural network with one or more hidden intermediate layers between the input and output layers [23]. More specifically, the focus of this thesis is deploying feedforward (unidirectional) *convolutional neural networks* (CNNs) although the approach and the tools are not limited to CNNs.

CNNs are powerful in applications such as computer vision because they have far fewer parameters compared to their fully-connected (dense) layer counterpart. This makes CNNs easier to train and deploy in practice [45]. They scale well to large input sizes such as an image, and because they have fewer parameters they are less prone to overfitting. We describe the component layers common to a CNN, and briefly describe the architectures of LeNet [43], MobileNetV1 [32], and ResNet [31].

The computational complexity of CNNs is characterized by the number of operations and the number of parameters, which include weights and biases.

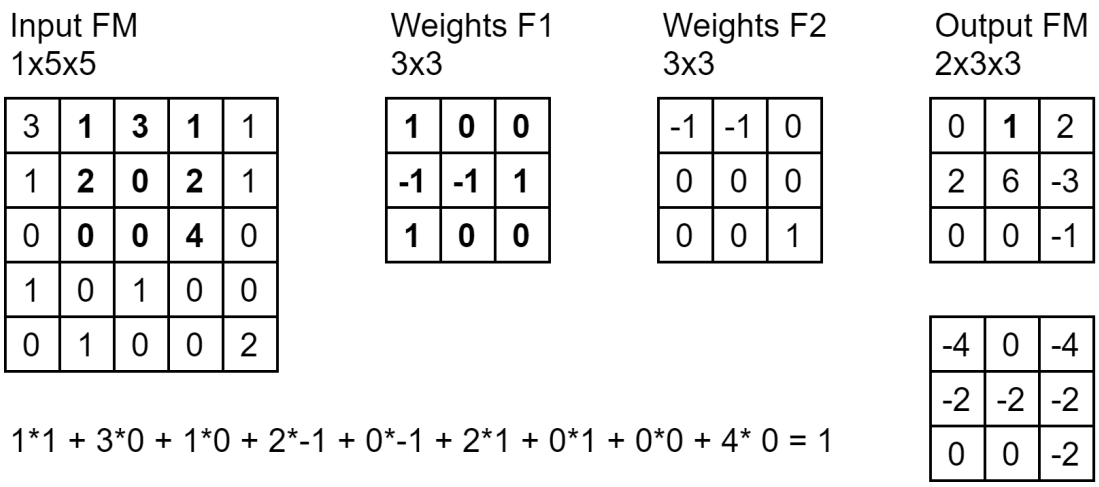

Figure 2.1: A sample 2-filter 3×3 convolution with a 5×5 input feature, resulting in an output feature map with dimensions 2×3×3. The calculation of one output is provided, with the relevant inputs and weights bolded.

2.1.2 CNN Layers

Convolutional layers

The convolution operation (or technically cross-correlation, the typical implementation of the convolution operation in neural network libraries [23]) takes two tensors[1] as input: a feature map I and a set of weights W, which is also known as kernels or filters. Typically, convolution requires four hyperparameters: number of filters (output channels) K, filter size F, stride S, and zero padding width P [45]. The output y, is referred to as the *output feature map*, also known as *output activation maps* or just simply *activations*.

The input feature map is a four-dimensional tensor with dimensions $N \times C_1 \times H_1 \times W_1$, where N is the batch size, C_1 is the number of input channels, H_1 is the input height, and W_1 is the input width. In our work, we do not extract parallelism from batching and assume[2] that $N = 1$. The weights are a four-dimensional tensor with the dimensions $K \times C_1 \times F \times F$. Convolution at point i, j is defined as:

$$y(i,j) = (W * I)(i,j) = \sum_{m}^{F} \sum_{n}^{F} I(i+m, j+n) W(m,n) \tag{2.1}$$

This is repeated for every element in I with i and j incrementing by S. The resulting output feature map is a four-dimensional tensor with dimensions $N \times C_2 \times W_2 \times H_2$ where:

[1] Tensors in deep learning frameworks are defined as multidimensional arrays with a uniform data type (e.g., float32).
[2] In real-world systems that process inference, requests should be processed as they are received, as argued in [21].

$$C_2 = K$$

$$H_2 = (H_1 - F + 2P)/S + 1$$

$$W_2 = (W_1 - F + 2P)/S + 1$$

We provide an example of a 2-filter ($K = 2$) 3×3 ($F = 3$) convolution in Figure 2.1 with $S = 1$ and $P = 0$. In this example, the input feature map has a shape of $1 \times 1 \times 5 \times 5$ and the weights have a shape of $2 \times 1 \times 3 \times 3$. The resulting output feature map has dimensions $1 \times 2 \times 3 \times 3$. A sample calculation of Equation 2.1 is provided in the figure and the relevant elements in the input feature map, weights, and the resulting output element have been bolded.

```
for (int ax1=0; ax1 < C_2; ax1++)
  for (int yy=0; yy < H_2; yy++)
    for (int xx=0; xx < W_2; xx++)
      for (int rc=0; rc < C_1; rc++)
        for (int ry=0; ry < F; ry++)
          for (int rx=0; ry < F; rx++)
            y[ax1][yy][xx] += I[rc][S*yy+ry][S*xx+rx] * W[ax1][rc][ry][rx];
```

Listing 2.1: A sample loop nest of 2D convolution

A loop nest that implements the above operation is provided in Listing 2.1. In this loop nest, it can be seen that the computational cost of convolution is $C_2 \times H_2 \times W_2 \times C_1 \times F \times F$. Indeed, convolutional layers make up the majority of computation time in convolutional neural networks since it typically has the largest amount of parameters [16]. Thus, when optimizing CNNs, convolutional layers should receive the most attention.

Depthwise separable convolutions [32], on the other hand, factorize standard convolutions into a depthwise convolution for filtering and a pointwise (1×1) convolution for combining outputs. In depthwise convolutions, a single filter is applied to every input channel, reducing the computational cost to $C_2 \times H_2 \times W_2 \times F \times F$. Following the depthwise convolution, the filtered outputs are combined to create new features by the pointwise convolutions, which have a complexity of $C_2 \times H_2 \times W_2 \times C_1$.

Pooling layers

Often inserted in between convolutional layers, a pool layer is used to decrease the number of parameters [45]. The two common pooling operations is *maxpool*, taking the maximum of a $F \times F$ region, and *avgpool*, taking the average of a $F \times F$ region. Since all computations are done with the input, there are no parameters in pooling layers. The input tensor has dimensions $C_1 \times H_1 \times W_1$ and with a region size of F and a stride S, the output tensor has dimensions:

$$C_2 = C_1$$

$$H_2 = (H_1 - F)/S + 1$$

$$W_2 = (W_1 - F)/S + 1$$

Fully-connected (dense) layers

The dense layer connects all neurons from one layer to all of the neurons to another layer, hence it is also commonly referred to as the fully-connected layer. The output of one neuron is a dot product between weights and the inputs. The input is typically two-dimensional, $N \times C_1$ and the weights are $C_2 \times C_1$, where N is the batch size and C_1 is the input dimension and C_2 is the output dimension. The output is $N \times C_2$. In practice, if the input is not one-dimensional, it is flattened before it is fed into a dense layer.

Activation functions

Often applied at the output of convolution and fully-connected layers, activation functions introduce non-linearity to a neural network. Two activation functions are used commonly: ReLU, and softmax. ReLU is a piecewise function to set all negative outputs to zero. It is defined as:

$$RELU(x) = max(0, x) \tag{2.2}$$

Modifiers can be made to ReLU, such as ReLU6, which is defined as:

$$RELU6(x) = max(6, x) \tag{2.3}$$

Typically used at the end of a neural network, softmax normalizes layer outputs to a probability (i.e., the sum of the output of softmax is 1). For some vector of size K, softmax is defined as:

$$S(x_i) = \frac{exp(x_i)}{\sum_j^K exp(x_j)} \tag{2.4}$$

with *exp* being the exponential function.

In practice (and in TVM), the maximum of the inputs is subtracted from each of the inputs in softmax implementations, i.e., softmax is computed by $S(z)$ where $z = x_i - max_i(x_i)$ to stabilize numerical computation against overflow/underflow [23].

Layer	LeNet-5	Output Size
conv1	3×3 conv, 6 filters, stride=1	6×26×26
pool1	2×2 maxpool, stride=1	6×13×13
conv2	3×3 conv, 16, stride=1	16×11×11
pool2	2×2 maxpool, stride=1	16×5×5
flatten	transform to 1D	400
dense1	120 hidden units	120
dense2	84 hidden units	84
dense3	10 hidden units, softmax	10

Table 2.1: LeNet architecture

2.1.3 LeNet-5

Published in 1998 by Yann LeCun [43], LeNet-5 is an early convolutional neural network used to classify digits from the MNIST database of handwritten digits. Originally, it is comprised of seven layers: convolution, subsampling, convolution, subsampling, convolution, fully connected, and the activation layer. In the original paper, LeCun uses Euclidean Radial Basis Function units for the output layer and sigmoidal functions for the intermediate activations. Since its publication both of these have fallen out of favor while softmax and ReLU became more popular activation functions for the output and intermediate layers respectively. In our experiments, we use ReLU in the convolution activation functions and softmax as the output layer. The layers of the LeNet architecture are listed in Table 2.1.

2.1.4 MobileNet V1

MobileNetV1 [32] is a CNN model published in 2017 that advanced the mobile state-of-the-art, achieving 70.6% Top-1 ImageNet accuracy with 4.2M parameters. It uses depthwise separable convolutions to reduce the number of parameters and the computation workload. 1×1 convolutions make up 94.86% of multiply-add operations in MobileNetV1, with depthwise 3×3 convolutions making up 3.06% and the rest by regular 3×3 convolutions and fully connected layers [32]. The network architecture is summarized in Table 2.2.

2.1.5 ResNet

ResNet [31], or deep residual networks, is the winning submission to the 2015 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [40] with a 3.57% Top-5 error on the ImageNet test set. The authors propose deep residual learning as a solution to the degradation problem, which is the observation that adding more layers to a sufficiently deep model leads to higher training error.

Layer	MobileNetV1	Output Size
conv_1	3×3 conv, 32 filters, stride=2	32×112×112
conv_2_dw	3×3 DW conv, stride=1	32×112×112
conv_2	1×1 conv, 64, stride=1	64×112×112
conv_3_dw	3×3 DW conv, stride=2	64×56×56
conv_3	1×1 conv, 128, stride=1	128×56×56
conv_4_dw	3×3 DW conv, stride=1	128×56×56
conv_4	1×1 conv, 128, stride=1	128×56×56
conv_5_dw	3×3 DW conv, stride=2	128×28×28
conv_5	1×1 conv, 256, stride=1	256×28×28
conv_6_dw	3×3 DW conv, stride=1	256×28×28
conv_6	1×1 conv, 256, stride=1	256×28×28
conv_7_dw	3×3 DW conv, stride=2	256×14×14
conv_7	1×1 conv, 512, stride=1	512×14×14
conv_8_dw conv_8	$\begin{bmatrix}\text{3×3 DW conv, stride=1}\\ \text{1×1 conv, 512, stride=1}\end{bmatrix}$ ×5	512×14×14
conv_9_dw	3×3 DW conv, stride=2	512×7×7
conv_9	1×1 conv, 1024, stride=1	1024×7×7
conv_10_dw	3×3 DW conv, stride=1	1024×7×7
conv_10	1×1 conv, 1024, stride=1	1024×7×7
pool	7×7 avgpool, stride=1	1024
fc	1000 hidden units, softmax	1000

Table 2.2: MobileNetV1 architecture

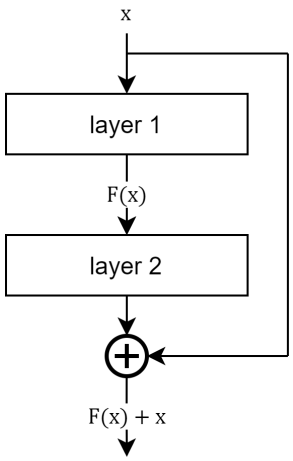

Figure 2.2: Residual block

Layer	ResNet-18	ResNet-34	Output Size
conv1	7×7 conv, 64, stride=2	7×7 conv, 64, stride=2	64×112×112
pool	3×3 maxpool, stride=2	3×3 maxpool, stride=2	64×56×56
conv2	$\begin{bmatrix}3\times3 \text{ conv, } 64, \text{ stride}=1\\ 3\times3 \text{ conv, } 64, \text{ stride}=1\end{bmatrix}\times2$	$\begin{bmatrix}3\times3 \text{ conv, } 64, \text{ stride}=1\\ 3\times3 \text{ conv, } 64, \text{ stride}=1\end{bmatrix}\times3$	64×56×56
conv3	$\begin{bmatrix}3\times3 \text{ conv, } 128, \text{ stride}=2\\ 3\times3 \text{ conv, } 128, \text{ stride}=1\end{bmatrix}\times2$	$\begin{bmatrix}3\times3 \text{ conv, } 128, \text{ stride}=2\\ 3\times3 \text{ conv, } 128, \text{ stride}=1\end{bmatrix}\times4$	128×28×28
conv4	$\begin{bmatrix}3\times3 \text{ conv, } 256, \text{ stride}=2\\ 3\times3 \text{ conv, } 256, \text{ stride}=1\end{bmatrix}\times2$	$\begin{bmatrix}3\times3 \text{ conv, } 256, \text{ stride}=2\\ 3\times3 \text{ conv, } 256, \text{ stride}=1\end{bmatrix}\times6$	256×14×14
conv5	$\begin{bmatrix}3\times3 \text{ conv, } 512, \text{ stride}=2\\ 3\times3 \text{ conv, } 512, \text{ stride}=1\end{bmatrix}\times2$	$\begin{bmatrix}3\times3 \text{ conv, } 512, \text{ stride}=2\\ 3\times3 \text{ conv, } 512, \text{ stride}=1\end{bmatrix}\times3$	512×7×7
pool	7×7 avgpool, stride=2	7×7 avgpool, stride=2	512
FC	512 hidden units, softmax	512 hidden units, softmax	1000

Table 2.3: ResNet architecture

ResNets consist of residual blocks (illustrated in Figure 2.2) that make use of shortcut connections or skip connections, where the connection is an identity mapping that is added to the outputs of stacked layers. A linear projection is required to match dimensions between $f(x)$ and x. These shortcut projections are performed by 1×1 convolutions [31].

Compared to regular CNNs, ResNets improve on model accuracy from having significantly increased depth while still having less complexity (i.e., less trainable parameters) than VGGNets [56], previously state-of-the-art and the winning submission of ILSVRC 2014. We list the layers of ResNet-18 and 34, which are the networks that we use in our experiments in Table 2.3.

2.2 Field Programmable Gate Arrays

2.2.1 Overview

Field Programmable Gate Arrays (FPGAs) are devices that contain a large number of logic blocks and programmable interconnects that can be re-configured to build various digital logic circuits. A typical logic block consists of a look-up table and registers, and repeats in both horizontal and vertical axes. Data is stored in registers and in hardened block RAM (BRAM) systems that are embedded within the FPGA fabric.

Typically, FPGAs also contain hardened blocks interleaved between the columns of logic blocks such as digital signal processor (DSP) blocks [33] or tensor blocks [37]. DSP blocks are a hardened implementation of floating-point operations such as float multiplication, add, and multiply-accumulate, and tensor blocks can harden matrix-matrix and matrix-vector multiplication. The use of hardened blocks over logic blocks saves logic resources and also allows the circuit operating frequency to be higher [33].

In this work, we use the Arria 10 and Stratix 10 families of Intel FPGAs which contain Adaptive Logic Modules (ALMs), composed of combinatorial look-up tables, dedicated adders, and registers [34]. ALMs are used to implement most of the kernel's logic functions and arithmetic operations. For reasons stated above, it is beneficial to map floating-point operations to DSPs, and indeed designers often opt for a DSP block to implement them.

FPGA circuit designs, and digital circuits in general, are typically written using hardware description languages (HDLs) such as Verilog [60], VHDL [4], or SystemVerilog [57], and given to an electronic design automation tool (e.g., Intel Quartus). Writing hardware designs in HDL, however, is an advanced and lengthy design process reserved for domain experts in hardware. To increase design productivity [49] and alleviate complexities in the hardware design flow, there has been considerable work in the recent decade on *high-level synthesis* (HLS) [8, 15, 47], which allows developers to use a higher level of abstraction such as C and C++ to describe circuit behavior. However, to achieve high-performance HLS implementations, optimizations that benefit hardware such as loop pipelining must be considered over conventional optimizations for software [47].

FPGA vendors have also made commercial HLS solutions available in addition to the academic work cited above, including Xilinx HLS and SDAccel, and Intel C++ HLS and OpenCL for FPGAs. In this work, we use the Intel OpenCL SDK for FPGAs, on which we elaborate in Section 2.4.

2.2.2 FPGA CNN Accelerators

Recent work has shown FPGAs to be a viable platform for accelerating compute workloads associated with machine learning (see [64] for a survey). FPGAs' programmability enable a higher level of versatility over their hardened counterparts such as GPUs or specialized ML processor architectures realized on application-specific integrated circuits. More specifically, FPGAs provide the following advantages:

- FPGAs can be re-configured with a circuit optimized for specific models, which can result in better hardware utilization and/or performance.
- FPGAs can implement networks with reduced/mixed precisions, which can improve performance by reducing memory footprint and using integer operations that are faster and less expensive to implement.
- FPGAs can achieve higher energy efficiency compared to GPUs which can make it suitable for low-power environments and less energy expended per operation [5].

Nonetheless, the use of FPGAs has several drawbacks.

- Designing an accelerator on an FPGA incurs long development cycles and requires expertise in digital circuit design. Indeed, this is a key barrier in the adoption of FPGA for accelerators to ML/software developers.
- There is a finite amount of on-chip resources, and external memory bandwidth is more restricted on FPGAs in comparison to GPUs, which can make deploying larger networks more challenging.
- Compared to GPUs, FPGA accelerators often lack supporting software infrastructure. This leads to a programmability gap between high-level machine learning frameworks and the hardware.

Early work in FPGA acceleration of CNNs [52, 67] implemented compute-intensive layers such as convolution and fully-connected layers and off-loaded their execution from the CPU to the FPGA. These accelerators are typically optimized for a specific target or network. They can be differentiated in two design dimensions: *architecture*, and *implementation*.

- **Architecture**: commonly a *single processing engine* [3, 21, 55, 66, 67], a *synchronous dataflow accelerator* [7, 22, 63], or a *vector processor* [19].

  Single processing engines often come in the form of an array of processing elements that are highly optimized for tensor operations and the target FPGA. Processor instructions orchestrate computations of the network, one layer at a time. A regular design such as a systolic array can scale to a target platform, but certain layers in the network may suffer from hardware underutilization [67].

In contrast, synchronous dataflow accelerators map each network layer to a dedicated compute unit. These compute units, often referred to as *templates* or *building blocks*, are hand-designed components in the accelerator hardware library that implement specific ML operations, such as convolutions. Once an arrangement of the building blocks is made, the inputs are streamed into the device. Accelerators of this type can exploit optimizations specific to the network, such as layer-specific vector sizes and bit precisions, provided that the building blocks can support these optimizations. However, finding an optimal mapping from the computation graph to these templates is more challenging since on-chip resource constraints must be considered. It is often necessary to time-multiplex dedicated units, also referred to as folding [63].

Finally, vector processors provide macro-instructions for the acceleration of CNN operations. An example is the ConvNet Processor [19], which is a programmable vector processor consisting of a vector arithmetic unit capable of performing 2D convolutions and dot products, and a soft processor that controls the vector unit's execution.

- **Implementation**: FPGA accelerator frameworks are built with RTL modules with hardware description languages (HDL) [52, 65] or high-level synthesis (HLS) [7, 22, 55, 66, 67].

  Implementing the accelerator with HDLs (e.g., Verilog, VHDL) can expose optimizations at the lowest level, providing the designer the ability to control the design at the granularity of cycles and directly optimize critical paths. Portability varies across designs, depending on how customized the design is for a target platform. For example, frameworks such as DNNWeaver [55] are capable of generating portable HDL for both Xilinx and Intel devices, whereas the RTL for DeepBurning [65] specifically targets Xilinx.

  In contrast, implementing designs in HLS can result in increased design productivity and benefit from portability granted from vendor-provided HLS tools. For example, Caffeine [66], fpgaConvNet [63] and FINN [7] synthesizes hardware using Xilinx Vivado HLS. Similarly, Intel DLA [3] is implemented in Intel OpenCL for FPGAs.

In existing work, deploying neural networks that utilize novel deep learning operations on these platforms can be difficult if the compute engine or the set of templates lack support for the operation. In this case, a module designed by hand must be added to the template library or the functionality of the compute engine must be modified.

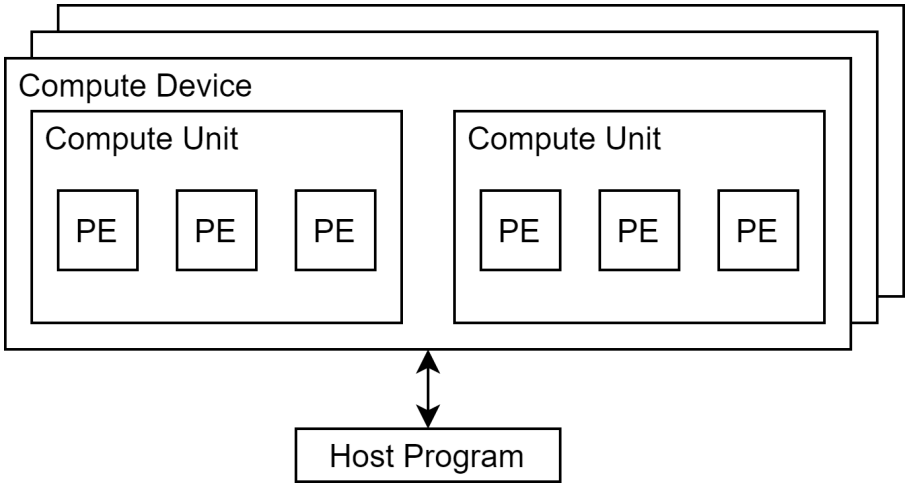

Figure 2.3: The OpenCL platform model

2.3 Open Computing Language (OpenCL)

Open Computing Language (or OpenCL) [25] is an open-source specification by the Khronos Group that provides a layer of abstraction for programmers to develop applications for heterogeneous parallel processing platforms. In practice, it is an API and extension of ISO C99 for parallel programming. Today, OpenCL supports a wide variety of computing platforms including but not limited to central processing units (CPUs), graphical processing units (GPUs), FPGAs, and DSPs.

The OpenCL standard consists of four models: the *Platform Model*, *Execution Model*, *Memory Model*, and the *Programming Model*. In the sections below, we briefly describe each model and how an OpenCL application is executed.

2.3.1 Platform Model

The OpenCL platform consists of one *host* and one or more *Compute Devices*. Devices contain *Compute Units* that contain *Processing Elements*. This hierarchy is visualized in Figure 2.3.

2.3.2 Execution Model

The execution model consists of two main components, the device *kernels* that defines the instructions for the computation and the *host program* which manages the execution of the compute devices.

The host program maintains a *context* which is the environment that contains key objects for execution. These objects include the *kernels*, which is the C code that defines computation, the *program*, a callable library of functions compiled from the kernel, *buffers* which represent a memory object that

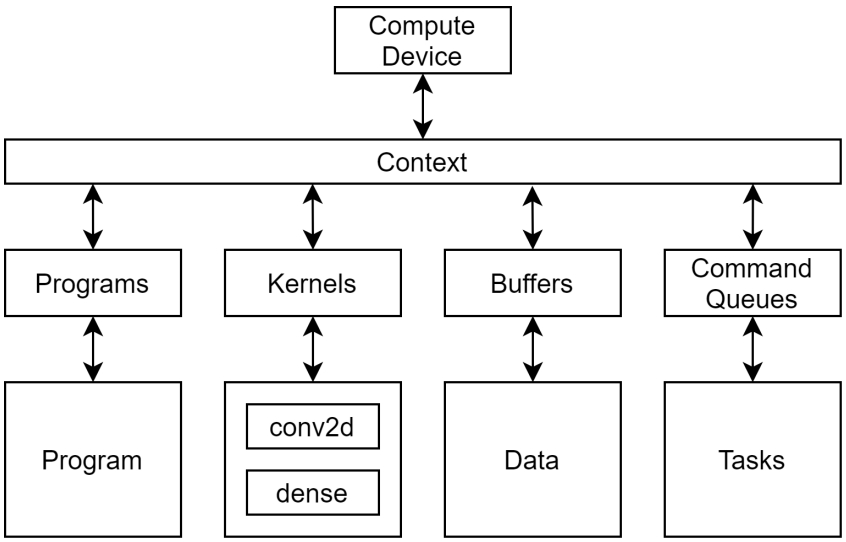

Figure 2.4: The OpenCL execution model

can be read and written by the host and device, and *command queues* which is a FIFO-queue of instructions (called *work-items*) from the host program that can be executed either in-order (commands block succeeding commands) or out-of-order (commands are non-blocking and any dependencies must be explicitly synchronized by the programmer) [26]. This is visualized in Figure 2.4.

In the specification, the smallest unit of work is referred to as a *work-item*, further grouped into *work-groups*, with a global work-item ID and a unique work-group ID. This index space is called an *NDRange*, an N-dimensional index space where work-items reside. This is expanded upon in Section 2.3.4.

2.3.3 Memory Model

There are four memory regions in the OpenCL memory model:

1. *Global memory*, a buffer that may be read/written by any work-item and both by the host and device.
2. *Constant memory*, a partition of global memory that remains constant during the execution and is visible to all workgroups.
3. *Local memory*, a buffer shared by work-items in the same work-group.
4. *Private memory*, memory private to a work-item.

Data is explicitly managed by the host program and the kernel. If there is data that resides within local or private memory, it must be copied to global memory to make that region of memory accessible by the host.

2.3.4 Programming Model

OpenCL provides two programming models, *Data Parallel* and *Task Parallel* [25]. In the Data Parallel model, kernels operate over NDRange in the units of work-items forming work-groups. In an explicit model, the programmer provides the number of work-items that may be executed in parallel and its divisions into work-groups. In an implicit model, just the number of work-items that may execute in parallel is declared and the OpenCL runtime may determine how it is divided across work-groups.

In the Task Parallel model, kernels execute on a compute unit as a single-work item. Instead of declaring a number of work-items that can be executed at the same time, parallelism is extracted by vectorized data types or enqueuing multiple tasks.

2.4 Intel's OpenCL FPGA Compiler

The Intel FPGA SDK for OpenCL is an implementation of the OpenCL specification to enable users to develop applications for FPGAs using a high-level language [36]. In this section, we describe a typical flow for development, how the Intel FPGA SDK implements OpenCL for FPGAs, and some considerations that should be made when compiling a high-level language for FPGAs compared to other, more conventional platforms.

2.4.1 Development Flow

To create an Intel FPGA OpenCL application, the user must develop a *host application* that is compiled on the host system, *OpenCL kernels* to be synthesized and programmed onto the FPGA by the offline compiler (*aoc*). A *custom platform* or *board support package* (BSP), developed by the vendor of the board to support OpenCL operations such as board programming, memory management, kernel execution, and other essential tasks, is also required [36].

The user provides OpenCL C kernels in a .cl file that is compiled by *aoc*, which generates a .aocx binary bitstream. This bitstream contains two components: the *static partition* (sometimes known in the literature as a *shell*), which implements lower-level logic such as instruction queues, kernel-to-buffer transfer, and vice versa and is part of the BSP; and the *kernel system*, which is the implementation(s) of the user kernel logic.

The host program, compiled by a standard C compiler, creates a context, a device, a CL program, command queues, and buffers necessary for execution. Upon creation of a CL program, the BSP partially reconfigures the FPGA with the new bitstream.

At this point, the host program may copy buffers into the device, set kernel arguments, and enqueue execution of kernels via the command queue(s). Once complete, the result of the kernel may be read back from global memory.

2.4.2 Mapping Kernel Code to FPGA Resources

The compiler implements constant, private, and local memory either in flip-flop registers or in embedded memory blocks. Global memory is implemented in the external memory(s) available to the device and supported by the board support package, which may include DDR4 or HBM2. Accessing data implemented in BRAMs and external memory requires the generation of load-store units (LSU), discussed in Section 2.4.3. LSUs are implemented with logic elements. If a cache is inferred for global memory access, BRAMs are used to implement it.

For constant, private, and local memory, the access pattern and the size of arrays declared in memory influences the compiler's decision to implement them in either registers of various configurations, including plain, shift, or barrel-shift registers, or in BRAMs [35]. If array accesses are not statically inferrable then for single-work item kernels, if the array is less than 64 bytes, it may be implemented in registers, otherwise in BRAMs [36]. For NDRange kernels, there is no size limit for registers. The designer may aid the compiler in inferring registers by unrolling loops to create static accesses to the array. Alternatively, the designer may explicitly add attributes to the buffer to force it to be implemented specifically by registers or BRAMs.

2.4.3 Accessing Memory

Overview

Depending on the storage scope, access pattern, stride, and size of access, the compiler selects an LSU type that it considers the most suitable [36]. There are four different LSUs to choose from: burst-coalesced, prefetching, streaming/semi-streaming, and pipelined LSUs. Each has different advantages and resource costs. The compiler makes an effort to select one that results in optimal performance[3].

Burst-Coalesced LSU

Burst-coalesced LSUs buffer requests until the largest burst can be made. They are the most commonly inferred LSU type in our experiments. This LSU type can provide highly efficient accesses to global

[3]In recent versions of Quartus, users can explicitly declare which LSU to select for a given load/store instruction, however, the Quartus versions required for the BSPs that we use in our experiments precede this feature.

memory, but consumes the most amount of resources on the FPGA, especially when a cached burst-coalesced LSU is inferred. It has three modes depending on the access pattern:

- Cached: when the access pattern seems repetitive, the compiler implements a cache in BRAM (often a 256 kbit or 512 kbit cache if a size cannot be statically determined).
- Write-Ack: when data dependencies exist and a write-acknowledgement signal is required.
- Non-aligned: when memory accesses are not aligned with the external memory word size. This is inferred if accesses cannot be determined to be aligned at compile-time. Many unaligned requests result in poor performance. Additional logic resources are required to implement nonaligned LSUs.

Prefetching LSU

Prefetching LSUs are FIFOs that burst reads from memory to keep the buffer full of valid data based on the previously accessed address, assuming that the next accesses are contiguous. Non-contiguous accesses are allowed, but they cause lower performance due to the penalty incurred from flushing the FIFO.

Streaming and Semi-Streaming LSU

Streaming LSUs also instantiate a FIFO burst read from memory, but are generated in more restrictive conditions in that reads must be in-order and addresses can be simply calculated as an offset to the base address. A semi-streaming LSU instantiates a read-only cache on top of the streaming LSU.

Pipelined LSUs

Used for local memory (BRAMs), pipelined LSUs send requests immediately after they are issued in a pipelined fashion. If no arbitration is created, a never-stall pipelined LSU is created. If there are an insufficient amount of read/write ports for the number of concurrent accesses being requested, then the compiler creates an arbitration system that causes resource overhead and degradation in performance.

Coalesced Accesses

If the access pattern is determined to be consecutive at compile-time, the compiler combines reads/writes into wider accesses which we refer to as *coalesced accesses*. Coalesced accesses result in more efficient use of global memory bandwidth compared to multiple smaller accesses contending with each other for bandwidth [69]. Since an excessive amount of LSUs exhausts board resources, it is ideal if the design

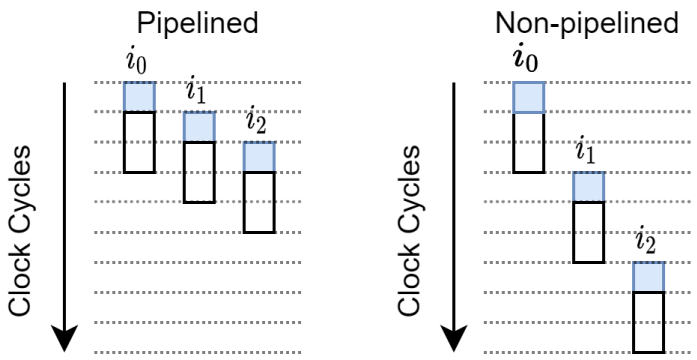

Figure 2.5: Launch frequency of a pipelined and non-pipelined loop (after [35])

has few wide read/writes to global memory to maximize throughput and cache read/writes locally in either BRAMs or registers.

2.4.4 Programming Model

As mentioned in Section 2.3.4, there are two types of kernels that OpenCL supports: data parallel and task parallel. Intel refers to these kernels as NDRange and single work-item (SWI) kernels respectively. While NDRange kernels are common for GPU programming (with work-item and work-groups being analogous to CUDA threads and thread blocks), Intel recommends that OpenCL kernels for FPGAs be written as SWI kernels [35]. An example of a SWI accumulation kernel is provided in Listing 2.2 and an equivalent NDRange kernel in Listing 2.3.

```
kernel void accum_swg (global int* a,
                       global int* c,
                       int size,
                       int k_size) {
    int sum[1024];
    for (int k = 0; k < k_size; ++k) {
       for (int i = 0; i < size; ++i) {
           int j = k * size + i;
           sum[k] += a[j];
       }
    }
    for (int k = 0; k < k_size; ++k) {
       c[k] = sum[k];
    }
 }
```

Listing 2.2: SWI Accumulation [35]

```
kernel void accum_ndr (global int* a,
                       global int* c,
                       int size) {
   int k = get_global_id(0);

   int sum[1024];
   for (int i = 0; i < size; ++i) {
     int j = k * size + i;
     sum[k] += a[j];
   }

   c[k] = sum[k];
}
```

Listing 2.3: NDRange Accumulation [35]

SWI kernels are written sequentially unlike NDRange kernels and can achieve parallelism through loop pipelining, which is an optimization that allows multiple iterations in a loop to be executed concurrently,

and vectorization through either loop unrolling or explicit OpenCL vectors. An ideal loop is one where every iteration is launched per clock cycle (i.e., an initiation interval, or II, of 1) which yields in maximum pipeline efficiency. Figure 2.5 shows the difference in execution between a non-pipelined (serial) loop and a pipelined loop with an II=1. By launching multiple iterations in-flight, a pipelined loop can complete execution in fewer cycles compared to a non-pipelined loop.

2.4.5 Design Considerations

Developing OpenCL kernels for FPGAs is different from that for conventional processors or GPUs due to a number of factors. The first and most critical factor is that the FPGA is a spatial compute device, whereas with the CPU and GPU, there is a program counter and a stream of instructions that go to the processing cores. Kernel logic must be implemented via logic elements and DSP blocks for floating-point operations. Programs with excessive control flows are inefficient when mapped to an FPGA and a designer should strive to reduce them. In addition, as loops incur area overhead due to the hardware created for control and boundary checking, deeply-nested loops should also be avoided.

Secondly, FPGAs lack a hierarchical memory system present in CPUs and GPUs. The compiler can provide some relief by interleaving buffers on multiple banks in external memory if it is available on the board and supported by the BSP[4]. Generally, global memory bandwidth must be conserved to improve kernel performance through the re-use of data on explicitly managed local on-chip caches. However, excessive concurrent accesses, particularly writes, to local memories results in high fan-in for write ports and fan-out for read ports, reduced operating frequency (f_{max}) and ultimately, routing congestion [69]. To accommodate concurrent writes without port sharing, the compiler replicates a buffer in multiple block RAMs. If it is available on the board, the compiler also has the option to configure the memory for double-pump mode which increases the number of ports by 2× at the cost of reducing the circuit f_{max} by half.

Thirdly, since the kernel must be mapped onto a finite number of on-chip resources, one must consider the resource constraints of the device since designs that do not fit on the device will not synthesize.

For the above reasons, OpenCL kernels that may have been optimized for other devices may suffer from greatly reduced performance or may not synthesize at all.

[4]The BSP for Stratix 10 MX with HBM2, for example, does not support implicit banking on the HBM and designers must explicitly split data across different HBM channels and allocate separate CL buffers.

2.5 TVM: A Deep Learning Optimizing Compiler

ML frameworks are commonly used to build, train, and deploy ML models. For example, Tensorflow [2] is a popular, open-source, high-performance ML framework based on dataflow programming, where computations in a model are described using a dataflow graph. For users that desire a higher level of abstraction, Keras is a high-level interface for Tensorflow. Other ML frameworks exist, such as MXNet [10], Caffe [38], and PyTorch [48].

However, maintaining support for numerous hardware backends across different machine learning frameworks requires a significant amount of upkeep and engineering effort. Further, deep learning frameworks are limited in the level of optimizations that can be applied since they are often unaware of the hardware and must rely on vendor-provided, platform-specific operator libraries (e.g., cuDNN) [11]. Deep learning compilers provide a level of abstraction that allow flexibility in the deployment of deep learning models on a wide variety of devices.

The Apache Tensor Virtual Machine (TVM) [14] is an open-source compiler framework that allows users to compile and optimize deep learning models from high-level machine learning frameworks and deploy them on a variety of target lower level languages such as LLVM, C, CUDA, OpenCL, Vulkan, and more. Originally a research project at the University of Washington that began in 2017 [11], it became an Apache top-level project in November 2020 [20]. The sections below describe the major components of TVM and its capabilities.

2.5.1 Capabilities and Compilation Flow

TVM can import a model from various deep learning frameworks into the top-level functional intermediate representation called Relay IR [53]. With the Relay representation, TVM can apply rules-based transformations that resemble traditional compiler passes such as operator fusion, dead code elimination, layout changes, among others that are not specific to the target hardware.

Following the transformations, the Relay operators are lowered to tensor expressions which is a domain-specific language for kernel construction. TVM checks if the operator is in its operator registry that contains common operators such as 2D convolution and fully-connected layers. These operators are called *compute functions*.

To implement compute functions, TVM determines which code optimizations can be applied to improve performance such as loop unrolling, fusion, tiling, and vectorization. These transformations are referred to as *schedule primitives*, and the collection of these primitives form a *schedule*.

For commonly used operators and platforms, TVM provides a library of compute functions and

schedules that is referred to as the TVM Operator Inventory (TOPI). Within these schedules there often exists a template for a guided search into finding the optimal parameters for scheduling such as tiling sizes which may vary, depending on the cache size and other architectural characteristics. This infrastructure for template-guided search is called AutoTVM. In more recent versions, TVM also provides a template-free schedule search space explorer called Auto-Scheduler or AutoTVMv2 (also published as Ansor [68]), which given a compute function and a device target as input, generates a schedule using evolutionary search and learned cost models.

Finally, the compute function and schedule are lowered to code that can be executed on the target and compiled to a platform-specific binary. For example, if the user desired to use OpenCL on a GPU, a code generator for OpenCL C is used to generate a kernel file, and the CPU host code is generated and compiled from LLVM IR using an LLVM code generator[5]. This host code calls a library of OpenCL-specific functions in the TVM runtime backend that call the OpenCL API (enqueue kernel, buffer copy, etc.) to execute model computation. TVM's compilation flow is visualized in Figure 2.6.

[5]LLVM is not used to generate or optimize the kernel code but is used to generate the host code. TVM uses LLVM IR to create the host program so that code generation can be generalized to multiple backends including CUDA and OpenCL.

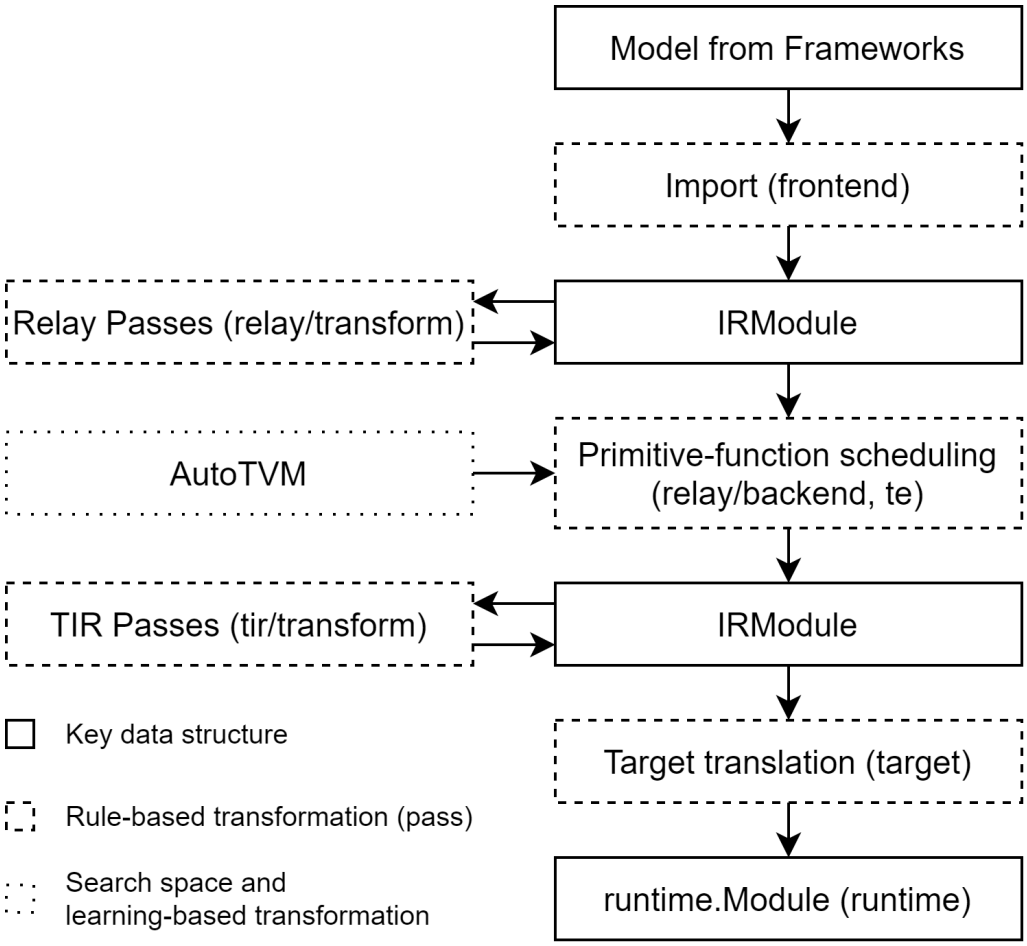

Figure 2.6: TVM compilation flow (after [50])

Chapter 3

End-to-End Network Deployment

This chapter describes our approach of using a chain of compiler tools to deploy deep neural networks starting from a model in a high-level machine learning framework to an FPGA. We describe the advantages and disadvantages of our approach in the context of existing approaches.

3.1 Compilation Flow

We propose a novel template-free compilation flow [13] that generates OpenCL kernels to implement CNN operations from high-level compute descriptions using a deep learning compiler. Specifically, we use TVM [11], an open-source deep learning compiler, and target the Intel OpenCL compiler for FPGA to compile and synthesize generated OpenCL kernels for CNN operations. Figure 3.1 illustrates the components of our proposed compilation flow.

The flow begins with a graph representation of a neural network that is trained in any of the high-level ML frameworks that TVM supports, such as PyTorch, Tensorflow, Caffe, or Keras. This representation is the input to the TVM compiler frontend (Frontend in Figure 3.1), which converts the graph into a set of Relay IR (see Section 2.5) functions. At this step, TVM executes graph optimization passes. This includes operator fusion, where injective (one-to-one map) functions such as addition are fused with other injective functions [11]. Similarly, complex functions such as 2D convolutions have element-wise operations fused with their output. This results in operations such as residual connections ("skip" connections that connect the output of a preceding layer) and batch normalizations being fused to the output of convolutions. The result is a distinct kernel generated for each convolution, dense, padding, and softmax layer.

From Relay, TVM constructs a low-level program representation (tensor IR) with tensor expressions

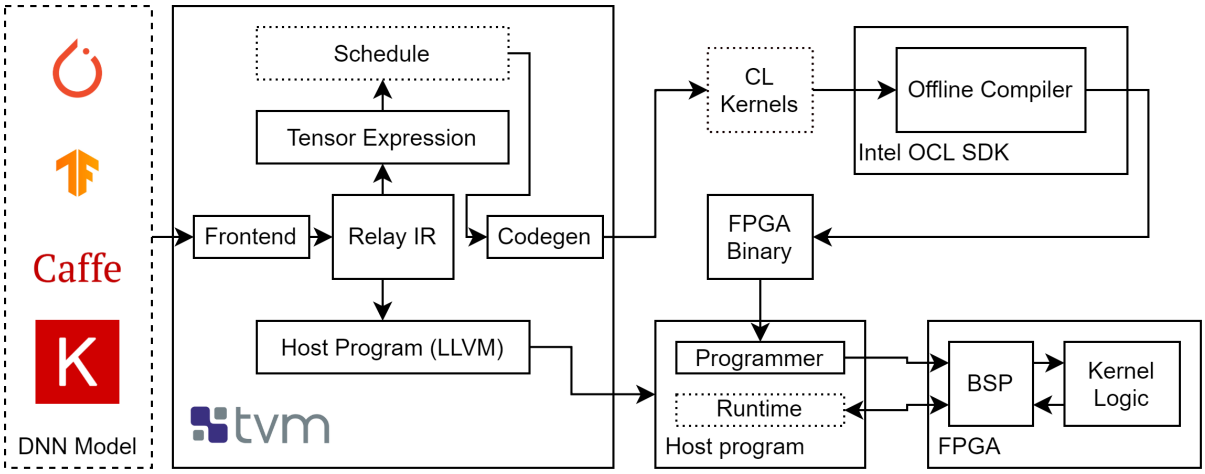

Figure 3.1: Diagram of the compilation flow from a CNN model definition in an ML framework to execution on FPGA. Modifications are made to dotted components.

(Tensor Expression) that are scheduled from a library of operators called the TVM Operator Inventory (referred to as TOPI in the TVM documentation, Schedule in Figure 3.1). The tensor IR stage contains lowering passes for target-specific vectorization and data types. Then, the OpenCL C code generator (Codegen) is used to generate kernel code (CL Kernels) from tensor IR. The host program is generated in LLVM IR and built with LLVM.

Following code generation, the kernel code is synthesized by Intel's OpenCL compiler (Offline Compiler) for FPGA, AOC. We make no modifications to AOC and therefore, consider it a black-box component in the process. AOC performs static analysis and synthesizes the design with Quartus, generating an FPGA binary. The host program reconfigures the FPGA with this binary. Then, the host program copies the parameters and inputs in sequence until the output is read back to host memory.

In our flow, we modify several components of TVM. Components encased in a dotted black line in Figure 3.1 are existing components in TVM that we modify to optimize the resulting accelerator. This includes the schedule, generated kernels, and the host runtime program.

Our flow supports two modes of execution: *pipelined* and *folded* execution. In pipelined execution, output feature maps are transferred between kernels using OpenCL channels[1], a kernel is generated per layer, and all kernels are active concurrently. To pipeline inference, all activations of a layer must be stored in a kernel's local memory. Given the scarcity of on-chip storage on FPGAs, this limits deployment to relatively small networks.

However, CNNs typically have a "workhorse" operation that is repeated. For example, 1×1 con-

[1]Channels are an extension to the OpenCL specification implemented by Intel to allow kernels to directly communicate using FIFOs [36]. Although it is not part of the original specification, we refer to them as OpenCL channels for the remainder of this document.

volutions constitute 94.9% of multiply-adds in MobileNetV1 [32]. This makes it possible to re-use parameterized (dynamic input/output tensor shapes) kernels for multiple layers in the network, consuming less FPGA resources compared to a fully-pipelined design. We refer to this as folded execution, and use this to deploy larger networks such as MobileNet and ResNet. Although it is theoretically possible to pipeline a network that is also folded, it is unlikely that there would be sufficient on-chip memory to implement both.

The result is an HLS-generated dataflow accelerator. This flow has several benefits over existing approaches:

- The approach enables the automatic generation of CNN kernels, isolating the user from hardware design and reducing development time while being flexible to implement different operations. We demonstrate this by evaluating our approach on three different CNNs with varied input sizes, and operations.
- It builds upon industry-standard tools; the use of Intel's OpenCL SDK for FPGAs promotes portability across Intel FPGA platforms, including upcoming ones. We demonstrate this by evaluating our approach across three different FPGA platforms.
- Using an open-source ML compiler provides advantages over supporting a singular framework. First, it takes advantage of TVM's frontend that supports multiple frameworks and CNN representations. Second, adding support for novel algorithms and/or layers requires the optimization of its schedule since kernel code is generated by the TVM codegen. In contrast, with flows such as DnnWeaver [55], fpgaConvNet [63], and MALOC [22], deploying other types of networks may require some overhaul in the accelerator architecture.

However, there are challenges and limitations to this approach:

- Representing layers as OpenCL kernels limits the kernels to synchronize at the granularity of the entirety of the input/output feature map. Further, like other dataflow accelerators, on-chip resource constraints must be considered when mapping network execution to kernels.
- Circuits generated by TVM/AOC will likely not be competitive with designs generated from RTL/HLS templates made by hand in terms of clock frequency and hardware utilization.
- Designs generated from naive schedules perform poorly, underutilize hardware, and inefficiently use global memory, requiring the optimizations that we perform. We propose and implement a set of optimizations to alleviate these limitations, as described in Section 3.2.

We believe the combination of an easy-to-use ML interface and the flexibility gained from generating kernels automatically from a tensor language may outweigh the downsides of the approach after some optimization. This work aims to provide an assessment of this hypothesis.

3.2 Performance Concerns

Without modification, the schedules for the high-level synthesis backend and the generated kernels do not produce designs with acceptable performance or fail to synthesize because the design exceeds available resources. Several issues prevent the FPGA from achieving better performance:

1. Fusing operations to convolutions gives rise to read-after-write dependencies, which prevents loop pipelining and forces serial execution (see Section 2.4.4). As a result, the kernels have degraded performance.

2. The generated kernels do not produce parallel hardware. Since parallelism must be extracted from unrolled loops, and loops are not implicitly unrolled, there is typically only one arithmetic operation performed per cycle at most, underutilizing FPGA resources.

3. With the underutilization of compute resources, external memory bandwidth is also underutilized. Further, the generated kernels use global memory for all data including accumulations, which can be moved into local memory to take advantage of higher on-chip memory bandwidth.

4. A one-to-one layer-to-kernel mapping can easily exhaust resources due to excessive logic usage for load-store units, preventing a design to synthesize for larger networks.

To address these issues, we propose a number of optimizations to the kernel in Chapter 4 and describe how we apply these transformations to the schedule within TVM to generate improved kernels in Chapter 5.

Chapter 4

Optimizing OpenCL Kernels

We propose the use of a number of optimizations to achieve a higher level of parallelism and increase the efficiency at which global memory is utilized. We group our optimizations into two categories:

- *Kernel Optimizations*: This category includes transformations made to an OpenCL kernel code and are intended to improve its performance or reduce its area usage when compiled by AOC. The transformations include loop unrolling, loop strip mining/tiling, loop fusion, loop-invariant code motion, cached writes, channelization, autorun, and parameterized kernels.
- *Host Optimizations and Best Practices*: There are modifications made to the OpenCL host runtime for optimal performance, as recommended by the Intel guide. They include concurrent execution, and optimized float operations.

In the remainder of this chapter, we describe the above transformations in more detail.

4.1 Loop Unrolling

Loop unrolling is a loop transformation that replicates a loop body by a specified factor, known as the unroll factor (UF), and reduces the loop trip count by the same factor. It reduces the overhead from executing a loop, such as branching and loop-closing tests, and is applicable to loops both with constant and compile-time unknown bounds [46]. In addition, loop unrolling can increase instruction parallelism and improve register, cache, and TLB locality [6].

Unrolling can be applied to any loop with some exceptions. In general, if it is not known at compile-time that loop iteration count is evenly divisible by the unrolling factor, a loop *epilogue* must be created, which is a separate loop that executes the remainder iterations not executed by the unrolled loop [6].

An example of the base loop and then an unrolled loop is provided in Listings 4.1 and 4.2 respectively. Note that in the given example, it is assumed that the loop count is always evenly divisible by the unroll factor.

```
kernel void vector_add(global float* a,
                       global float* b,
                       global float* c,
                       int size) {
    for (int i = 0; i < size; ++i) {
      c[i] = a[i] + b[i];
    }
 }
```

Listing 4.1: Vector addition

```
kernel void vector_add_ur(global float* a,
                       global float* b,
                       global float* c,
                       int size) {
    int exit = size / 4;
    for (int i = 0; i < exit; i += 4) {
      c[i] = a[i] + b[i];
      c[i+1] = a[i+1] + b[i+1];
      c[i+2] = a[i+2] + b[i+2];
      c[i+3] = a[i+3] + b[i+3];
    }
}
```

Listing 4.2: Vector addition, UF = 4

On FPGAs, unrolling has the effect of replicating hardware that performs the computations of the loop body. Loop unrolling increases concurrency by executing multiple load/store instructions and/or arithmetic operations per clock cycle; increasing parallelism and memory bandwidth utilization. In more specific terms, it can have the following effects:

- If the loop contains computations, AOC replicates the compute logic whether in the form of logic elements or DSPs, depending on the type of the input and output data. For example, one DSP is generated in Listing 4.1, whereas with UF = 4 there are four DSPs generated in Listing 4.2. The increase in parallelism improves performance, theoretically by a factor of 4× in this example.

- If the loop contains global memory loads/stores, then AOC widens the LSU access width by the unrolling factor if the memory access pattern is consecutive. Otherwise, it replicates the LSU. The latter incurs a significantly greater cost in logic and RAM if cached burst-coalesced LSUs are inferred [35]. Widening the bit width allows LSUs to request more data at once to scale with the increased parallelism from the replicated compute logic. For example, there are 3 32-bit (loading a single float of 4 bytes per clock cycle to load elements from arrays a, b, and c) LSUs generated

for Listing 4.1, and 3 128-bit LSUs generated for Listing 4.2 since it can be determined that the memory accesses are consecutive at compile-time.

- If the loop contains local memory loads/stores, the compiler either replicates or banks the BRAM to create more read/write ports to allow for concurrent read/writes for the LSU. The number of write accesses should be reduced for buffers in local memory to reduce the replication required and thus avoid memory arbitration that may stall accesses and negatively impact performance [69].

AOC provides two methods to implement loop unrolling [36]. One method is to use a `#pragma unroll N` directive placed right before the loop declaration. If $N = 1$, the loop is explicitly prevented from being unrolled. If N is not provided, the entire loop is unrolled completely, removing the loop control structure altogether. Any positive integer (within reasonable constraints) provided as N is used as an unroll factor to partially unroll the loop.

However, Intel manuals state a few exceptions [36]. First, a loop that has loop-carried dependencies with a large number of iterations may not synthesize if specified to be fully unrolled. Second, non-constant loop bounds cannot be unrolled fully because the bounds cannot be determined at compile-time. Finally, a loop that has complex control flows, such as exit conditions that cannot be statically determined at compile time is not unrolled. In these cases, AOC throws a warning and the loop is not unrolled.

The second method is to explicitly write unrolled loops as it is done in Listing 4.2. AOC is still able to coalesce the memory accesses to a, b, and c as long as it can be determined at compile-time that they are consecutive accesses, which is the case for this particular example.

4.2 Loop Strip Mining/Tiling

Loop strip mining, also referred to as loop sectioning, controls the granularity of a loop by partitioning a loop into smaller segments (or strips) [6]. It can be used to increase the number of independent computations in the inner loop nest for vectorization, or used for SIMD compilation.

Loop tiling, or loop blocking, can be regarded to as the multi-dimensional form of strip mining where multiple nested loops may be broken down to multiple smaller segments. It is primarily used to improve cache locality, and can also benefit processor, register, TLB, and/or page locality [6].

An example of vector-matrix multiply before any transformation, after strip mining, and after strip mining and unrolling is shown in Listings 4.3, 4.4, and 4.5 respectively. Note that in this example, it is also assumed that the loop bounds are also evenly divisible by the tiling factor.

```
// x is 64-element vector
// y is a matrix with dimensions [128][64]
// result is a 64-element vector
for (int i = 0; i < 128; ++i) {
  int sum = 0;
  for (int k = 0; k < 64; ++k) {
    sum += x[k] * Y[i][k];
  }
  c[i] = sum;
}
```

Listing 4.3: Base implementation of $c = Yx$

```
for (int i = 0; i < 128; ++i) {
  int sum = 0;
  for (int ko = 0; ko < 64/4; ++ko) {
    for (int ki = 0; ki < 4; ++ki) {
      sum += x[ko*4+ki] * Y[i][ko*4+ki];
    }
  }
  c[i] = sum;
}
```

Listing 4.4: k loop strip mined by a factor of 4

```
for (int i = 0; i < 128; ++i) {
  int sum = 0;
  for (int ko = 0; ko < 64/4; ++ko) {
   sum += x[ko*4+0] * Y[i][ko*4+0];
   sum += x[ko*4+1] * Y[i][ko*4+1];
   sum += x[ko*4+2] * Y[i][ko*4+2];
   sum += x[ko*4+3] * Y[i][ko*4+3];
  }
  c[i] = sum;
}
```

Listing 4.5: Inner loop ki fully unrolled

Neither strip mining nor tiling on their own are desirable transformations for FPGA kernel code as the extra loop(s) created after the transformation causes extra loop control to be generated, incurring more area and possibly degrading performance. If the loop count is not evenly divisible by the tiling factor, cleanup code (code to complete the remaining iterations) must be placed following the stripmined/tiled loops.

4.3 Loop Fusion

Loop fusion takes two adjacent loops and merges them into a single loop[1]. It is applicable to loops with no backward flowing dependences from statements at the second loop to statements at the first loop. In the case where the loop count is not equal for the two loops, the extra iterations can be peeled [6].

On FPGAs, this has the effect of having less loop control structures, resulting in reduced area [36]. It may also improve data locality since a variable needs to be only accessed once and can decrease storage requirements. As an example, we use a 1×1 convolution kernel generated by the default TVM schedule in Listing 4.6. By fusing the reduction loop and the loop that performs the activation function before writing back into global memory, as shown in Listing 4.7, the second loop is eliminated. Fusion also makes it unnecessary to use a scratchpad as a 2D-array. It can be made into a smaller local register, as is discussed in Section 4.5. This decreases global memory contention by removing global LSUs for accessing the scratchpad and frees more bandwidth for loading input feature maps and weights.

```
kernel void conv2d_1x1_base(global float *scratchpad, global float *in_fm, global
    float *w, global float *out_fm) {
  for (int ax1 = 0; ax1 < 128; ++ax1) { // output feature map depth
    // reduction loop
    for (int yy = 0; yy < 28; ++yy) { // output rows
      for (int xx = 0; xx < 28; ++xx) { // output columns
        scratchpad[yy][xx] = 0.0f;
        for (int rc = 0; rc < 64; ++rc) { // input feature map depth
          scratchpad[yy][xx] += in_fm[rc][yy][xx] * w[ax1][rc];
        }
      }
    }
    // writeback
    for (int ax2 = 0; yy < 28; ++ax2) { // output rows
      for (int ax3 = 0; xx < 28; ++ax3) { // output columns
        out_fm[ax1][ax2][ax3] = max(scratchpad[ax2][ax3], 0.0f);
      }
    }
  }
}
```

Listing 4.6: Conv2D 1×1, default schedule in TVM

[1]Loop fusion as a pragma (`#pragma loop_fuse`) and as an automatic optimization in AOC v20.1. In our experiments, however, Quartus versions required for the BSPs that we use precede this feature, and thus we use TVM scheduling primitives to combine adjacent computations.

```
kernel void conv2d_1x1_base(global float *scratchpad, global float *in_fm, global
    float *w, global float *out_fm) {
  for (int ax1 = 0; ax1 < 128; ++ax1) { // output feature map depth
    // fused loop
    for (int yy = 0; yy < 28; ++yy) { // output rows
      for (int xx = 0; xx < 28; ++xx) { // output columns
        scratchpad[yy][xx] = 0.0f;
        for (int rc = 0; rc < 64; ++rc) { // input feature map depth
          scratchpad[yy][xx] += in_fm[rc][yy][xx] * w[ax1][rc];
        }
        out_fm[ax1][yy][xx] = max(scratchpad[yy][xx], 0.0f);
      }
    }
  }
}
```

Listing 4.7: Fused loops

4.4 Loop-Invariant Code Motion

Loop-invariant computations refer to statements in a loop body that results in the same value regardless of the iteration of the loop, and thus may be moved out of the loop [46]. Doing so eliminates a redundancy and may reduce the number of clock cycles spent on loop-invariant computations.

For example, a kernel that generates values where each element is divided by the maximum of the array is shown in Listing 4.8. Computing the maximum value in the a array can be taken out of the loop because the value of `a_max` is not affected by the outer loop iteration variable i. Therefore, it can be moved outside and before the loop so that the maximum value is computed only once as shown in Listing 4.9, resulting the loop and its body being executed 128× less.

```
for (int i = 0; i < 128; ++i) {
  float a_max = -9.9e+99f;
  // computing maximum of array
  for (int j = 0; j < 128; ++j)
    a_max = max(a_max, a[j]);
  b[i] = a[i] / a_max;
}
```

Listing 4.8: Normalization, unoptimized

```
float a_max = -9.9e+99f;
// moved out of original loop
for (int j = 0; j < 128; ++j)
  a_max = max(a_max, a[j]);
for (int i = 0; i < 128; ++i)
  b[i] = a[i] / a_max;
```

Listing 4.9: Normalization, with computation of maximum moved out of main loop

4.5 Cached Writes

Writes to global memory generate a store unit that utilizes global memory bandwidth. However, due to the amount of logic it consumes, it can restrict the extent to which a loop can be tiled and unrolled. If possible, writes should be cached[2] in local memory (i.e., either registers or BRAM) to avoid the generation of expensive store units, especially in the case of accumulations where the summation needs to be loaded before the result is added with the previous sum.

We re-use the example in Section 4.3, shown again in Listing 4.10, and optimize the accumulation by changing the scratchpad buffer in global memory to a single local float that is implemented by a register in Listing 4.11. This also eliminates a global LSU for the scratchpad variable.

In TVM, this optimization can be implemented in two ways. The first method is to change the thread scope to local using `set_scope`. The second method is to add a stage to the computation, where the original computation is performed locally and the added stage copies the local tensor to a global tensor. Since TVM allocates memory in the outermost scope of the threading boundary, the innermost loop using the buffer must be declared parallel (e.g., loop with iterator `xx` in Listing 4.11).

```
kernel void conv2d_1x1_base(global float *scratchpad, global float *in_fm, global
    float *w, global float *out_fm) {
  for (int ax1 = 0; ax1 < 128; ++ax1) { // output feature map depth
    for (int yy = 0; yy < 28; ++yy) { // output rows
      for (int xx = 0; xx < 28; ++xx) { // output columns
        scratchpad[0] = 0.0f;
        for (int rc = 0; rc < 64; ++rc) { // input feature map depth
          scratchpad[0] += in_fm[rc][yy][xx] * w[ax1][rc];
        }
        out_fm[ax1][yy][xx] = max(scratchpad[0], 0.0f);
...
```

Listing 4.10: Base convolution with accumulations in global memory

```
kernel void conv2d_1x1_cached(global float *in_fm, global float *w, global float *
    out_fm) {
  for (int ax1 = 0; ax1 < 128; ++ax1) { // output feature map depth
    for (int yy = 0; yy < 28; ++yy) { // output rows
      for (int xx = 0; xx < 28; ++xx) { // output columns
        float sum = 0.0f;
        for (int rc = 0; rc < 64; ++rc) { // input feature map depth
          sum += in_fm[rc][yy][xx] * w[ax1][rc];
        }
        out_fm[ax1][yy][xx] = max(sum, 0.0f);
...
```

Listing 4.11: Local register for accumulation

[2]Although this is not technically a cache (user-transparent memory management) as much as it is a scratchpad, we refer to this as a cache because it is described as such by TVM's scheduling documentation.

4.6 Channelization

Kernels are separate compute units and do not implicitly share any memory system. Consequently, kernel-to-kernel communication is done via buffers in global memory or by using OpenCL channels [36]. An alternative to global memory is channels, which allow kernels to communicate directly with each other without host control by placing a direct datapath between them implemented by registers. An example of three arithmetic operations separated to their kernel is provided in Listing 4.12, and one that utilizes channels in Listing 4.13.

```
kernel void A(global float restrict *a, global float restrict *b) {
  for (int i = 0; i < 8; ++i)
    b[i] = a[i] + 1;
}
kernel void B(global float restrict *b, global float restrict *c) {
  for (int i = 0; i < 8; ++i)
    c[i] = b[i] * 0.35;
}
kernel void C(global float restrict *c, global float restrict *d) {
  for (int i = 0; i < 8; ++i)
    d[i] = c[i] / -1.1;
}
```

Listing 4.12: Three arithmetic operations in different kernels, connected by global memory

```
channel float c0;
channel float c1 __attribute__((depth(8)));
kernel void A(global float restrict *a) {
  for (int i = 0; i < 8; ++i)
    write_channel_intel(c0, a[i] + 1);
}
kernel void B() {
  for (int i = 0; i < 8; ++i)
    write_channel_intel(c1, read_channel_intel(c0) * 0.35);
}
kernel void C(global float restrict *d) {
  for (int i = 0; i < 8; ++i)
    d[i] = read_channel_intel(c1) / -1.1;
}
```

Listing 4.13: Three arithmetic oeprations in different kernels, connected by channels

In Listing 4.12, 6 LSUs are generated with 2 per kernel whereas in Listing 4.13, communication with the intermediate kernel `B()` is replaced with primitives `write_channel_intel(channel, value)` and `read_channel_intel(channel)`. Consequently, there are 2 LSUs generated for Listing 4.13, only 1 for reading array `a` in global memory by `A()` and writing array `d` to global memory by `C()`.

The reduction of 4 LSUs has a marked reduction in chip resource usage. Further, since the traffic from `A()` to `B()` and from `B()` to `C()` is moved from global memory to channels, there is less contention for global memory if all three kernels execute concurrently. The decrease in the number of LSUs contending

for global memory and access to on-chip bandwidth results in better performance.

However, if a kernel needs to re-use data that it is consuming from a channel, it needs to store channel reads into local memory. Then, as it is stored onto a register or BRAM, it can be re-accessed, since once data is read from a channel it is discarded.

Channel reads stall when attempting to read from an empty channel, while channel writes stall when writing to a channel that is full [35]. As a result, unequal consumer/producer rates cause stalls in the kernel that degrades performance. To help alleviate this problem, a designer can implement buffered channels with a depth attribute where a FIFO queue can be added with a user-specified depth, as shown with the 8-float buffered channel `c1` in Listing 4.13.

4.7 Autorun Kernels

Kernels that do not have arguments (i.e., no accesses to global memory) can be declared *autorun*. Autorun kernels execute independent of the host, and are equivalent to wrapping a kernel with a `while(1)`. This decreases the overhead of command queue control from software. Autorun may be beneficial if the kernel execution times are small, where overhead is a relatively substantial amount.

We re-use the example from the previous subsection and declare function `B()` autorun in Listing 4.14. For a kernel to become autorun, two statements must be added: `__attribute__((max_global_work_dim(0)))`, which prevents the generation of kernel ID dispatch logic and `__attribute__((autorun))`.

```
channel float c0, c1;
kernel void A(global float restrict *a) {
  for (int i = 0; i < 8; ++i)
    write_channel_intel(c0, a[i] + 1);
}
__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
kernel void B() {
  for (int i = 0; i < 8; ++i)
    write_channel_intel(c1, read_channel_intel(c0) * 0.35);
}
kernel void C(global float restrict *d) {
  for (int i = 0; i < 8; ++i)
    d[i] = read_channel_intel(c1) / -1.1;
}
```

Listing 4.14: Listing 4.13 with autorun

4.8 Concurrent Execution

A single command queue for queuing tasks to a device serializes kernel execution. To have multiple kernels running in parallel, multiple command queues are required (i.e., using separate command queues for every kernel) [36]. Concurrently executing kernels can increase hardware utilization because multiple kernels are active at the same time, at the cost of no additional hardware. However, if dependencies exist between kernels, they must be synchronized.

Kernels may be synchronized through the use of channels, where a kernel blocks on a channel read until the preceding kernel writes an expected amount of output to the incoming channel [35]. They may also be synchronized in software using CL events.

4.9 Parameterized Kernels

For our approach, there is a kernel created for every layer by TVM, as described in Chapter 3. Although this is not a problem when compiling with a GPU compiler where the consequence may be a larger binary, this often results in a non-fitting design due to the resources consumed by the generation of the LSUs, especially if the kernel has been tiled and unrolled.

Thus, we group and parameterize similar kernels so that they may be reused across the network. Specifically, we group operations by the filter size and stride of convolutions. The number of filters, input channels, and the input feature map spatial dimensions are the parameters (kernel arguments) and can be set at runtime to execute different layers of the network.

For example, Figure 4.1 shows a set of kernels that are not grouped on the top half, with separate compute units for a 3×3 convolution with 64 filters, 128 filters, and 256 filters. The bottom half shows the set of kernels after parameterization, where one 3×3 convolution kernel is reused for the three convolutional layers in the top half of the figure.

This is implemented using symbolic shape execution in TVM, and is discussed in more details in Section 5.3.

4.10 Optimized Float Operations

AOC provides two optimizations for floating point operations: implementing floating-point operations in balanced trees and reducing rounding operations. The first two optimizations are provided as a flag to the compiler and are used in all of our experiments. We summarize each optimization below.

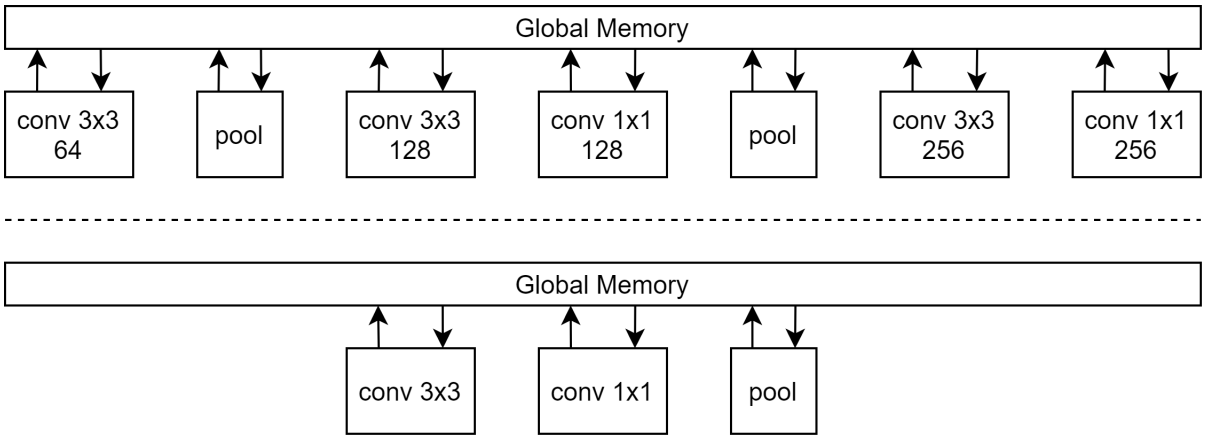

Figure 4.1: The grouping of kernels based on kernel size

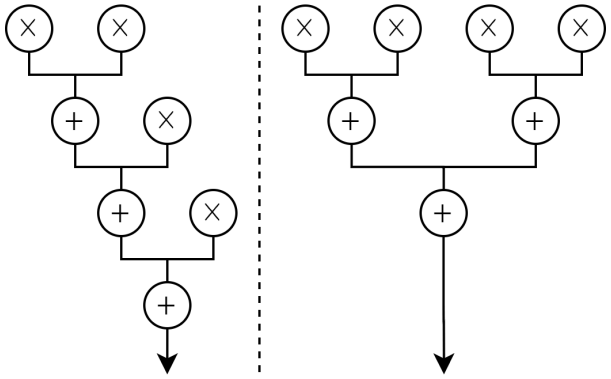

Figure 4.2: An illustration of multiply-adds seen in convolutions and fully-connected layers, where the left side shows the operation without tree balancing, and the right shows the operation with balanced trees

Tree Balancing

Floating point operations are not associative [1], and OpenCL enforces a strict order of operations. Specifically, multiply-accumulations in convolutions and fully-connected layers are implemented in a fixed sequence of additions following multiplications (illustrated on the left side of Figure 4.2). AOC provides a compiler flag `-fp-relaxed` that allows a relaxation of the order of operations. This allows multiply-adds to be implemented in a balanced tree (illustrated on the right side of Figure 4.2). We use this flag to enable balanced tree implementations, which reduces the number of cycles needed for computations.[3]

[3]Tree balancing may result in differences in floating-point results that are not compliant with IEEE standard 754-2008 [35]. Since the target applications in this study can tolerate these differences, this flag is enabled for our experiments.

Optimization	Pipelined	Parameterized	Parameter	Applied to
Loop unrolling	✓	✓	Factor	All kernels except transpose/padding
Loop fusion	✓	✓		Activation/batchnorm in Conv, FC, pooling
Loop ICM	✓	✓		Softmax
Cached Writes	✓	✓		All kernels except transpose/padding
Channelization	✓		Channel depth	Movement of activations, all layers
Autorun Kernels	✓			Pooling, transpose/padding
Concurrent Execution	✓			Host optimization
Parameterized Kernels		✓		Convs with same stride and filter size
Loop tiling		✓	Factor	Conv, FC
Optimized Float Operations	✓	✓		`-fpc -fp-relaxed` applied for all bitstreams

Table 4.1: Summary of optimizations and their applicability

Reducing Rounding Operations

The balanced tree optimization includes rounding operations that require resources to implement, and is not automatically reduced due to standard compliance. This can be reduced with the `-fpc` flag to remove intermediate rounding operations and conversions, and only perform them once at the end of the tree. This optimization may result in fused floating-point operations, such as fused multiply-accumulate (FMAC) [35]. This has the effect of potentially reduced area usage, and is enabled for our experiments.

4.11 Optimization Application

We list the optimizations discussed in this chapter and when they are applied in Table 4.1. Loop unrolling, loop fusion, invariant code motion, and cached writes are applied based on patterns that we see in the operation. For example, we fuse the loops for activations and batch normalizations to the convolution loops when the opportunity is present. Similarly, we see an opportunity to optimize invariants in the softmax schedule (see Section 5.1.3). These four optimizations are applicable to both kernels that are pipelined and parameterized, and are checkmarked for both in the table. However, the application of the other optimizations depends on the mode of execution.

Channels are used to implement kernel-to-kernel transfer of output feature maps in pipelined kernels. In pipelined kernels, layers without weights or biases have no accesses to global memory, and therefore can be declared autorun. Further, separate command queues can be created for each kernel in pipelined kernels to implement concurrent execution.

For parameterized kernels, these optimizations are not applicable. Since kernels can be used more than once, there is no longer a single downstream path for output feature maps and must be stored in global memory. Thus, channels are not used, and because kernels must read feature maps from global memory, they cannot be declared autorun. Further, since channels were used to synchronize

kernels executing concurrently, concurrent execution is also not enabled for parameterized kernels either. Instead, we opt to tile and unroll in multiple dimensions instead (see Section 5.1).

Lastly, we take advantage of both optimizations for float operations exposed by the compiler for all bitstreams.

The application of the optimizations require the specification of three factors: unrolling factors, tiling factors for parameterized kernels, and channel depth for pipelined kernels. Unrolling factors and tiling factors must be specified when applying unrolling/strip mining/tiling. Since we unroll all inner loops resulting from strip mining/tiling for vectorization, the unrolling/tiling factors are all the same in this context. We set three requirements for choosing a factor:

1. For loops that access global memory (and the data does not fit to a cache), the unrolling factor should not exceed theoretical peak external memory bandwidth of the FPGA. For example, the Arria 10 has a theoretical bandwidth of 34.1 GB/s. Assuming a 250 MHz operating frequency, this can support 136.4 bytes/cycle, which is approximately 32 floats. Therefore, the factor should not exceed 32 for the Arria 10.[4]

2. The loop count must be evenly divisible by the factor. If this requirement is not met, a prologue or epilogue must be created to compute the remainder. This is undesirable due to the additional control overhead.

3. The design must not exceed board resources.

While DSP usage can be predicted by counting the number of multiply-accumulate operations and multiplying by the unroll/tile factors, it is not straight forward to predict logic and BRAM usage based on the OpenCL code. It is not feasible to compile and sweep different tiling configurations as it is done on CPUs and GPUs because we require place and route. This can range from 5 to 12 hours, depending on the size of the FPGA and the complexity of the design. Placement is needed because AOC's initial area estimation reports often grossly overestimates logic usage compared to Quartus' fitter, and routing is also needed because in some cases, routing resources are exhausted due to congestion.

Therefore, a model-based prediction is the best option for estimating resource usage. However, predicting BRAM usage is difficult because compiler behavior with data interleaving, banking, and port allocation is unknown [69]. Further, how a compiler selects and configures an LSU is also unknown and not straight-forward to model. This is troublesome since our designs are often bound by LSU BRAM or logic. For these reasons, we manually select parameters that adhere to the requirements set above. A

[4]Observed by others in previous work as the bandwidth roof, part of the roofline model [67].

design space explorer would benefit the performance of work by maximizing overall network performance and resource utilization rather than the performance of individual layers. We leave resource modeling and exploration for a DSE to future work.

A buffer depth must also be specified for buffered channels. In our LeNet experiment, the channel depths are large enough to fit the output feature map of the producer kernel (i.e., if the output of a convolution is 256 floats, then the channel FIFO can hold up to 1024 bytes). This is adequate to prevent channels from stalling.

Chapter 5

Implementation

This chapter describes the implementations related to the optimizations we propose to use. The default and optimized schedules for convolution, dense, and softmax layers are described in Section 5.1. A custom OpenCL host program that we develop is described in Section 5.2. The implementation of parameterized kernels is described in Section 5.3. Lastly, implementation of the automatic application of some optimizations left to future work is discussed in Section 5.4.

5.1 Schedule Optimization

5.1.1 Convolution Layers

2D Convolutions (see Section 2.1.2) take an input feature map (FM) I and a set of weights W with the number of filters K, filter size F, and stride S as hyperparameters. We use TVM's generic NCHW (channel[1]-first) convolution schedule for the HLS backend as the starting point, as shown in Listing 5.1. The convolution operation has six nested loops: `ax1` is an index of the output FM channel, `yy` is an index to the output FM rows, `xx` is an index to the output FM columns, `rc` is an index to the input FM channel, `ry` and `rx` is an index of filter rows and columns. There is a scratchpad tensor that holds the dot product accumulations for a single output FM channel with dimensions $H_2 \times W_2$ where an activation function or batch normalization is typically applied following the convolution. In our example, we show a 2D S-strided $F \times F$ convolution simplified kernel with multi-dimensional array indexing, omit the kernel arguments, and place a dummy activation function.

[1]Not to be confused with OpenCL channels. In this chapter, channels refer to feature maps, or the data for a color of an input image (e.g., a grayscale image is considered single-channeled).

```
for (int ax1=0; ax1 < C_2; ax1++)
  for (int yy=0; yy < H_2; yy++)
    for (int xx=0; xx < W_2; xx++)
      scratchpad[yy][xx] = 0.0f;
      for (int rc=0; rc < C_1; rc++)
        for (int ry=0; ry < F; ry++)
          for (int rx=0; ry < F; rx++)
             scratchpad[yy][xx] += I[rc][S*yy+ry][S*xx+rx] * W[ax1][rc][ry][rx];
  for (int ax2=0; ax2 < H_2; ax2++)
    for (int ax3=0; ax3 < W_2; ax3++)
      y[ax1][ax2][ax3] = activation(scratchpad[ax2][ax3]);
```

Listing 5.1: Default schedule for convolution

This schedule suffers from the following issues:

- There exists a data dependency across iterations of `ax1` on the scratchpad variable due to the activation function being in a separate loop from the convolution computation. This forces the loop to execute serially, preventing loop pipelining.
- Due to that data dependency, the single-cycle accumulator cannot be inferred and the initiation interval of the inner most loop (iterations of `rx`) is 5 cycles.
- Absent of any unrolling, only one DSP is used to implement the operation, which limits performance even when the above issues are resolved.

To remove the data dependency, we fuse the activation function by changing the axis of the activation computation and write to `y` in the loop body with the `xx` iterator. We always fully unroll the inner loops `ry` and `rx` for increased parallelism. In addition, we create read caches for I and W although not shown in the listings below for brevity. A write cache is also created, changing the scope of the scratchpad variable (now `tmp`) to a local variable. These transformations result in Listing 5.2.

```
for (int ax1=0; ax1 < C_2; ax1++)
  for (int yy=0; yy < H_2; yy++)
    for (int xx=0; xx < W_2; xx++)
      float tmp = 0.0f;
      for (int rc=0; rc < C_1; rc++)
        #pragma unroll
        for (int ry=0; ry < F; ry++)
          #pragma unroll
          for (int rx=0; ry < F; rx++)
             tmp += I[rc][S*yy+ry][S*xx+rx] * W[ax1][rc][ry][rx];
      y[ax1][yy][xx] = activation(tmp);
```

Listing 5.2: Fused schedule for convolution with $F \times F$ reduction unrolled

AOC is now able to schedule the inner most loop (i.e., `rc`) with an II=1. Thus, with no data dependencies, all loops are pipelined[2].

[2]Not to be confused with kernel pipelining. See Section 2.4.4 for a description of loop pipelining.

Unrolling the two innermost reduction loops reduces the total trip count by a factor of $F \times F$, but this can be improved further by tiling along the input FM channel `rc` and output FM columns `xx` and unrolling the inner loops `rci` and `xxi`, as shown in Listing 5.3. The inner loop `xxi` has also been moved into the reduction so that it can be unrolled without affecting the `rco` outer loop. The caveat is that now there are three copies of the loop for initialization (shown in the listing as a list initialization), reduction, and the write to y. All three loops are unrolled.

```
for (int ax1=0; ax1 < C_2; ax1++)
  for (int yy=0; yy < H_2; yy++)
    for (int xxo=0; xxo < W_2; xxo += W_2vec)
      float tmp[W_2vec] = {0.0f}; // list initialization
      for (int rco=0; rco < C_1; rco += C_1vec)
        #pragma unroll
        for (int rci=0; rci < C_1vec; rci++)
          #pragma unroll
          for (int xxi=0; xxi < W_2vec; xxi++)
            #pragma unroll
            for (int ry=0; ry < F; ry++)
              #pragma unroll
              for (int rx=0; ry < F; rx++)
                tmp[xxi] += I[rco+rci][S*yy+ry][S*(xxo+xxi)+rx] * W[ax1][rco+rci][ry][
    rx];
      #pragma unroll
      for (int xxi=0; xxi < W_2vec; xxi++)
        y[ax1][yy][xxo+xxi] = activation(tmp[xxi]);
```

Listing 5.3: Convolution schedule tiled in two dimensions, `xx` and `rc`

By tiling and unrolling in two additional dimensions, `xx` and `rc`, the total trip count is reduced by a factor of $C_{1vec} \times W_{2vec} \times F \times F$. The number of DSP blocks allocated for computation is also increased by the same factor. Since the weight reads are contiguous in the dimensions that we unroll, the reads are coalesced into an access width that is $32 \times C_{1vec} \times F \times F\times$ bits wide. However, this is not the case for reading I, and therefore there are $C_{1vec} \times F$ LSUs for I with $32 \times W_{2vec} \times F$ bit reads. The LSU bit width for writing to the output tensor y is $32 \times W_{2vec}$ bits. The temporary variable that stores the accumulation also increases in size by the same amount.

It is better to have few LSUs that are wide than to have many LSUs that have short accesses. Therefore, it is typically better in our schedule to increase W_{2vec} over C_{1vec}. However, the dimensions of input feature maps are typically small (ranging from as small as 7×7 to 224×224), and so unrolling in this direction comes with diminishing returns for layers where the input feature map shapes are smaller than W_{2vec}. The number of input channels—with the exception of the input layer with one or three channels—is typically much larger, in a typical range of 64 to 1024 filters.

We classify 1×1 convolutions as a special case. Since $F = 1$, the two innermost reduction axes `ry`

and `rx` have a trip count of 1, and the trip count reduction factor is now $C_{1vec} \times W_{2vec}$. To increase the reduction, we tile and unroll along the `ax1` axis for a total trip count reduction of $C_{2vec} \times C_{1vec} \times W_{2vec}$. We show this schedule in Listing 5.4.

```
for (int ax1o=0; ax1o < C_2; ax1o += C_2vec)
  for (int yy=0; yy < H_2; yy++)
    for (int xxo=0; xxo < W_2; xxo += W_2vec)
      float tmp[C_2vec][W_2vec] = {0.0f};
      for (int rco=0; rco < C_1; rco += C_1vec)
        #pragma unroll
        for (int xxi=0; xxi < W_2vec; xxi++)
           #pragma unroll
          for (int ax1i=0; ax1i < C_2vec; ax1i++)
            #pragma unroll
            for (int rci=0; rci < C_1vec; rci++)
              tmp[ax1i][xxi] += I[rco+rci][S*yy][S*(xxo+xxi)] * W[ax1][rco+rci];
      for (int ax1i=0; ax1i < C_2vec; ax1i++)
        #pragma unroll
        for (int xxi=0; xxi < W_2vec; xxi++)
          y[ax1o+ax1i][yy][xxo+xxi] = activation(tmp[ax1i][xxi]);
```

Listing 5.4: Convolution schedule tiled in two dimensions, `xx` and `rc`

5.1.2 Dense Layers

The unbatched fully-connected layer is essentially a matrix-vector multiplication with an optional bias addition between the input vector I and weight matrix W. A sample of the kernel generated from the base schedule is shown in Listing 5.5. In this sample, I is an N-element vector, with the weights having the shape $(M \times N)$, resulting in an M-element output. A `bias` vector is added following the operation.

```
for (int j = 0; j < M; ++j) {
  dot[0] = 0.000000e+00f;
  for (int k = 0; k < N; ++k)
    dot[0] += I[k] * W[((j * N) + k)];
  y[j] = (dot[0] + bias[j]);
}
```

Listing 5.5: Base schedule for FC layers

To increase parallelism, we strip mine the inner loop with the iterator k by a factor that maximizes global memory utilization and unroll the inner loop, as shown in Listing 5.6. In this example, we use a strip mine factor of 4, increasing the memory access width of both I and W to 128-bit reads. The cache size for I is large enough for the vector to fit in BRAM for our evaluation networks. Reading weights on the other hand do not have data re-use and influences the kernel's global memory utilization.

In a similar fashion done with convolution, we also change the scope of the temporary variable containing the dot product accumulations to local to reduce global memory use.

```
for (int j = 0; j < M; ++j) {
  float dot = 0.000000e+00f;
  for (int k = 0; k < N/4; ++k) {
    dot += I[k] * W[((j * N) + k * 4)];
    dot += I[k+1] * W[((j * N) + k * 4 + 1)];
    dot += I[k+2] * W[((j * N) + k * 4 + 2)];
    dot += I[k+3] * W[((j * N) + k * 4 + 3)];
  }
  y[j] = (dot + bias[j]);
}
```

Listing 5.6: Base schedule for FC layers

5.1.3 Softmax Layers

Relative to other layers, the softmax activation layer (see Section 2.1.2) is fairly lightweight especially when the number of classes is small as it is the case with LeNet (10 classes). It does, however, require exponential functions and floating-point division for implementation. The base schedule of softmax (see Section 2.1.2), shown in Listing 5.7, re-computes the maximum element and the denominator (sum of exponents) for every iteration of `i1`, despite the values being invariant to the loop. This results in wasted cycles and performance can be easily improved. The schedule is modified so that loop-invariant values are computed once in a separate axis, shown in Listing 5.8.

```
for (int i1 = 0; i1 < 1000; ++i1) {
  T_softmax_maxelem[(0)] = -3.402823e+38f;
  for (int k = 0; k < 1000; ++k) {
    T_softmax_maxelem[(0)] = max(T_softmax_maxelem[(0)], placeholder[(k)]);
  }
  for (int i11 = 0; i11 < 1000; ++i11) {
    T_softmax_exp[(i11)] = exp((placeholder[(i11)] - T_softmax_maxelem[(0)]));
  }
  T_softmax_expsum[(0)] = 0.000000e+00f;
  for (int k1 = 0; k1 < 1000; ++k1) {
    T_softmax_expsum[(0)] = (T_softmax_expsum[(0)] + T_softmax_exp[(k1)]);
  }
  T_softmax_norm[(i1)] = (T_softmax_exp[(i1)] / T_softmax_expsum[(0)]);
}
```

Listing 5.7: Base schedule for softmax

```
T_softmax_maxelem[(0)] = -3.402823e+38f;
for (int k = 0; k < 1000; ++k) {
  T_softmax_maxelem[(0)] = max(T_softmax_maxelem[(0)], placeholder[(k)]);
}
for (int i1 = 0; i1 < 1000; ++i1) {
  T_softmax_exp[(i1)] = exp((placeholder[(i1)] - T_softmax_maxelem[(0)]));
}
T_softmax_expsum[(0)] = 0.000000e+00f;
for (int k1 = 0; k1 < 1000; ++k1) {
  T_softmax_expsum[(0)] = (T_softmax_expsum[(0)] + T_softmax_exp[(k1)]);
}
for (int i11 = 0; i11 < 1000; ++i11) {
  T_softmax_norm[(i11)] = (T_softmax_exp[(i11)] / T_softmax_expsum[(0)]);
}
```

Listing 5.8: Optimized schedule for softmax

5.2 Custom Host Code

We implement an OpenCL C/C++ host program external to TVM to demonstrate the use of certain optimizations that are unsupported in the TVM host generation code in its current state. It is used to demonstrate channels, autorun kernels, concurrent execution, and parameterized kernels. These optimizations are done by hand. We implement the following capabilities in this host code:

- Loading parameters and kernel buffer sizes exported from TVM
- Toggleable OpenCL event profiling and Intel OpenCL SDK profiling by macros
- Executing kernels one or more times with different sets of buffers and parameters, permitting the reuse of a kernel for multiple layers
- Toggleable concurrent execution by creating a separate command queue per kernel
- Asynchronous OpenCL task enqueuing (non-blocking calls) for increased concurrency
- Output verification and debugging capabilities (per-layer activation dump)

The combination of concurrent execution, asynchronous OpenCL task enqueuing, and channelized kernels implement a layer-pipelined forward pass. Asynchronous OpenCL task enqueuing and concurrent execution is disabled when the OpenCL or Intel OpenCL event profiler is enabled as it is required for an event to be completed before profiling information is retrieved from it.

5.3 Symbolic Shape Execution

In typical deployments with TVM, a graph representing a CNN contains nodes with known shapes at compile-time and when kernels are generated, loop boundaries and array access expressions are constant. When the input shapes are not known, TVM can generate kernels with variable shapes in the computation by replacing the constant with a symbolic placeholder (referred to as a `te.var` object). We use this feature to group the execution of convolutions with the same stride and filter size to the same kernel. We show the Relay IR generated for a kernel with constant shapes in Listing 5.9 and the same kernel with symbolic shapes in Listing 5.10.

```
primfn(in_1: handle, w_1: handle) -> ()
  attr = {"global_symbol": "main", "tir.noalias": True}
  buffers = {w: Buffer(w_2: Pointer(float32), float32, [64, 64, 3, 3], []),
             in: Buffer(in_2: Pointer(float32), float32, [1, 64, 58, 58], [])}
  buffer_map = {in_1: in, w_1: w} {
  attr [compute: Pointer(float32)] "storage_scope" = "global";
  allocate(compute, float32, [200704]);
  attr [compute_1: Pointer(float32)] "storage_scope" = "global";
  allocate(compute_1, float32, [3136]);
  attr [IterVar(pipeline: int32, (nullptr), "ThreadIndex", "pipeline")] "
    pipeline_exec_scope" = 1;
  for (i1: int32, 0, 64) {
    for (yy: int32, 0, 56) {
      for (xx: int32, 0, 56) {
        compute_1[((yy*56) + xx)] = 0f32
  ...
```

Listing 5.9: Relay IR for kernel with constant shapes

```
primfn(in_1: handle, w_1: handle) -> ()
  attr = {"global_symbol": "main", "tir.noalias": True}
  buffers = {w: Buffer(w_2: Pointer(float32), float32, [ff: int32, rc: int32, 3, 3], [
    stride: int32, stride_1: int32, stride_2: int32, stride_3: int32], type="auto"),
             in: Buffer(in_2: Pointer(float32), float32, [1, rc, xx: int32, xx], [
    stride_4: int32, stride_5: int32, stride_6: int32, stride_7: int32], type="auto")}
  buffer_map = {in_1: in, w_1: w} {
  attr [compute: Pointer(float32)] "storage_scope" = "global";
  allocate(compute, float32, [((ff*(xx - 2))*(xx - 2))]);
  attr [compute_1: Pointer(float32)] "storage_scope" = "global";
  allocate(compute_1, float32, [((xx - 2)*(xx - 2))]);
  attr [IterVar(pipeline: int32, (nullptr), "ThreadIndex", "pipeline")] "
    pipeline_exec_scope" = 1;
  for (i1: int32, 0, ff) {
    for (yy: int32, 0, (xx - 2)) {
      for (xx_1: int32, 0, (xx - 2)) {
        compute_1[((yy*(xx - 2)) + xx_1)] = 0f32
  ...
```

Listing 5.10: Relay IR for kernel with symbolic shapes

In Listing 5.9, the constants for loop boundaries and array reference expressions are folded as seen in lines 7, 9, 11-14. With symbolic shapes, loop boundaries and array subscript expressions are a function of strides (lines 3 and 4) that are kernel integer arguments in global memory. These arguments can be set to execute layers with different shapes, whereas with constant shapes, the kernel can execute only one layer. We use symbolic shape execution to automatically generate a kernel that is time-multiplexed for multiple layers in a network.

There is a caveat to using kernels generated with symbolic shapes in that the generated array subscript expressions prevent memory accesses from being coalesced. For the innermost reduction loop with the iterator `rx`, there exists a stride that despite always having a value of 1 in any case, AOC is unable to coalesce memory accesses since it cannot be determined at compile-time that the accesses are contiguous. We work around this by explicitly defining these constants with a value of 1, as it is shown in Listing 5.11.

```
{
const int stride_2 = 1;
...
              pad_temp_local[(0)] = in[(((((rc_outer * 4) * stride) + ((i2 * 2) *
    stride1)) + ((i3_outer * 14) * stride2)))];
              pad_temp_local[(1)] = in[(((((rc_outer * 4) * stride) + ((i2 * 2) *
    stride1)) + (((i3_outer * 14) + 1) * stride2)))];
              pad_temp_local[(2)] = in[(((((rc_outer * 4) * stride) + ((i2 * 2) *
    stride1)) + (((i3_outer * 14) + 2) * stride2)))];
              pad_temp_local[(3)] = in[(((((rc_outer * 4) * stride) + ((i2 * 2) *
    stride1)) + (((i3_outer * 14) + 3) * stride2)))];
...
}
```

Listing 5.11: Coalescing memory accesses by manually setting stride

5.4 Summary

We have shown how tiling, unrolling, write caches, and parameterized kernels for convolutions and dense layers are automatically applied using TVM primitives and symbolic shape kernels.

To support channels, Relay IR must be expanded to support pipeline operations. The OpenCL codegen must be expanded to translate these operations into Intel channel intrinsics. Kernels that have only channel operations and no buffers in global memory can be marked autorun. Concurrent execution requires changes to the host code so that a kernel is created per kernel, rather than per device.

We implement channels, autorun kernels, and concurrent execution by hand to demonstrate its viability and impact on performance. However, the scope of this work is to optimize at the scheduling level, i.e., without making modifications in the IR or codegen infrastructure. Thus, we leave the automation of these optimizations for future work.

Chapter 6

Evaluation

This chapter presents the evaluation of the compilation and deployment of three CNNs on three FPGA platforms using our approach. The evaluation is segmented into three sections:

1. An assessment of a manual application of optimizations in Section 6.3. These transformations are applied to assess the viability and performance of our approach before and after optimization. LeNet is deployed to demonstrate pipelined execution and MobileNet is deployed to demonstrate folded execution. Then, we automate the application of some of these transformations and validate this process.

2. An evaluation of the optimized deployments of LeNet-5 (Section 2.1.3), MobileNetV1 (Section 2.1.4), and ResNet-18/ResNet-34 (Section 2.1.5) CNNs in Section 6.4. We also compare the performance of our optimized deployment to highly-optimized deployments on CPU and GPU.

3. A comparative analysis between this work and three closely related works in Section 6.6. We examine the differences in functionality and performance.

The methodology and performance metrics are described in Section 6.1. The platforms we use are listed in Section 6.2. We summarize our results in Section 6.7.

6.1 Methodology and Metrics

6.1.1 Methodology

For LeNet-5, we define the model in Keras and train the network using an NVIDIA GTX 1060. For MobileNet, we use the network architecture and pretrained ImageNet parameters from Keras Applica-

tions [39], a library of DNN models and pretrained weights. Lastly, for the ResNets we use the models and pretrained parameters from the `image-classifiers` Python library.

We use the test set of 10000 images from the MNIST dataset to test LeNet. For MobileNet and ResNets, we use randomly generated ImageNet-size inputs because input values do not alter computation time. These inputs are still generated on the host and copied to the device (i.e., device transfer time is still accounted for in our measurements). A real image is used to validate the implementation once before evaluating performance.

6.1.2 Metrics

We evaluate the performance of our designs using two metrics:

1. Frames per second (FPS): number of forward passes that can be processed in a second
2. Floating-point operations per second (FLOPS): number of multiply and add operations that can be performed in a second

To calculate FPS, we measure the execution time t it takes to classify N images and calculate N/t. This measurement is made using the OpenCL kernel event profiler. This is a straight forward measurement that is useful when comparing real-world performance of different platforms using identical benchmark networks.

However, often published work use different benchmark networks in addition to different FPGA platforms and arithmetic precisions. Thus, we also calculate the number of floating-point operations computed per second or FLOPS. With integer representations, we refer to this metric as OPS.

To calculate FLOPS, we require the total number of floating-point (FP) operations computed in one forward pass of the network. FLOPS is a rate of floating operations performed per second, and is equivalent to $2\times$ multiply-accumulates (MACCs) sometimes used in existing work[1]. FLOPS is computed as a product of FPS and number of FP operations.

6.2 Platforms

We conduct our experiments on three different FPGA platforms. Two of these are provided by the Intel Labs Academic Compute Environment [41] with the following FPGA system classes: (a) fpga-pac-a10, a PCIe Programmable Acceleration Card with an Arria 10 GX FPGA (referred to as the A10) and (b) fpga-pac-s10, a PCIe D5005 Programmable Acceleration Card with a Stratix 10 SX FPGA (S10SX).

[1]Some authors consider a fused multiply-add to be one floating-point operation. Calculations of FP operatioons in this work consider assume addition and multiplication to be separate operations.

Platform	Arria 10 GX	Stratix 10 SX	Stratix 10 MX HBM
vLab FPGA Class	fpga-pac-a10	fpga-pac-s10	N/A
FPGA SKU	10AX115N2F40E2LG	1SX280HN2F43E2VG	1SM21CHU2F53E1VG
Board	Intel PAC w/ Arria 10 GX	Intel PAC D5005	Intel Stratix 10 MX HBM Development Board
External Memory	[8 GB DDR4, 2 banks]	[32 GB DDR4, 4 banks]	[8 GB HBM2, 32 channels DDR4 unsupported by BSP]
Theoretical Peak BW	34.1 GB/s	76.8 GB/s	12.8 GB/s/PC 409.6 GB/s total
PCIe	Gen 3×8	Gen 3×16	Gen 3×8
Host CPU	Intel Xeon Platinum 8180	Intel Xeon Platinum 8280	Intel i9-7940X
CPUs×Cores/Threads	2×28/56	2×28/56	1×14/28
Host Memory	384 GB DDR4	768 GB DDR4	64 GB DDR4
Process	20 nm	14 nm	14 nm
Quartus Ver.	17.1.1	18.1.2	19.1

Table 6.1: Overview of FPGA platforms used

Evaluation Platform	Arria 10 GX	Stratix 10 SX	Stratix 10 MX HBM
Engineering Sample	No	No	Yes
Total ALUTs	740500	1666240	1405440
Total FFs	1481000	3457330	2810880
Total RAMs	2336	11254	6847
Total DSPs	1518	5760	3960
Static Partition ALUTs	113900 (15%)	200000 (12%)	13132 (1%)
Static Partition FFs	227800 (15%)	275150 (8%)	20030 (1%)
Static Partition RAMs	377 (16%)	467 (4%)	112 (2%)
Static Partition DSPs	0 (0%)	0 (0%)	0 (0%)

Table 6.2: Overview of chip resources and static partition (see Section 2.4.1) resource utilization on evaluated FPGAs

Platform	CPU	GPU
Processor	Intel Xeon Platinum 8280	NVIDIA GTX 1060
Cores/Threads	2×28/56	1280 CUDA
Memory	768 GB DDR4	6 GB GDDR5
Cache	L1/L2/L3 1.75/28/38.5 MB	L1/L2 48/1536 KB
Clock	2.7 - 4.0 GHz	1.5 - 1.7 GHz
Family	Cascade Lake	Pascal
Process	14 nm	16 nm
Notes	AVX-512 instructions enabled	CUDA 10.1.105, cuDNN 7.6

Table 6.3: Overview of CPU and GPU platforms used

The third platform is an Intel Stratix 10 MX Development Kit with 16 GB of HBM2 memory (S10MX). This board is enabled with 32 HBM pseudo-channels (PCs), where each PC is a 256-bit data interface that operates at 400 MHz at a capacity of 256 MB. Unlike the other two devices, the BSP for the S10MX does not support banking and the location of CL buffers must be explicitly mapped to a PC. In this work, only one HBM PC is utilized. In addition, the card that we have is an engineering silicon sample with an experimental and unsupported BSP from Intel. As a result, performance may be lower compared to other platforms and to production Stratix 10 MX boards.

The three FPGA platforms are summarized in Table 6.1. In comparison to the Stratix 10, the Arria 10 has around half the number of logic elements and a quarter of the RAM and DSP blocks. A summary of available FPGA resources and static partition (see Section 2.4.1) area utilization is provided in Table 6.2.

We evaluate CPU performance on server-grade Intel Xeon Platinum 8280s with 28 cores/56 threads. The evaluation system has two physical CPUs for a total of 56 cores and 112 threads with 768 GB DDR4 memory. We use an NVIDIA GTX 1060 with 6 GB of VRAM for the GPU evaluations. Although this is an older generation consumer GPU, this is what was available to us when carrying out this work. A summary of CPU/GPU platforms is provided in Table 6.3.

CPU and GPU performance is measured using two frameworks: Keras with Tensorflow labeled TF-CPU[2], and the LLVM-CPU backend in TVM executing with n threads labeled as TVM-nT. GPU performance is measured using Tensorflow for the GPU, with the CUDA Deep Neural Network Library (cuDNN) and is labeled as TF-cuDNN.

For all experiments, we use TVM release v0.7 (commit `728b829`), Keras 2.3.1, and Tensorflow 2.1.0.

6.3 Evaluation of Manual Optimizations

The evaluation of manual optimizations consists of three steps. In Section 6.3.1, we demonstrate the effectiveness of optimizations for a network (LeNet) where all kernels for the network fit on-chip. We propose and apply the following: loop unrolling, loop fusion, write caches, OpenCL channels, autorun kernels, and concurrent execution. Each optimization is applied by hand, and a unique bitstream is generated to examine the cumulative effects on accelerator performance and area.

In the second step, we implement loop unrolling, loop fusion, and write caches with TVM scheduling primitives. We generate kernel code with these transformations automatically applied by TVM. OpenCL

[2]Tensorflow makes use of a thread pool to exploit intra/inter-op parallelism. We use the default configuration, which uses all threads available on the CPU. We observe that TF uses 4 threads for LeNet and all 112 threads for MobileNet, ResNet-18, and ResNet-34.

Bitstream	Applied	Unrolling	CL Channels	Autorun	Note
Base	Manual	✓[4]			Default TVM schedule.
Unrolling	Manual	✓			Convolution inner product loop unrolled ($F \times F$). Layers dense1, dense2, dense3 unrolled by a factor of 40/40/4.
Channels	Manual	✓	✓		OFM written to next layer via buffered channel. Activation fused with channel operation.
Autorun	Manual	✓	✓	✓	Pooling and flatten layers declared autorun (kernels execute as soon as buffer is full) .
TVM-Autorun	Automatic	✓	✓	✓	Same as Autorun, but unrolling and activation fusing implemented with TVM scheduling. .

Table 6.4: Generated bitstreams with different optimizations for LeNet-5

channels and autorun kernels are still applied by hand to maintain a fair comparison between the fully hand-optimized kernels and automatically optimized kernels. This is done to validate the automation of loop unrolling, fusion, and write caches.

In the final step, we demonstrate the effectiveness of optimizations for a network (MobileNetV1) where kernels do not fit on-chip in Section 6.3.2. We apply and justify a different set of optimizations for MobileNet: loop tiling, unrolling, and kernel reuse by hand.

For these experiments, we implement and use a OpenCL host program independent of TVM for two reasons: to isolate behavior/interactions with the TVM-generated host code, and to implement layer-pipelined execution with concurrent execution and channels.

6.3.1 Pipelined Execution for LeNet

Performance Results

We begin our evaluations with LeNet-5, a lightweight CNN for classifying MNIST handwritten digits with 389K FP operations and 60K parameters[3]. Five bitstreams (listed in Table 6.4) are generated for LeNet-5, each building upon optimizations of the preceding one.

In addition, we compare the performance of serial and concurrent execution of kernels. For serial and concurrent execution (labeled [CE] in our plots) we declare a single command queue and as many command queues as kernels respectively. We classify test set images and plot the effect of each optimization on the base configuration in Figure 6.1. We make the following observations:

- The base bitstream operates at 568, 524, and 402 FPS for the S10MX, S10SX, and the A10 respectively.

[3]Parameters include trainable numerical values in the network (i.e., weights and biases).

[4]Quartus versions ($<$ 19.1) for A10 and S10SX automatically unroll loops with a small trip count. This includes a $F \times F$ unroll factor for these platforms.

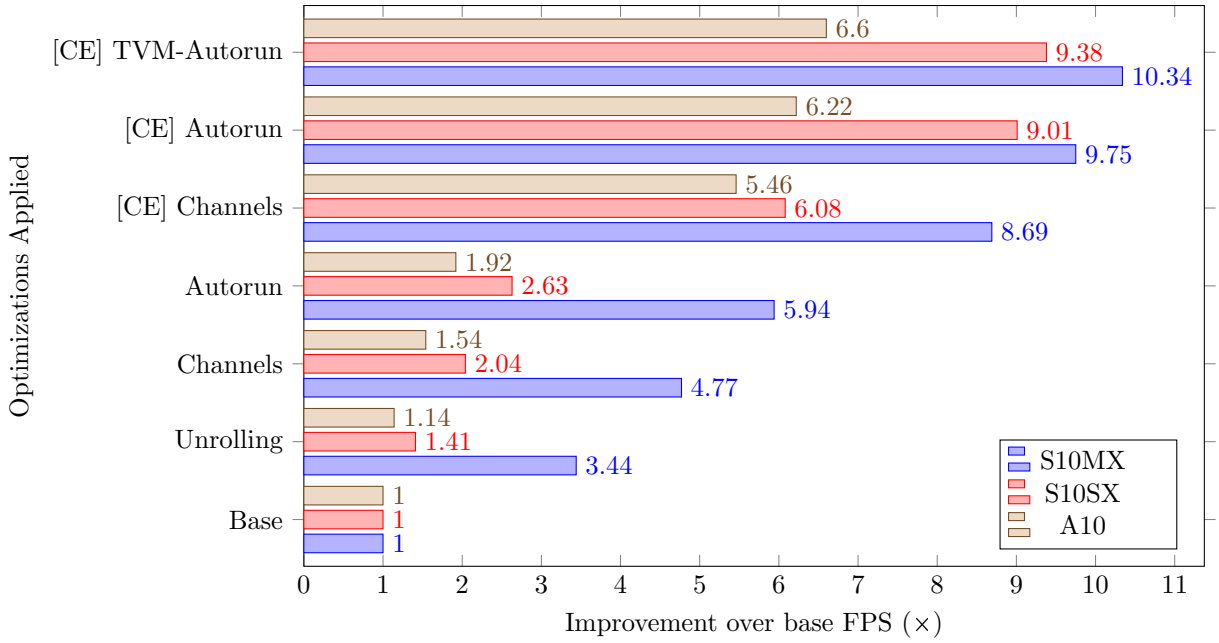

Figure 6.1: Impact of optimizations on FPS for 5 bitstreams on 3 different platforms

- Unrolling improves runtime by a factor of 1.14× and 1.41× with the A10 and the S10SX. The S10MX has a more significant increase, with a 3.44× improvement. The difference is that in the base bitstream for the S10SX and A10, the inner reduction loops are already unrolled automatically by the compiler. In other words, the improvements for the S10SX and A10 are gained from unrolling the fully-connected layers.

- Channels improves runtime by a factor from 1.54× to 4.77×. Moving activations on-chip through registers incurs less cycles than accessing global memory, and this is evident in the observed improvement.

- Autorun improves runtime by a factor from 1.92× to 5.94×. This is due to the fact that the host no longer needs to issue tasks for the pooling and flattening layers, thus the overhead from executing OpenCL kernels is reduced. This overhead is significant for LeNet, which has kernels with short compute times.

- Enabling concurrent execution has a significant effect on performance on the channel-enabled and autorun bitstreams, with improvements ranging from 5.46× to 9.75× over base. The best configuration is the autorun bitstream with concurrent execution. For the S10MX, S10SX, and A10 platforms respectively, the bitstream performs at 1609 FPS (9.75×), 4917 FPS (9.01×), and 2499 FPS (6.22×). Concurrent execution with channels is an implementation of layer-pipelined

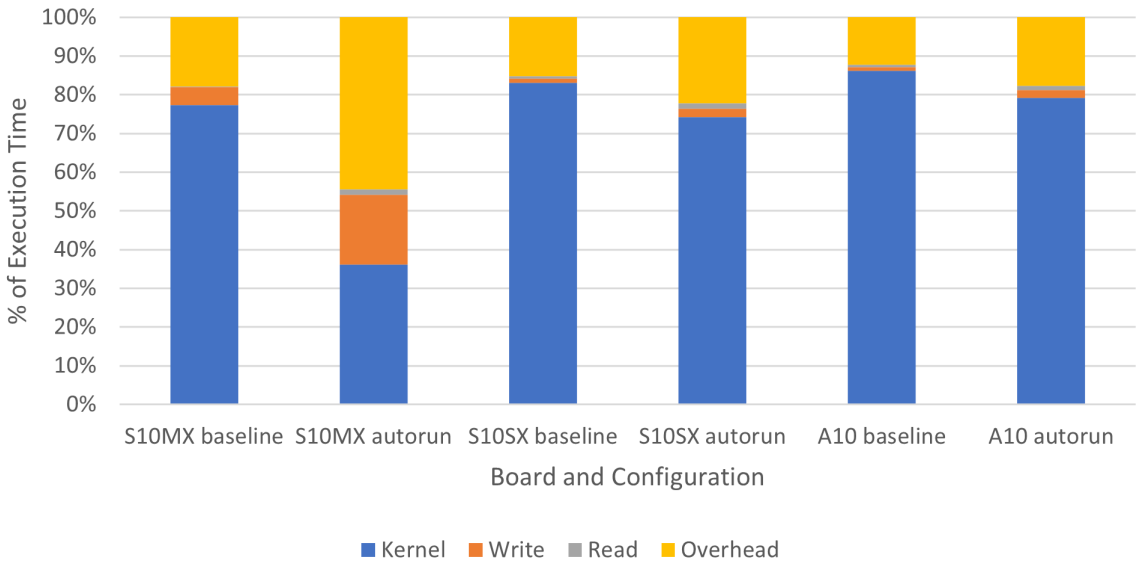

Figure 6.2: OpenCL event profiling for the baseline and autorun kernels for each platform. Note that most of the overhead in this image can be attributed to profiling since kernel times are short.

	S10MX				S10SX				A10			
Fit Report	Logic	RAM	DSP	f_{max}	Logic	RAM	DSP	f_{max}	Logic	RAM	DSP	f_{max}
Base	32%	21%	3%	250	32%	21%	3%	209	39%	81%	8%	201
Unrolling	44%	38%	7%	259	32%	23%	5%	202	45%	83%	13%	210
Channels	32%	26%	6%	318	24%	18%	5%	234	29%	45%	21%	192
Autorun	32%	26%	6%	307	24%	18%	5%	220	28%	45%	21%	200
TVM-Autorun	36%	26%	4%	300	25%	19%	5%	218	36%	37%	14%	217

Table 6.5: Area usage for each bitstream for LeNet-5

execution, which results in the largest performance improvement for LeNet.

- Automatically implementing unrolling and write caches through the use of TVM schedule primitives results in slightly better results than hand-applied optimizations, performing at 1706 FPS (10.34×), 4917 FPS (9.38×), and 2653 (6.60×) faster than the base. This shows that the automation of these transforms is effective.

The S10MX performs relatively poorly compared to the A10 and the S10SX. This is due to reduced host-to-device bandwidth. We measure the breakdown of runtime on kernel execution, write and read events in Figure 6.2 and observe that for the S10MX, the time spent on data transfer is significantly larger than the other two platforms, particularly for writes. We also measure host-to-device bandwidth and show these results in Appendix A.

We show board resource utilization, based on Quartus' fitter reports, in Table 6.5. We observe that unrolling increases utilization in logic, RAM blocks, and DSP blocks. This is expected since unrolling

replicates loop bodies. This is especially the case for the S10MX, where the inner reduction loops in convolutions are not automatically unrolled in the base bitstream. Therefore, the increase in logic, RAM, and DSP utilization is larger in the S10MX when moving from base to unrolling, compared to the S10SX and the A10.

Channels reduce the number of RAMs since caches for loading activations are no longer required. We see that this is the case for the S10MX; RAM usage drops from 38% to 26%. It can also increase f_{max} as it is observed for the S10MX, from 259 to 318 MHz. This can be attributed to reduced routing complexity when moving the transfer of input/output activations from global memory LSUs to directly between kernels via registers. Finally, autorun has no effect on neither area usage nor operating frequency; small reductions/increases in f_{max} can be attributed to variation in placement/routing.

6.3.2 Folded Execution for MobileNet

In this section, we explore folded execution for MobileNetV1 acceleration by using optimized convolution kernels that are re-used for multiple layers instead of a fully layer-pipelined arrangement of kernels.

1×1 Convolutions

We optimize a parameterized 1×1 convolution kernel that is re-used for all MobileNetV1 1×1 convolution layers on the Arria 10[5]. We choose 1×1 convolutions because it has the largest share of computations in MobileNet.

Using the same 1×1 convolution schedule described in Section 5.1.1, we choose to tile and unroll in three dimensions: input feature map rows (W_{2vec}), the output channel (C_{2vec}), and input feature map channel (C_{1vec}). The tiling/unrolling factor can range from 1, which is time-multiplexing a DSP block for the convolution layer, to the entirety of the loop trip count. We evaluate the performance of select tiling factors that evenly divide the loop sizes of the network. We list the tiling sizes and resource utilizations of the featured configurations in Table 6.6.

The default convolution schedule in TVM is the base configuration in our comparison. We use the improvement of the sum of execution time for all 1×1 convolutions in MobileNetV1 over the base schedule as the measure of performance. In Figure 6.3, two plots are made for each tiling configuration with x-axis values corresponding to the configuration numbers in Table 6.6: one for the number of DSPs (corresponding to the blue columns, and the left y-axis (Number of DSPs)) and the speedup factor for 1×1 convolution (corresponding to the orange line, and the right y-axis (Improvement)).

[5]One platform is selected for this experiment to save time on synthesis. In addition, improvements are expected to be proportional to tiling sizes across all platforms. Full deployment on all platforms is shown in the next section.

Configuration	W_{2vec}	C_{2vec}	C_{1vec}	Logic (%)	RAM (%)	DSPs	f_{max}
1	7	4	8	35	36	275	195
2	7	4	16	40	57	531	168
3	7	8	4	33	34	267	213
4	7	8	8	34	47	507	194
5	7	8	16	48	67	987	137
6	7	16	4	42	48	507	180
7	7	16	8	45	63	971	141

Table 6.6: Select tiling configurations for 1×1 convolutions with area consumption for the Arria 10. Logic and RAM utilization is presented as a percentage, but DSP blocks are presented as numbers (maximum 1518 on board).

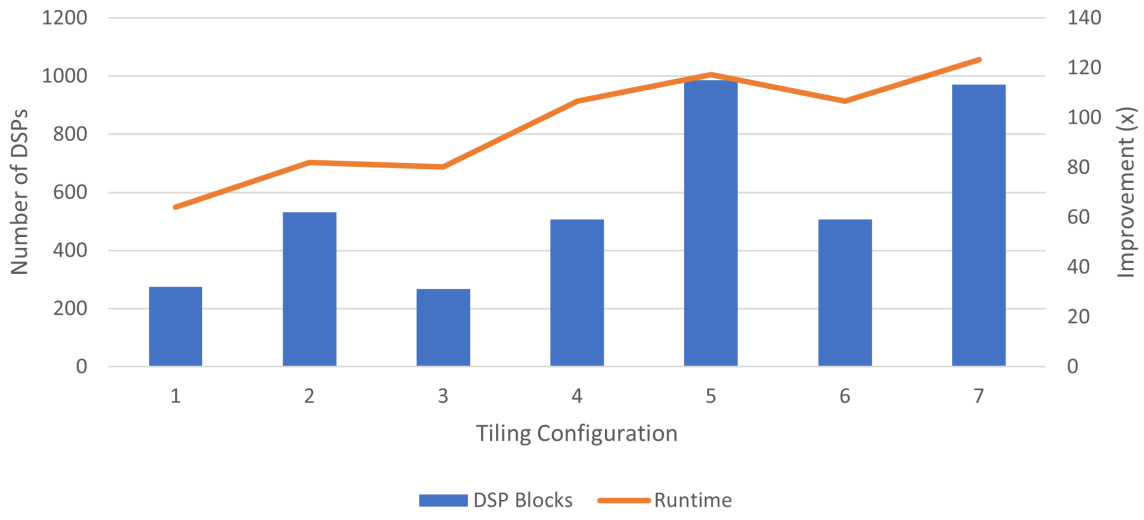

Figure 6.3: Performance of select tiling configurations for 1×1 convolutions in MobileNetV1 on the Arria 10. The column plot represents the number of DSP blocks used for the configuration and corresponds with the left vertical axis. The line plot represents the improvement over the base schedule and corresponds with the right vertical axis.

Kernel	Tiled Dimensions	Unroll Factors
1×1 conv	W_2, C_2, C_1	S10MX: 7/32/4 S10SX: 7/16/4 A10: 7/8/8
3×3 conv	C_1, F, F	$3 \times 3 \times 3$
3×3 DW conv, $S = 1$	W_2, F, F	$7 \times 3 \times 3$
3×3 DW conv, $S = 2$	W_2, F, F	$7 \times 3 \times 3$
dense	C_1	32

Table 6.7: Parameterized kernels and others used for MobileNetV1 deployment

We observe the following:

- Increased tile sizes (and therefore unroll factors) correlates directly with the number of DSP blocks used to implement the design. The increase in parallelism benefits performance.
- The base schedule takes an average of 1326.35 ms (0.79 GFLOPS) to execute all 1×1 convolutions once. With tiling and unrolling, runtime improves to a range between 20.74 ms (50.8 GFLOPS) in configuration 1 and 10.75 ms (98.0 GFLOPS) in configuration 7, which is a speedup of 64× and 123× respectively.
- Larger tile sizes result in lowered f_{max} due to increased fanout of distributing activations and weights from global memory LSUs into the DSP blocks. This stresses the router, which resorts to using longer paths. For example, while configuration 5 uses 987 DSPs in comparison to configuration 4's 507 DSPs, f_{max} drops by 30% and the incremental improvement is only 9.9%.

Network Deployment

The bitstream generated from the base schedule executes at 4743.01 ms/image or 0.21 FPS for the S10MX and 5883.19 ms/image or 0.17 FPS for the S10SX. For the Arria 10, the network does not synthesize due to insufficient board resources. Indeed, the challenges of deploying a larger network become apparent with the baseline configuration being unusable for deployment.

We proceed to deploy MobileNetV1 with the listed optimized convolution kernels in Table 6.7. For the optimized bitstream, we choose tiling configurations that have a high improvement for 1×1 convolutions that do not have severely degraded f_{max}, from those shown in the previous section. The final bitstream is synthesized with $W_{2vec}/C_{2vec}/C_{1vec} = 7/32/4$ as the tiling configuration for the S10MX, 7/16/4 for the S10SX and 7/8/8 for the A10.

After optimization, the S10MX, S10SX, and A10 achieves 17.7 FPS, 30.3 FPS, and 18.0 FPS respectively. This demonstrates a speedup of 84.0× for the S10MX and 183.8× for the S10SX over the base

Operation	% of FP Ops	S10MX GFLOPS	S10SX GFLOPS	A10 GFLOPS	S10MX Time	S10SX Time	A10 Time
1x1 conv	94.8%	43.99	88.20	57.20	47.6%	30.2%	36.3%
3x3 DW conv	3.1%	1.81	1.72	1.65	28.8%	44.5%	33.8%
3x3 conv	1.9%	4.23	8.48	6.54	8.2%	6.3%	6.0%
dense	0.2%	2.49	4.24	3.07	1.3%	1.2%	1.2%
pad	0.0%	0	0	0	12.7%	15.5%	20.7%

Table 6.8: Average optimized kernel GFLOPS and percentage of runtime for operations that take up more than 1% of runtime in MobileNet

configuration, and a successful deployment on the A10 whereas before it did not fit on the Arria 10. In terms of GFLOPS, the FPGAs operate at 19.70, 33.69, and 20.0 GFLOPS on average for MobileNet.

We show board resource utilization, based on Quartus' fitter reports, in Table 6.11. We observe an increase in DSP block usage in all three platforms which is an expected consequence of unrolling the tiled loops. On the other hand, we also see a decrease in logic up to 47% less for the S10SX compared to the base and 25% less for the S10SX in BRAM since we are grouping execution into parameterized kernels.

Wider layers in MobileNetV1 compared to LeNet with larger input feature maps and increased number of input channels can be processed more efficiently in parallel. The improvement over the baseline kernels for MobileNet is much larger at up to 183.8× faster compared to 10.34× observed for LeNet, despite the lack of layer-pipelined execution. Further, the results observed on the A10 confirm that parameterized kernels can resolve resource issues observed with the 1-to-1 mapping.

To identify areas of improvement, we profile kernel execution time and calculate the average throughput for each kernel in Table 6.8. Compared to the default schedule's baseline kernels with the S10MX, we observe speedups of 136.4×, 163.2×, 9.23×, and 2.11× on 1×1 convolutions, 3×3 convolution, 3×3 depthwise convolutions and other layers respectively. In Table 6.8 we observe 44.0, 88.2, and 57.2 GFLOPS on 1×1 convolutions on the S10MX, S10SX, and A10 respectively. The 3×3 convolution layer and depthwise layers however see GFLOPS that is around a tenth of that, or lower. Evidently, throughput across layers is currently unbalanced with manually-controlled parallelism.

In addition, zero padding becomes a significant portion of the optimized runtime ranging from 12.7% to 20.7%. Since padding does not perform computation, this decreases the throughput of our accelerator. The generated padding kernel uses modulo[6] addressing and a conditional statement whether to write a zero or the buffer value which may be efficient in other platforms, but does not generate efficient hardware.

In summary, the results above show us that:

- Tiling and unrolling in multiple dimensions can speed up 1×1 convolutions between a factor of

[6] An expensive operation for FPGAs, and is not recommended by Intel [35].

64× and 123× over the baseline FPGA schedule for the Arria 10.

- Parameterized kernels enable high performance convolution kernels on all platforms and make end-to-end MobileNet deployment possible on the Arria 10.

- The combination of our optimizations can improve baseline MobileNet performance up to 183.8× evaluated on three different FPGA platforms.

6.3.3 Summary

In summary, our hand-application of optimization demonstrates the following:

- Pipelining kernels is desirable for small networks when all layers can fit on chip, and parallelism does not cause kernels to exceed memory bandwidth. We implement layer-pipelined activations with channels on LeNet-5 with unrolling that can improve baseline performance up to 10.34×.

- Automatically unrolled kernels implemented with TVM schedule primitives match/marginally exceed hand-optimized designs, demonstrating the effectiveness of our automation of these optimizations within TVM.

- Parameterized kernels is desirable for networks with many layers, where pipelining causes exhaustion of soft logic required to implement each kernel or memory bandwidth due to the transfer of large parameters that exceed on-chip cache capacity. We implement parameterized kernels that are tiled on MobileNetV1 that can improve baseline performance up to 183.8×, and allows the network to fit on the Arria 10.

6.4 Inference Performance

In this section, we present comparative evaluations of the optimized deployment flow for LeNet-5, MobileNetV1, ResNet-18, and ResNet-34 on FPGA, CPU, and GPU platforms.

6.4.1 LeNet-5

The performance in FPS, GFLOPS and resource utilization across FPGAs are listed in Table 6.9. The table summarizes FPGA performance and resource utilization. Optimized FPGA bitstreams are labeled as the S10MX, S10SX, and A10. Baseline bitstreams are labeled with the suffix “Base”.

[7] The S10MX experiences a slowdown due to high buffer write times. See Section 6.3.1 for details.

LeNet-5	S10MX-Base	S10MX	S10SX-Base	S10SX	A10-Base	A10
FPS	564	1706	524	4917	402	2653
CNN FP Ops	389K					
Parameters	60K					
GFLOPS	0.22	0.66	0.20	1.91	0.16	1.03
Speedup (×)	-	3.02×	-	9.38×	-	6.60×
Logic (%)	32%	36%	32%	25%	39%	36%
BRAM (%)	21%	26%	21%	19%	81%	37%
DSP (%)	3%	4%	3%	5%	8%	14%
f_{max} (MHz)	250	300	209	218	201	217
Loop Unrolling	-	✓	-	✓	-	✓
Write Caches	-	✓	-	✓	-	✓
Channels	-	✓	-	✓	-	✓
Autorun	-	✓	-	✓	-	✓
Conc. Exec.	-	✓	-	✓	-	✓

Table 6.9: Comparison of FPS, GFLOPS, and resource usage across FPGAs for LeNet-5 inference

	FPS	TF-CPU *1075*	TVM-1T *2345*	TF-cuDNN *1604*
S10MX	*1706*	1.59×	0.73×[7]	1.06×
S10SX	*4917*	4.57×	2.10×	3.07×
A10	*2653*	2.47×	1.13×	1.65×

Table 6.10: Comparison of FPS across CPU, GPU, and FPGAs for LeNet-5 inference. Italicized numbers represent raw FPS.

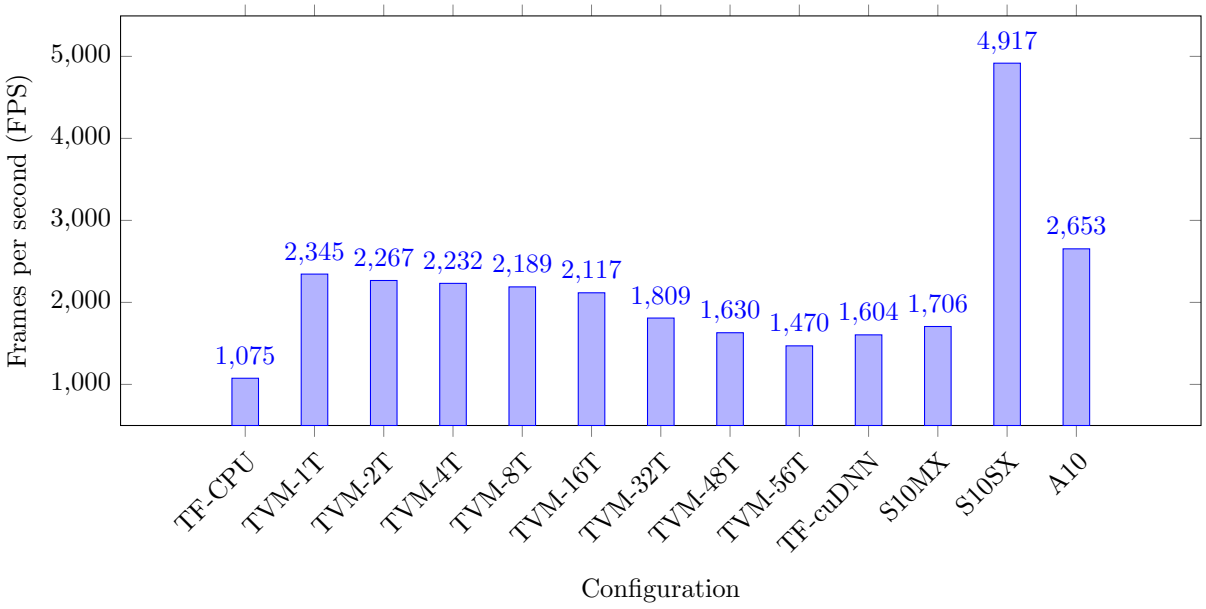

Figure 6.4: CPU, GPU, and FPGA performance in FPS on LeNet-5 inference

Table 6.10 lists the performance results for Tensorflow (labelled TF-CPU), TVM (labelled TVM-1T), GPU-Tensorflow with cuDNN (labelled TF-cuDNN). Raw FPS values are italicized while cell values represent FPGA (in row) speedup over the reference platform (in column). Additionally, accelerator performance in terms of FPS is plotted in Figure 6.4 for TF-CPU, TVM with a sweep on the number of threads (ranging from 1 thread to 56 threads[8]), TF-cuDNN, and FPGAs.

As reported in Section 6.3.1 for the FPGAs, the base (Base from Section 6.3.1) bitstream operates at 568, 524, and 402 FPS for the S10MX, S10SX, and the A10 respectively. After optimization, the best configuration (TVM-Autorun from Section 6.3.1) operates at 1706, 4917, and 2953 FPS. This demonstrates speedups of 3.02×, 9.38×, and 6.60× over the base bitstream. In terms of GFLOPS, the FPGAs operate at 0.7, 1.9, and 1.0 GFLOPS on average.

The CPU achieves 1075 FPS with Tensorflow and 2345 FPS with TVM-1T, which is the best performing CPU configuration. Compared to Tensorflow, the S10MX, S10SX, and A10 are 1.59×, 4.57×, 2.47× faster with all of our optimizations. Compared to TVM, the S10MX, S10SX, and A10 are 0.73×, 2.10×, and 1.13× faster. Both Tensorflow and TVM are highly optimized ML frameworks and are considered state-of-the-art CPU implementations. These speedups demonstrate that our optimizations can have a significant impact on automatically generated FPGA kernels to the extent that it can surpass the performance of optimized implementations on CPUs.

The GPU achieves 1604 FPS with Tensorflow/cuDNN, a highly optimized deep learning kernel library from NVIDIA. We demonstrate that the FPGAs are 1.06×, 3.07×, 1.65× faster. Given that the network is small and has few parameters, the larger memory bandwidth on GPUs does not provide an advantage since the weights can be stored in on-chip caches with the FPGAs. Further, since batching is not used, it is possible that the GPU is underutilized for a network this size whereas we are able to have increased utilization with layer-pipelined execution (i.e, channels and concurrent execution). Therefore, the FPGAs can demonstrate superior performance.

We conclude that the optimized bitstreams across the three FPGA platforms can accelerate LeNet up to 2.47× faster than TF-CPU, up to 2.10× faster than TVM, and up to 3.07× faster than TF-GPU.

6.4.2 MobileNetV1

The performance in FPS, GFLOPS and resource utilization across FPGAs are listed in Table 6.11. The table summarizes FPGA performance and resource utilization. Optimized FPGA bitstreams are labeled as the S10MX, S10SX, and A10. Baseline bitstreams are labeled with the suffix “Base”.

[8]We observe a decrease in performance as the number of threads increase, but do not investigate the cause.

MobileNetV1	S10MX-Base	S10MX	S10SX-Base	S10SX	A10-Base	A10
FPS	0.21	17.7	0.17	30.3	na	18.0
CNN FP Ops	1.11G					
Parameters	4.2M					
GFLOPS	0.23	19.70	0.19	33.69	na	20.0
Speedup (×)	-	84.3×	-	178.2×	-	-
Logic (%)	73%	51%	71%	46%	na	72%
BRAM (%)	46%	34%	51%	48%	na	88%
DSP (%)	2%	34%	3%	15%	na	40%
f_{max} (MHz)	232	198	166	187	na	181
Loop Unrolling	-	✓	-	✓	-	✓
Loop Tiling	-	Table 6.7	-	Table 6.7	-	Table 6.7
Write Caches	-	✓	-	✓	-	✓

Table 6.11: Comparison of FPS, GFLOPS, and resource usage across CPU, GPU, and FPGAs for MobileNetV1 inference

	FPS	TF-CPU *21.6*	TVM-1T *15.6*	TVM-16T *90.1*	TF-cuDNN *43.7*
S10MX	*17.7*	0.82×	1.13×	0.20×	0.41×
S10SX	*30.3*	1.40×	1.94×	0.34×	0.69×
A10	*18.0*	0.83×	1.15×	0.20×	0.41×

Table 6.12: Comparison of FPS across CPU, GPU, and FPGAs for MobileNetV1 inference. Italicized numbers represent raw FPS.

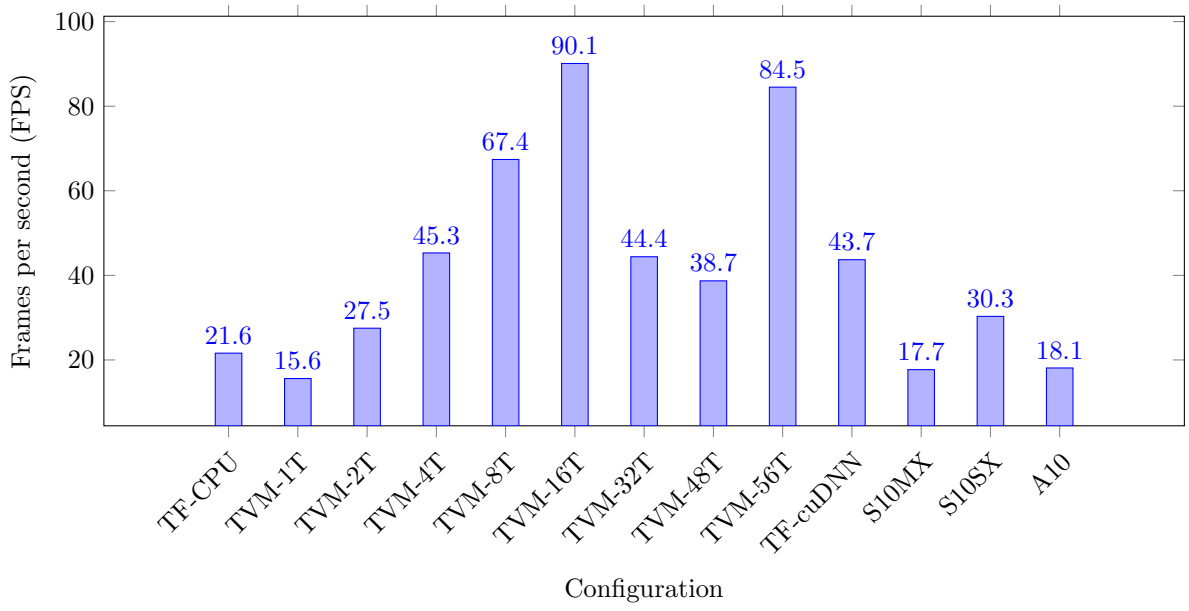

Figure 6.5: CPU, GPU, and FPGA performance in FPS on MobileNetV1 inference

Table 6.12 lists the performance results for Tensorflow (labelled TF-CPU), TVM (labelled TVM-1T), GPU-Tensorflow with cuDNN (labelled TF-cuDNN). Raw FPS values are italicized while cell values represent FPGA (in row) speedup over the reference platform (in column). Additionally, accelerator performance in terms of FPS is plotted in Figure 6.5 for TF-CPU, TVM with a sweep on the number of threads (ranging from 1 thread to 56 threads[9]), TF-cuDNN, and FPGAs.

As reported in Section 6.3.2 for the FPGAs, the base bitstream operates at 0.21, and 0.17 FPS for the S10MX, and the S10SX respectively. The naive mapping without parameterized kernels does not synthesize for the A10. After optimization, the FPGAs operate at 17.7, 30.3, and 18.0 FPS. This demonstrates speedups of 84.3× and 178.2× over the base bitstream. In terms of GFLOPS, the FPGAs operate at 19.70, 33.69, and 20.0 GFLOPS.

The CPU achieves 21.6 FPS with Tensorflow. With TVM, performance ranges from 15.6 FPS to 90.1 FPS from 1 to 56 threads. Compared to Tensorflow, the FPGAs operate at a factor of 0.82×, 1.40×, and 0.83× in FPS. Compared to TVM, the S10MX and A10 operate at a speed comparable to 1 thread, about 1.13× faster for the S10MX and at 1.15× for the A10. The S10SX is comparable to 2 threads, operating at 1.1× the performance of a 2-threaded TVM. The GPU is able to operate at 43.7 FPS, which puts the FPGAs at 0.41×, 0.69×, and 0.41× of its performance.

Our optimizations demonstrate a much higher max speedup with 183.8× than the one we observed in LeNet, with a maximum of a 9.75×. While that speedup is sufficient to surpass CPU and GPU performance for LeNet, the gap between base FPGA configuration and CPU/GPU implementations is significantly larger.

The reasons for this are twofold. First, the CPU and GPU references operate much more efficiently with increased parallelism. For example, the x86 schedule in TVM parallelizes the computation of output feature maps (C_{2vec}) on different cores. Since C_2 is small in LeNet, the CPU did not benefit from multi-threading. In MobileNetV1, C_2 is large; consequently, near-linear improvements with increasing number of threads (up to 16 threads) can be observed in Figure 6.5. Second, the performance of 3×3 depthwise convolutions and the other layers (padding, softmax, dense) currently limits our implementation (see Section 6.5 for a detailed discussion).

We conclude that the optimized bitstreams across the three FPGA platforms can accelerate MobileNetV1 ranging from 0.82× to 1.40× faster than TF-CPU, 1.13× to 1.94× faster than TVM-1T, and is slower by a factor of 0.41× to 0.69× than TF-GPU. See Section 6.3.2 for a discussion of the factors that limit MobileNet performance.

[9] We do not observe a linear increase in performance with the number of threads, but do not investigate the cause.

Kernel	Tiled Dimensions	Unroll Factors
7×7 conv	F, F	7×7
3×3 conv, $S = 1$	W_2, C_1, F, F	7/8/3/3
3×3 conv, $S = 2$	W_2, C_1, F, F	7/8/3/3
1×1 conv	C_1	8
3×3 pool	F, F	3×3
Softmax	na	1 (not unrolled)

Table 6.13: Parameterized kernels and others used for ResNet deployment

6.4.3 ResNet

In a similar fashion to our MobileNet deployment, we parameterize convolution kernels listed in Table 6.13 and deploy ResNet-18 and ResNet-34. For both networks, the set of kernels are identical and only differ in the number of kernel calls (ResNet-34 has more repetitions of the same residual block substructure behind ResNet-18).

Performance in FPS and GFLOPS and resource utilization across FPGAs are listed in Table 6.14. Table 6.15 lists the performance results for CPU and GPU. Raw FPS values are italicized while values in intersecting cells represent FPGA (in row) speedup over the reference platform (in column). Additionally, accelerator performance in terms of FPS is plotted on the y-axis in Figures 6.6 and 6.7 for TF-CPU, TVM with a sweep on the number of threads (ranging from 1 thread to 56 threads), TF-cuDNN, and FPGAs.

The base schedule executes ResNet-18 at 146.3 s/image or 6.83×10^{-3} FPS for the S10MX and 120.4 s/image or 8.30×10^{-3} FPS for the S10SX, and ResNet-34 at 308.3 s/image or 3.2×10^{-3} FPS for the S10MX and 249.2 s/image or 4.01×10^{-3} FPS for the S10SX. The base schedule does not synthesize for the A10 due to insufficient BRAMs.

After optimization, the S10MX and S10SX achieve 4.1 FPS and 7.04 FPS for ResNet-18. This demonstrates a speedup of 600× for the S10MX and 846× for the S10SX. For ResNet-34, the S10MX and S10SX achieve 2.6 and 4.6 FPS, which is a speedup of 1009× and 1150×. For the Arria 10, the network still does not synthesize due to insufficient BRAM. One reason is that there are more kernels (and therefore LSUs) in ResNet than MobileNet. Another is that more LSUs are generated for 3×3 convolutions than 1×1 convolutions in MobileNet. While unrolling over C_1 for 3×3 convolutions increases parallelism, accesses to the input activations cannot be coalesced in this case, therefore AOC creates $C_1 \times F$ LSUs. This is larger than $C_1 + W_2$ LSUs generated for 1×1 convolutions. This prevents us from scaling to higher DSP utilizations, which is a limitation.

For ResNet-18, the CPU achieves 16.3 FPS with Tensorflow. With TVM, performance ranges from 5.8 FPS to 54.3 FPS from 1 to 56 threads. Compared to Tensorflow, the FPGAs operate at a factor

ResNet-18	S10MX-Base	S10MX	S10SX-Base	S10SX	A10-Base	A10
FPS	6.83e-3	4.1	8.3e-3	7.04	na	na
CNN FP Ops	3.66G					
Parameters	11.7M					
GFLOPS	2.50e-2	15.00	3.04e-2	25.53	na	na
Speedup (×)	-	600×	-	846×	-	-
ResNet-34	S10MX-Base	S10MX	S10SX-Base	S10SX	A10-Base	A10
FPS	3.2e-3	2.6	4.01e-3	4.6	na	na
CNN FP Ops	7.36G					
Parameters	21.8M					
GFLOPS	2.36e-2	19.41	2.95e-2	29.76	na	na
Speedup (×)	-	1009×	-	1150×	-	-
Logic (%)	72%	60%	63%	59%	na	na
BRAM (%)	42%	73%	38%	61%	na	na
DSP (%)	2%	8%	1%	16%	na	na
f_{max} (MHz)	250	197	183	125	na	na
Loop Unrolling	-	✓	-	✓	-	✓
Loop Tiling	-	Table 6.13	-	Table 6.13	-	Table 6.13
Write Caches	-	✓	-	✓	-	✓

Table 6.14: Comparison of FPS, GFLOPS, and resource usage across CPU, GPU, and FPGAs for ResNet inference

	ResNet-18 FPS	TF-CPU *16.3*	TVM-1T *5.8*	TVM-56T *54.3*	TF-cuDNN *46.5*
S10MX	*4.1*	0.25×	0.71×	0.08×	0.09×
S10SX	*7.04*	0.43×	1.21×	0.13×	0.15×
	ResNet-34 FPS	**TF-CPU** *10.7*	**TVM-1T** *1.2*	**TVM-56T** *13.7*	**TF-cuDNN** *31.7*
S10MX	*2.6*	0.24×	2.17×	0.19×	0.08×
S10SX	*4.6*	0.43×	3.83×	0.34×	0.15×

Table 6.15: Comparison of FPS across CPU, GPU, and FPGAs for ResNet inference. Italicized numbers represent raw FPS.

of 0.25×, and 0.43× in FPS. Compared to TVM, the S10MX is slower than a CPU thread, operating at 0.71× FPS. The S10SX is comparable to 1 thread, operating at 1.21× the performance of a 1-threaded TVM. The GPU is able to operate at 46.5 FPS, which puts the FPGAs at 0.09× and 0.15× its performance.

For ResNet-34, the CPU achieves 10.7 FPS with Tensorflow. With TVM, performance ranges from 1.2 FPS to 13.7 FPS from 1 to 56 threads. Compared to Tensorflow, the FPGAs operate at a factor of 0.24×, and 0.43× in FPS. Compared to TVM, the S10MX is equal to 2 threads. The S10SX is comparable to 4 threads, operating on par with 4-threaded TVM. The GPU is able to operate at 31.7 FPS, which puts the FPGAs at 0.08×, and 0.15× its performance.

The performance gap between the baseline FPGA and CPU is even larger than the one we observed with MobileNet. This is because in the baseline 1×1 convolution, there is one serial (unpipelined) loop which iterates over the output feature map channels (`ax1`). But in the baseline 3×3 convolution,

ResNet-18	% of FP Ops	S10MX GFLOPS	S10SX GFLOPS	S10MX Time	S10SX Time
3x3 S=1	82.2%	9.46	26.51	44.6%	33.5%
3x3 S=2	9.5%	12.01	31.53	2.7%	14.2%
7x7	6.5%	1.98	2.97	2.2%	17.1%
1x1	1.8%	14.00	38.28	24.0%	15.4%
pad	0.0%	11.19	25.04	22.3%	17.7%
ResNet-34	% of FP Ops	S10MX GFLOPS	S10SX GFLOPS	S10MX Time	S10SX Time
3x3 S=1	91.2%	34.91	70.36	71.6%	49.9%
3x3 S=2	4.7%	26.65	17.82	4.8%	9.3%
7x7	3.2%	8.00	9.72	7.8%	11.2%
1x1	0.9%	2.77	2.91	7.4%	10.2%
pad	0.0%	0.00	0.00	7.8%	18.0%

Table 6.16: Average optimized kernel GFLOPS and percentage of runtime for operations that take up more than 1% of runtime in ResNet

data dependencies prevent pipelining in two loops, the output FM channel and the input FM channel `rc`). Since the number of input/output channels (or filters) are much larger than the dimensions of the feature map, the trip count in these loops is large. Resolving dependencies and allowing the compiler to pipeline these loops significantly improves performance.

We profile average kernel GFLOPS and its impact on runtime in ResNets in Table 6.16. Since we have allocated the largest tiling factors to 3×3 convolutions, we see a GFLOPS up to 70.36 GFLOPS observed on ResNet-34 with the S10SX. Expectedly, the other convolution layers are slower since they do not have as much parallelism. The 7×7 convolution layer, for example, ranges from 1.98 GFLOPS to 9.72 GFLOPS. Similar to what we observed with MobileNet, the zero padding layers also constitute a large amount of runtime, from 7.8% to 22.3% of runtime on the S10MX for ResNet-34 and ResNet-18 respectively.

We conclude that while observations can improve base performance up to 1150× (observed on the S10SX with ResNet-34), we observe slowdowns from 0.24× to 0.43× on TF-CPU, performance equivalent to 1 thread for ResNet-18 and 2 to 4 threads for ResNet-34, and slowdowns from 0.15× to 0.08× on TF-cuDNN.

6.5 Limitations

In this section, we analyze factors that limit the performance of our accelerators. These factors vary across the deployed models, so we discuss each in turn.

LeNet is limited by low DSP block utilization. To improve performance, DSP block utilization must be increased. Since there are still resources remaining for all three FPGAs, computations can be unrolled to a larger extent. In convolutions, there are four other dimensions that we can exploit parallelism from: C_2, H_2, W_2, and C_1. The dot product in the dense layers can be unrolled further.

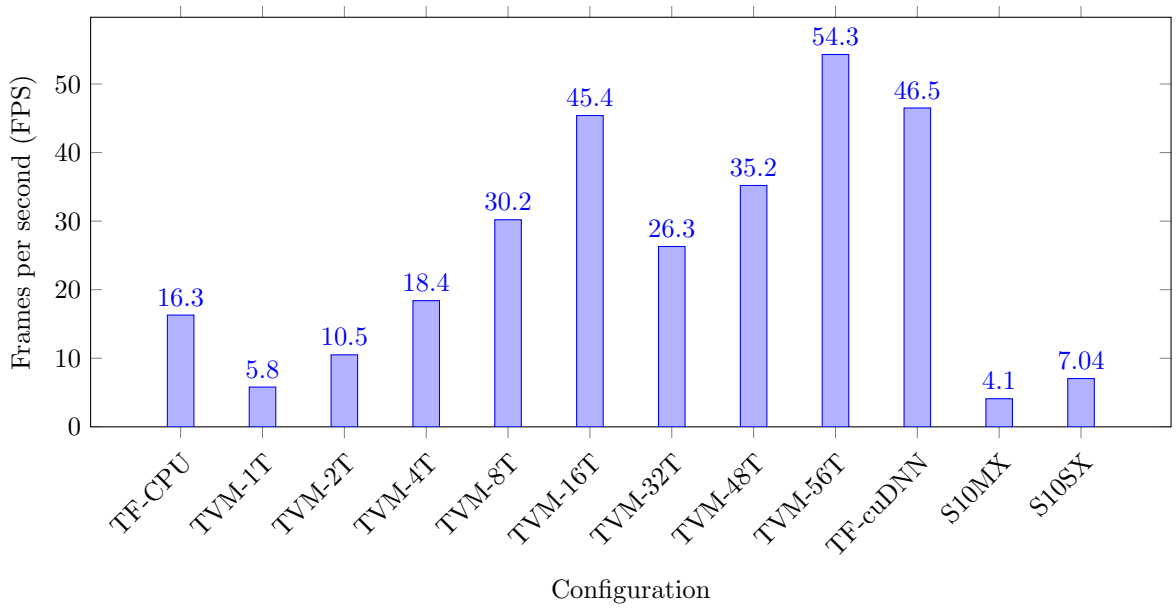

Figure 6.6: CPU, GPU, and FPGA performance in FPS on ResNet-18 inference

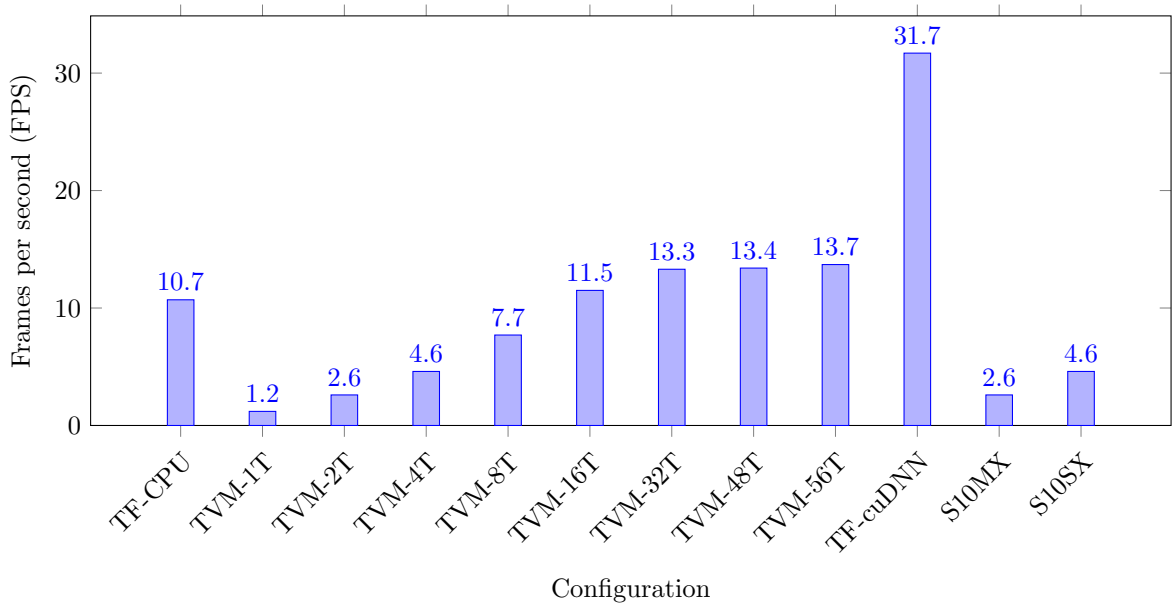

Figure 6.7: CPU, GPU, and FPGA performance in FPS on ResNet-34 inference

Since activations are pipelined with registers and weights fit into on-chip caches for LeNet, the accelerator is not memory-bound. However, when attempting to fully unroll the inner loop for each dense layer (400, 120, and 84 MACs respectively for dense0, dense1, dense2), AOC estimates 164% logic usage while only using 13% of DSPs. The bloat in logic usage is caused by the global LSU for loading weights. With plenty of DSP blocks remaining on-chip, the accelerator is logic-bound for LeNet.

This can be solved by declaring weights global-scope constants, which AOC would implement in ROMs. This removes LSUs entirely (except for the input image and the network output). While this is possible for a small network, this would severely limit the variety of networks we can deploy considering that we are deploying dense models with full single-point floating precision.

In MobileNetV1, high tiling factors cause degradation in f_{max}. Large drops in f_{max} affect overall network performance because it slows down other kernels. Congestion causes routing failure at sufficiently large factors on bigger boards. For example, a tiling configuration such as $W_{2vec}/C_{2vec}/C_{1vec} = 7/32/8$ (1792 DSPs) or 7/16/8 (896 DSPs) does not successfully route on the S10MX and the S10SX respectively, despite having enough DSPs to implement this design. In Figure 6.8, Quartus shows widespread high utilization of routing resources with 1×1 convolutions with a 7/16/8 configuration on the S10SX.

While we can improve the performance of the other convolutions, the 1×1 convolution tiling experiment shows that the accelerator is ultimately bounded by routing congestion for MobileNet. With large tiling factors, either more LSUs are generated or their bit width is widened. In addition, the fanout from distributing weights/activations from global memory causes f_{max} to drop. Eventually, congestion causes the router to fail. This prevents us from scaling our design to utilize all compute resources on chip. Since we cannot control the design beyond the functional description in OpenCL, this highlights a limitation from using AOC's memory systems.

The factors that limit ResNet are similar to the ones that limit MobileNet. The accelerator is bound by BRAM usage consumed by LSUs and prevents the use of larger tile sizes. Since there are more LSUs generated for 3×3 convolution than 1×1 convolutions (see Section 5.1.1, this makes it even more difficult to scale DSP block utilization. In addition, timing is even more difficult, with f_{max} at 125 MHz for ResNet compared to 187 MHz for MobileNet on the S10 SX.

Logic and BRAM overhead from weight and activation LSUs prevents our accelerators from fully utilizing compute resources on chip. To alleviate these issues, we can either decrease their complexity or the number of LSUs. Currently, the generated LSUs are nonaligned burst-coalesced LSUs (see Section 2.4.3. These are more complex to implement and incur larger logic overheads than their aligned counterparts. To reduce complexity, we can use OpenCL vector types, which have the same effect as unrolling while using less logic. However, this is not trivial to do with generated kernel code.

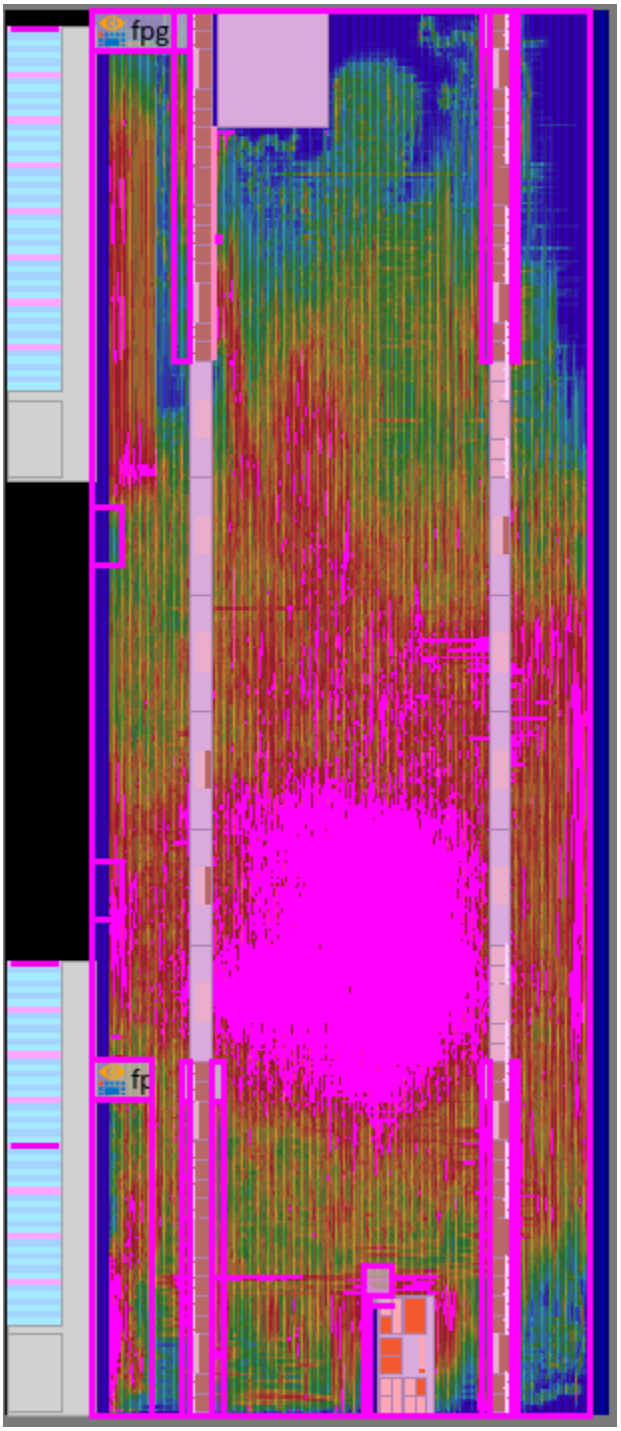

Figure 6.8: Routing utilization on the S10SX with a tiling configuration of 7/16/8 for 1×1 convolutions. Areas colored in magenta represent a routing resource utilization of over 95%.

To reduce the number of LSUs, a mix of pipelining and folding can be explored. In our experiments, we implement both in extremes, although it is possible to parameterize some components of the network while layer-pipelining others. Moving the loading of feature maps in some kernels on-chip will decrease the number of LSUs, reducing area bloat.

Finally, switching to integer arithmetic instead of floating-point will also alleviate the limitations in two ways. First, compute efficiency is higher, with two low-precision integer operations computed per cycle as opposed to one per DSP for floating-point. Second, the reduced amount of bits decreases LSU bit width and cache sizes, which alleviates LSU area bloat. We leave the exploration of quantized networks in future work.

6.6 Comparison to Existing Work

We compare the capabilities and performance of our work to those in three related works in CNN acceleration: Caffeinated FPGAs (DiCecco et al.) [18], TensorFlow to Cloud FPGAs (Hadjis et al.) [27], and DNNWeaver (Sharma et al.) [55]. Caffeinated FPGAs is a modification of the Caffe ML framework with support for a hand-optimized FPGA Winograd 3×3 convolution engine in OpenCL. TensorFlow to Cloud FPGAs is a closely related framework that allows the compilation of TensorFlow models to Amazon FPGA devices using a hardware IR called Spatial. DNNWeaver is an accelerator generation framework constructed from hand-optimized hardware templates implemented in RTL with a design space explorer.

We compare to these works to determine the competitiveness of our compiler flow against a hand-optimized HLS approach, a hardware IR generation approach (a higher abstraction than OpenCL HLS), and an RTL generation approach from hand-optimized parameterized RTL components respectively.

6.6.1 Capabilities

Caffeinated FPGAs and DNNWeaver use Caffe descriptions as input, while TensorFlow to Cloud FPGAs uses frozen models from TensorFlow. In contrast, our work uses TVM which supports all major high-level machine learning frameworks, including Caffe, PyTorch, Tensorflow, and MXNet. This makes our work more flexible in accepting inputs, which is better for adoption.

DiCecco et al. [18] implements a filter and stride specific convolution engine that is hand-designed in Xilinx SDAccel (OpenCL). This limits the framework to off-load only single-strided 3×3 convolutions and uses the host CPU to execute layers that are not defined for the FPGA. Further, the Winograd

transform[10], a key optimization in this work, is not applicable to all types of convolutions. For example, pointwise (1×1) convolutions cannot benefit from this transformation. Our work is an end-to-end (i.e., entirety of the model is executed on FPGA) deployment and utilizes loop optimizations that apply to any CNN layer since we implement direct convolutions. In addition, DiCecco et al. uses 32-bit floating point precision for their accelerator, which is the same as this work.

Hadjis et al. [27] uses an HDL abstraction (Spatial) that allows the authors to deploy on both Xilinx and Intel FPGAs, which provides an edge on portability over Intel OpenCL tools. Further, folding batch normalizations into weights is not an optimization that TVM performs. A key differentiation is that Hadjis et al. allocates DSPs proportionally to the number of multiply-accumulate operations associated with a layer. Our work currently requires allocating of compute resources by controlling tiling/unrolling factors. Hadjis et al. uses 32-bit fixed point (Q10.22) representation, which differs from the 32-bit floating-point precision used in this work.

Sharma et al.'s DNNWeaver [55] is an automated design flow that generates Verilog for a given CNN and FPGA. In contrast to this work and the two aforementioned works, neither HLS or HDL abstractions are used. Rather, accelerators are generated from hand-optimized RTL templates. This results in predictable resource usages and achieves high performance, but requires the maintenance of more than 10000 lines of template code, which is not required for this work. DNNWeaver also performs a heuristics-based search for determining allocation of compute resources and operation scheduling to maximize performance. Sharma et al. uses 16-bit fixed point (Q3.13) representation, different from the 32-bit floating-point precision used in this work. Given the lower bit precision and DNNWeaver's highly-optimized hardware library, it is expected to deliver higher performance.

6.6.2 Performance

DiCecco et al. reports the performance of single-strided 3×3 convolutions in AlexNet (batch size of 64), VGG A (32), Overfeat (64), and GoogLeNet (64) on a Xilinx Virtex 7 FPGA. Since this work does not include the deployment of these networks, it is not possible to make a direct comparison in terms of FPS or latency. The closest comparison that can be made is to measure the GFLOPS of single-strided convolutions in ResNet-34 kernels. We summarize this comparison in Table 6.17.

The geometric mean of 3×3 convolution throughputs on the four networks used in [18] is reported to be 50 effective GFLOPS. Effective GFLOPS is calculated using the number of operations in the network assuming direct convolutions. However, the actual GFLOPS is lower because the Winograd transform

[10]The Winograd transformation trades computation complexity at runtime at the expense of increased storage footprint. It reduces the number of multiplications in 3×3 convolutions by a factor of $2.25\times$ [42].

reduces the number of operations that the accelerator must compute. Further, DiCecco et al. uses input batching as a source of parallelism. This work focuses on parallelism in a live deployment scenario where it is assumed that batching cannot be used. Adjustments are not made for either factors in this comparison since the Winograd and batching are still valid optimizations, although not ones we choose to implement due to their limitations. For ResNet-34 1-stride 3×3 convolutions we measure an average of 70.4 GFLOPS, which is a speed up of 1.41× compared to their performance. However, this evaluation carries the disclaimer that there is a 5-year gap in both silicon and compiler technology.

	DiCecco et al. [18]	This work
Workload	Geomean of 3×3 convolutions in AlexNet, VGG A, Overfeat, and GoogleNet.	3×3 convs in ResNet-34.
Batch size	32 (VGG A), 64 (Others)	1
Platform	Virtex 7 XC7VX690T-2	Stratix 10 SX
Total DSPs	3600	5760
Precision	32b float	32b float
DSP usage	36.3%	16%[11]
f_{max} (MHz)	200	125
GFLOPS	50	70.4

Table 6.17: Comparison of single-strided 3×3 convolution throughput between Caffeinated FPGAs and this work

Hadjis et al. benchmarks LeNet, TF CIFAR, Djinn-ASR, and ResNet-50 performance on the Xilinx UltraScale+ VU9P FPGA. Since we also deploy LeNet, it is possible to make a direct comparison in latency. They report a latency of 0.656 ms per image with a batch size of 1, while we measure 0.203 ms on the Stratix 10 SX, which is a speedup of 3.23×. However, they report that LeNet has 2.29M FP operations whereas we calculate 389K FP operations. This suggests that there is a difference in their implementation or calculations. Assuming that LeNet is 2.2M FP ops, they achieve 3.49 GFLOPS with LeNet, and if 389K FP ops is assumed, 0.59 GFLOPS. Our LeNet accelerator operates at 1.91 GFLOPS assuming the model is 389K FP ops.

Further, the authors report a 216 ms latency for ResNet-50 (7.8G FP ops) or 36.1 GFLOPS. A direct comparison is not possible in this case since we do not evaluate this network. However, we can use the GFLOPS of our ResNet-34 (7.36 GFLOPS) implementation. We measure 29.8 GFLOPS on ResNet-34, which is 17.5% slower than the GFLOPS they record for ResNet-50.

The performance gap can be attributed by the difference in DSP utilization. There are a few reasons why their infrastructure scales DSP utilization better than this work. Using a hardware IR such as Spatial allows them to explicitly control memory hierarchies and have better control of the resulting

[11] Includes DSPs used in other layers (2-stride 3×3 convolution, 7×7 convolution, etc.) in ResNet.

	Hadjis et al. [27]	This work	Hadjis et al. [27]	This work
Workload	LeNet	LeNet	ResNet-50	ResNet-34
Batch size	1	1	1	1
Platform	Xilinx UltraScale+ VU9P	Stratix 10 SX	Xilinx UltraScale+ VU9P	Stratix 10 SX
Total DSPs	6840	5760	6840	5760
Precision	32b fixed	32b float	32b fixed	32b float
DSP usage	26.7%	5%	87.8%	16%
f_{max} (MHz)	125	218	125	125
Latency (ms)	0.656	0.203	216	217
GFLOPS	3.49	1.91	36.1	29.8

Table 6.18: Comparison of LeNet and ResNet performance between TensorFlow to Cloud FPGAs and this work

design. This allows them to make the following optimizations: (1) removing unused IPs from Amazon's F1 shell, resulting in 8% and 19% reduction in LUT and BRAM usage, (2) assigning large blocks of data to Xilinx UltraRAMs, allowing them to use on-chip memory systems larger than SRAM, and (3) improved banking support, which is a 2× reduction in LUT. In contrast, we cannot implement any of these optimizations without modifying the behavior of the HLS compiler, which we treat as a black box. Therefore, Hadjis et. al can use 6006 DSP blocks for their ResNet-50 accelerator, while we can use 922 DSP blocks, a gap of 6.5×. This comparison is summarized in Table 6.18.

Sharma et al. benchmarks 8 different CNNs, including LeNet and AlexNet. However, the authors only do a comparison against CPU and GPU performance in [55], and so it is not possible to make a direct comparison. Therefore, we compare speedups over CPU with their reported speedups for LeNet. They report a 12× improvement over a 4-core Xeon-E3, while we observe a 2.47× improvement over a 56-core Xeon-8280. However, it is difficult to make a comparison here since they used Caffe, while we used Keras/TensorFlow.

Further, Venieris et al. [64] included DNNWeaver in their survey paper and reported that DNNWeaver performed at 184.33 GFLOPS for AlexNet on the Arria 10 GX115. A direct comparison is not possible since we do not evaluate this network. Thus, we compare their AlexNet (1.33G FP ops) performance with our MobileNet (1.11G FP ops) performance. We measure 20.0 GFLOPS on the Arria 10 with MobileNetV1, making their AlexNet accelerator faster by 9.22×.

The use of hand-designed RTL templates makes for highly-efficient and predictable designs. This, in addition to the design space exploration performed by DNNWeaver, allows it to achieve near full DSP utilization at 94.86% (6.8× compared to this work) for LeNet and 88.54% for AlexNet (2.68×). Using 16-bit precision arithmetic allows use of the 18×18 DSP reduced-precision mode on Intel FPGAs, where 2 fixed-point operations can be done per DSP block as opposed to 1 in 27×27. In addition, this reduces

	Sharma et al. [55]	This work	Sharma et al. [64]	This work
Workload	LeNet	LeNet	AlexNet	MobileNet
Batch size	1	1	1	1
Platform	Arria 10 GX	Arria 10 GX	Arria 10 GX	Arria 10 GX
Total DSPs	1518	1518	1518	1518
Precision	16b fixed	32b float	16b fixed	32b float
DSP usage	94.86%	14%	88.54%	33%
f_{max} (MHz)	200	217	200	181
v. CPU	12× Xeon-E3	2.47× Xeon-8280	4.2× Xeon-E3	0.83× Xeon-8280
GFLOPS	Unknown	1.0	184.33	20.0

Table 6.19: Comparison of LeNet and AlexNet/MobileNet performance between DNNWeaver and this work

the memory footprint of weights and activations, which allows for more data to fit in on-chip memory. We summarize this comparison in Table 6.19.

6.7 Summary

We summarize the evaluation into the following points:

- We demonstrate the deployment of LeNet, MobileNet, and ResNet CNNs from a high-level model description on three different Intel FPGAs, the Intel Arria 10, Stratix 10 MX, and Stratix 10 SX.
- We apply various compiler optimizations and OpenCL kernel optimizations to implement activation-pipelined kernels in LeNet. For MobileNet and ResNet, we use optimized parameterized kernels to implement efficient 1×1 and 3×3 convolutions. We demonstrate significant improvements over the naive implementation generated by TVM in all three cases.
- For LeNet, we achieve performance that surpasses highly-optimized CPU and GPU implementations with the LeNet-5 network on the Stratix 10. We achieve performance that is equivalent to 1-4 CPU threads for MobileNet and ResNet.
- Compared to DiCecco et al. [18], our accelerator is 1.14× faster in 3×3 convolutions. Compared to Hadjis et al. [27], our accelerator is 3.23× faster in LeNet single-image latency and only 8% slower in ResNet by GFLOPS. Compared to Sharma et al. [55], our accelerator is 9.2× slower in MobileNetV1 by GFLOPS compared to their AlexNet GFLOPS.

Chapter 7

Related Work

The acceleration of neural networks and AI workloads on reconfigurable hardware has been an active area of research. In the last five years, there has been over 329 published works in major FPGA conferences related to this subject (Figure 7.1). Therefore, we can only highlight salient work in the area.

Existing work in FPGA acceleration of DNNs can be categorized into three broad categories: (1) exploring specialized accelerator architectures or exploiting algorithmic optimizations, (2) design space exploration to automatically optimize specialized architectures for user-defined networks, and (3) automated design flows to ease complexities in end-to-end network deployment. We describe some prominent existing work and their contributions in implementing DNN accelerators on FPGAs in Table 7.1. Since our work is most related with the third category, this section describes prominent existing work in this topic.

Some of the early work can be categorized as *statically mapped* DNN accelerators that are fixed for a specific workload (such as a model or an operation) or a specific FPGA target. This includes Zhang et al.'s work [67], the work of Qiao et al. [52], and the work of DiCecco et al. [18], which we discuss in detail in Section 6.6.

More recent work in the literature often exhibits a higher degree of flexibility through the implementation of a design flow that maps a deep learning model from a high-level description to a fixed architecture or one that is dynamically generated for the model specifically. These can be considered *dynamically mapped* DNN accelerators, and include Caffeine [66], DeepBurning [65], DNNWeaver [55] (see Section 6.6), fpgaConvNet [63], Intel DLA [3], the work of Hadjis et al. [27] (see Section 6.6), and HPIPE [28].

Caffeine is an end-to-end deployment framework by Zhang et al. [66] Using a roofline model, the

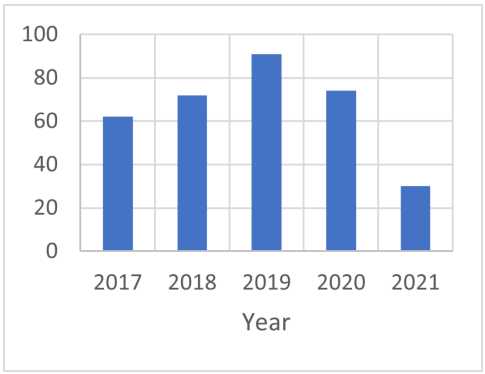

Figure 7.1: Number of publications with titles containing CNN, DNN, or neural networks in FPGA, FPL, and FCCM conferences

Table 7.1: Summary of Related Work

	Author/Project	Year	Framework	Contribution
[18]	DiCecco	2016	Caffe	3×3 Winograd conv for Caffe
[66]	Zhang/Caffeine	2016	Caffe	Caffe to FPGA bitstream; systolic array
[65]	Y. Wang/DeepBurning	2016	Caffe	RTL template-driven engine/compiler
[55]	Sharma/DNNWeaver	2016	Caffe	(DNN,FPGA) mapping from Caffe, novel ISA
[63] [62]	Venieris/fpgaConvNet	2016	Caffe/Torch	CNN as SDF, global optimizer DSE
[3]	Abdelfattah/Intel DLA	2018	Caffe/Tensorflow	High-performance overlay and graph compiler
[27]	Hadjis	2019	Tensorflow	Toolflow for TF model to cloud FPGA
[28]	Hall/HPIPE	2020	Tensorflow	Sparse CNN architecture and compiler flow

authors extend deployment to fully-connected, pooling, and activation layers. Caffeine is a CNN engine that provides an automated flow with two components: software-definable parameters include things such as the convolution filter size and stride and can be changed without reconfiguration, and hardware-definable parameters such as BRAM size and kernel size. As the authors use Vivado HLS templates, Caffeine can be generated for any Xilinx device. Similar to our deployment with MobileNet and ResNet, layers are computed one by one and input feature maps and weights are fetched in tiles. On the other hand, the Caffeine architecture resembles a systolic array that scales better to larger FPGAs.

fpgaConvNet [63] is a design flow that models CNNs as synchronous dataflows (SDF), where nodes are computations and edges are the data streams. The authors define three types of transformations: graph partitioning, coarse, and fine-grain folding. Graph partitioning is splitting components of the graph into multiple subgraphs that are synthesized into separate bitstreams, aiming to mitigate the reconfiguration overhead through processing multiple inputs in a pipeline. Coarse-grain folding involves the number of convolution units which effectively unrolls independent operations and fine-grain folding involves unrolling the reduction, ranging from time-multiplexing a single MACC unit filter width$\times$height times to computing the dot product in one cycle. In comparison, we do not parameterize fine-grain folding but always opt to unroll the filter dimensions fully. The authors use simulated annealing to maximize throughput [63] or minimize latency [62] constrained by resources, both of which are modeled analytically. Vivado HLS is used map to different Xilinx FPGAs.

DeepBurning [65] is also a Caffe-oriented design flow that maps DNN layers to building blocks in the NN component library, which are written in RTL. The authors employ two types of "folding" to fit models: spatial folding, i.e., mapping layers to different building blocks, and temporal folding, i.e., using those building blocks at different times. Parameterized kernels in this work can be considered analogous to temporal folding in DeepBurning.

Caffeine and fpgaConvNet rely on Vivado HLS templates to implement network operations while DeepBurning and DNNWeaver requires the maintenance of an RTL component library. In contrast, adding a new operation using this approach involves a high-level description in TVM compute functions and optimizing the schedule once. In addition, these four works are Caffe-oriented design flows, while we use TVM to take advantage of its support for a variety of frontends.

Intel DLA [3] differs from other approaches mentioned in the chapter. Rather than a design flow, DLA is an FPGA overlay written in OpenCL and is a 1D systolic array of dot product engines mapped by a proprietary graph compiler. Parallelism is achieved by vectorizing in four dimensions, input/output channel/filter and input feature map height/width, similar to other works. However, the authors employ further optimizations to increase utilization, such as mapping pooling and FC layers to convolution PEs,

and mapping 1×1 convolutions PEs that are optimized/unrolled for 3×3 convolution PEs. These optimizations result in state-of-the-art GoogLeNet performance. The addition of a new operator, such as a new activation kernel, requires the addition of a new functional unit added to the crossbar. It is not clear how a user provides a new functional unit or kernel to DLA.

HPIPE [28] is a FPGA sparse network acceleration architecture and graph compiler. Parameters for parallelism are selected algorithmically by the compiler that balances throughput of all layers while maximizing DSP utilization. The defining features of HPIPE are that it is a layer-pipelined dataflow architecture, capable of zero-skipping (see related thesis [29] for details), and stashes all weights on-chip. This limits execution to models with parameters that can fit into on-chip memory. HPIPE alleviates this limitation with support for pruned networks, and achieves state-of-the-art performance on a sparse ResNet-50. In contrast, we make use of external memory and are not limited to models with parameters that can fit on-chip.

Chapter 8

Conclusion and Future Work

This thesis explores the viability of compiler-generated OpenCL kernels for CNN acceleration on FPGAs. This approach benefits from being able to deploy different neural network operations without maintaining a hardware component library. We identify the key issues with naively generated kernels: parallel hardware is not created, suffer from serial execution, and underutilize memory bandwidth.

We propose two modes of execution, pipelined and folded execution. Associated with these two modes, we propose using a set of optimizations for the kernels and the host runtime to alleviate these three issues. Pipelined kernels are desirable when the input CNN has a relatively small number of parameters and intermediate activations that can fit in on-chip buffers. We use CL channels, autorun kernels, and concurrent execution to facilitate layer-pipelined execution. For larger models that necessitate time-multiplexing of kernels, parameterized kernels are necessary to allow the generated design to fit on the FPGA. However, re-using kernels for different layers in the network prevents layer-pipelining. Thus, we use loop tiling for more unrolling, effectively creating more parallelism.

LeNet is used to demonstrate a deployment using pipelined execution. For LeNet, we show that combination of these optimizations can improve baseline kernel performance up to 10.34×. This is 4.57× faster than Keras/TF-CPU (4 threads), and 3.07× faster than Keras/TF-GPU. For MobileNetV1, we show that the baseline kernel performance can be improved up to 183.8×. This is 40.3% faster than Keras/TF-CPU.

Further, MobileNetV1 and ResNet-18/34 are used to demonstrate deployments using folded execution. For ResNet-18 and 34 respectively, we show that the baseline kernel performance can be improved up to 842× and 1150×. ResNet-18 suffered a 57% to 76% slowdown compared to Keras/TF-CPU (112 threads), and is slightly faster than 1 TVM CPU thread. ResNet-34 suffered a 63% slowdown compared

to Keras/TF-CPU, and is comparable to 2 TVM CPU threads for the S10MX and 4 threads for the S10SX.

Overall, the results show that compiler-generated OpenCL produce viable kernels that perform competitively against optimized CPU and GPU for small CNNs. Our flow is particularly successful with pipelined execution. Therefore, this deployment approach may be attractive for deploying models that can be pipelined, i.e., networks with parameters and feature maps that fit in on-chip memories. Further, pre-production environments benefit from this approach since the cost of exploration is low. However, more work on optimization is needed to achieve competitive performance to the state-of-the-art, specifically in deploying larger CNNs.

8.1 Future Work

This work can be expanded in multiple directions. Described below are several directions that we believe can have an impact on this work.

- Quantized networks. Reducing bit precision for weight/activation representation can reduce arithmetic complexity (i.e., pack more operations per DSP) and memory footprint from floating point to integer or binary operations. This can lead to increased unrolling/tiling.

- Sparse networks. Supporting sparse workloads is useful when deploying pruned networks or when the workload is inherently sparse, such as graph neural networks. TVM has support for sparse matrix operations [51], but full deployment of sparse networks remains an on-going effort.

- Design space exploration. The parameter space for mapping CNNs into kernels is large, both in selecting which layers to parameterize and associated tiling sizes. The performance of this work is likely to benefit from DSE optimization since our design suffers from imbalanced performance across layers.

- Other types of DNNs. This work focuses on CNNs, and one direction would be to expand framework capability to recurrent neural networks (RNNs) and transformers. LSTM-RNNs are memory-bound [3] in comparison to CNNs and present different challenges.

- Multi-device execution for partitioned graphs. Our results demonstrate good performance with layer-pipelined execution. However, we are limited to the size of the network. Partitioning execution on multiple devices may alleviate this limitation.

Bibliography

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[3] Mohamed S. Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O'Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C. Ling, and Gordon R. Chiu. DLA: compiler and FPGA overlay for neural network inference acceleration. In *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*, pages 411–418. IEEE Computer Society, 2018.

[4] Peter J Ashenden. *The designer's guide to VHDL*. Morgan Kaufmann, 2010.

[5] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. An OpenCL™ deep learning accelerator on Arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 55–64, New York, NY, USA, 2017. Association for Computing Machinery.

[6] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[7] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Trans. Reconfigurable Technol. Syst.*, 11(3), December 2018.

[8] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, page 33–36, New York, NY, USA, 2011. Association for Computing Machinery.

[9] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *CoRR*, abs/1605.07678, 2016.

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[12] Y. Chen, T. Yang, J. Emer, and V. Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[13] S. Chung and T. S. Abdelrahman. Optimizing OpenCL kernels and runtime for DNN inference on FPGAs. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 151–154, 2020.

[14] Apache TVM Committee. Apache TVM, 2021. `https://tvm.apache.org/`.

[15] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[16] Jason Cong and Bingjun Xiao. Minimizing Computation in Convolutional Neural Networks. In Stefan Wermter, Cornelius Weber, Włodzisław Duch, Timo Honkela, Petia Koprinkova-Hristova, Sven Magg, Günther Palm, and Alessandro E. P. Villa, editors, *Artificial Neural Networks and Machine Learning – ICANN 2014*, pages 281–290, Cham, 2014. Springer International Publishing.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[18] Roberto DiCecco, Griffin Lacey, Jasmina Vasiljevic, Paul Chow, Graham Taylor, and Shawki Areibi. Caffeinated FPGAs: FPGA framework for convolutional neural networks. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 265–268, 2016.

[19] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. CNP: An FPGA-based processor for convolutional networks. In *2009 International Conference on Field Programmable Logic and Applications*, pages 32–37, 2009.

[20] The Apache Software Foundation. Apache TVM Committee, 2021. `https://projects.apache.org/committee.html?tvm`.

[21] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale DNN processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.

[22] L. Gong, C. Wang, X. Li, H. Chen, and X. Zhou. MALOC: a fully pipelined FPGA accelerator for convolutional neural networks with all layers mapped on chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2601–2612, 2018.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[24] A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, 2013.

[25] Khronos Group. The OpenCL Specification, Version 1.0, 2009. `https://www.khronos.org/registry/OpenCL/specs/opencl-1.0.pdf`.

[26] Khronos Group. OpenCL Details, 2012. https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf.

[27] Stefan Hadjis and Kunle Olukotun. Tensorflow to cloud FPGAs: Tradeoffs for accelerating deep neural networks. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 360–366, 2019.

[28] Mathew Hall and Vaughn Betz. HPIPE: Heterogeneous layer-pipelined and sparse-aware CNN inference for FPGAs. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA'20, page 320, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Mathew Kent Hall and Vaughn Betz. Architecture and automation for efficient convolutional neural network acceleration on field programmable gate arrays. Master's thesis, University of Toronto, 2020.

[30] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, 2015.

[31] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[32] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications, 2017.

[33] Intel. Enabling Impactful DSP Designs on FPGAs with Hardened Floating-Point Implementation, 2020. https://www.intel.ca/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01227-enabling-dsp-designs-on-fpgas-with-hardened-floating-point.pdf.

[34] Intel. Intel Arria 10 Core Fabric and General Purpose I/Os Handbook, 2020. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_handbook.pdf.

[35] Intel. Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide, 2020. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/archives/aocl-best-practices-guide-19-1.pdf.

[36] Intel. Intel FPGA SDK for OpenCL Pro Edition: Programming Guide, 2020. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/archives/aocl_programming_guide-19-1.pdf.

[37] Intel. Intel Stratix 10 NX FPGAs, 2021. https://www.intel.ca/content/www/ca/en/products/programmable/fpga/stratix-10/nx.html.

[38] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In Kien A. Hua, Yong Rui, Ralf Steinmetz, Alan Hanjalic, Apostol Natsev, and Wenwu Zhu, editors, *Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, November 03 - 07, 2014*, pages 675–678. ACM, 2014.

[39] Keras. Keras Applications, 2021. https://keras.io/api/applications/.

[40] UNC Chapel Hill Vision Lab. ImageNet Large Scale Visual Recognition Challenge 2015, 2015. http://image-net.org/challenges/LSVRC/2015/.

[41] Intel Labs. IL Academic Compute Environment, 2021. https://wiki.intel-research.net/.

[42] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 4013–4021. IEEE Computer Society, 2016.

[43] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[44] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[45] Fei-Fei Li, Ranjay Krishna, and Danfei Xu et al. CS231n Convolutional Neural Networks for Visual Recognition. https://cs231n.github.io/.

[46] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[47] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS 2019*, 2019.

[49] M. Pelcat, C. Bourrasset, L. Maggiani, and F. Berry. Design productivity of a high level synthesis compiler versus hdl. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 140–147, 2016.

[50] Apache TVM Project. TVM v0.8.dev0 Documentation, Design and Architecture, 2021. `https://tvm.apache.org/docs/dev/index.html#example-compilation-flow`.

[51] Apache TVM Project. TVM v0.8.dev0 Documentation, tvm.relay.nn, 2021. `https://tvm.apache.org/docs/api/python/relay/nn.html`.

[52] Yuran Qiao, Junzhong Shen, Tao Xiao, Qianming Yang, Mei Wen, and Chunyuan Zhang. FPGA-accelerated deep convolutional neural networks for high throughput and energy efficiency. *Concurrency and Computation: Practice and Experience*, 29(20), 2017.

[53] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new IR for machine learning frameworks. *CoRR*, abs/1810.00952, 2018.

[54] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[55] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to FPGAs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[56] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[57] Stuart Sutherland, Simon Davidmann, and Peter Flake. *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Springer Science & Business Media, 2006.

[58] Shan Tang. AI Chip (ICs and IPs) . `https://github.com/basicmi/AI-Chip`.

[59] The AlphaFold Team. AlphaFold: a solution to a 50-year-old grand challenge in biology, Nov 2020. `https://deepmind.com/blog/article/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology`.

[60] Donald Thomas and Philip Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008.

[61] Neil C Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F Manso. The computational limits of deep learning. *arXiv preprint arXiv:2007.05558*, 2020.

[62] S. I. Venieris and C. S. Bouganis. Latency-Driven Design for FPGA-based Convolutional Neural Networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2017.

[63] Stylianos I. Venieris and Christos-Savvas Bouganis. fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47. Institute of Electrical and Electronics Engineers (IEEE), May 2016.

[64] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions. *ACM Comput. Surv.*, 51(3), June 2018.

[65] Ying Wang, Jie Xu, Yinhe Han, Huawei Li, and Xiaowei Li. Deepburning: Automatic generation of FPGA-based learning accelerators for the neural network family. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 110:1–110:6, New York, NY, USA, 2016. ACM.

[66] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD '16, pages 12:1–12:8, New York, NY, USA, 2016. ACM.

[67] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM.

[68] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association, November 2020.

[69] Hamid Reza Zohouri. High Performance Computing with FPGAs and OpenCL. *CoRR*, abs/1810.09773, 2018.

Appendix A

FPGA Buffer Transfer Speeds

Figure A.1 shows measured buffer read/write/copy (a buffer copy is equivalent to a read and write) throughput from the host to the S10SX and S10MX FPGAs over PCIe for different payload sizes (shown with logarithmic scale). The figure shows that the S10MX is substantially slower for all payload sizes, especially with small payloads (smaller than 32 KB). Even for large payloads, there is close to a 2× gap in transfer speeds.

This has implications for input image transfer times. For example, a LeNet input image is 3136 B. 4 KB payloads for the S10MX are written at 9.71 MB/s whereas for the S10SX, at 372.2 MB/s. Therefore, image transfers are slower by a factor of 38.3×.

This can be attributed to a few factors, first that for the S10MX the PCIe link speed is limited to 8× and for the SX, 16×. Second, banking is supported for the S10SX whereas the S10MX is limited to one HBM PC unless the buffer is explicitly allocated to another PC (which we do not do). Third, it is an engineering sample and so the board on average is clocked at lower operating frequencies for both the kernel and the memory controller.

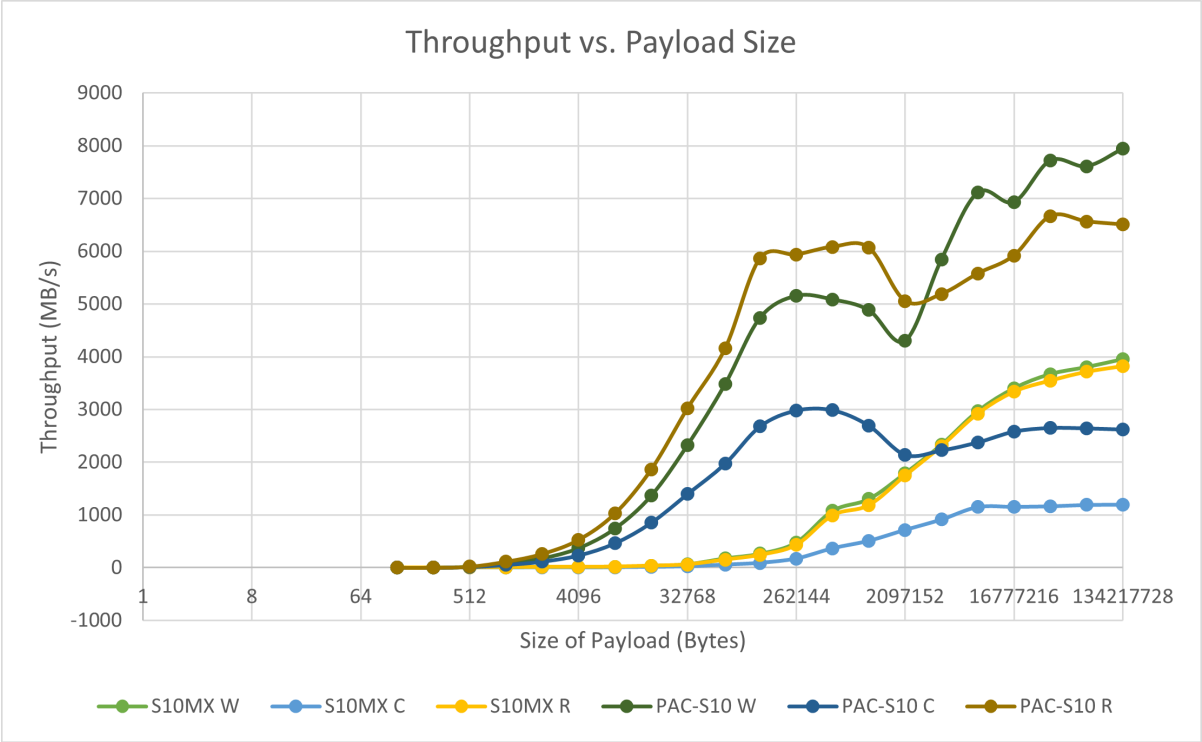

Figure A.1: Buffer read (R), write (W), copy (C) speeds from host to the S10SX and S10MX FPGAs. The size of payload is in logarithmic scale.