Improving the usability and scalability of FINN, a DNN compiler for FPGAs

by

U.M. Vimal Kumar

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday September 27, 2021 at 1:00 PM.

Student number:	5059941	
Project duration:	November 1, 2020 – August 31, 2021	
Thesis committee:	Dr. S. D. Cotofana,	TU Delft, supervisor
	Dr. ir. J. S. S. M. Wong ,	TU Delft
	Dr. ir. T. G. R. M. van Leuken ,	TU Delft
	Dr. L. Petrica,	Xilinx, Daily Supervisor

TU Delft

Abstract

FINN is a framework developed by Xilinx Research Labs that compiles Deep Neural Network software descriptions into fast and scalable dataflow architectures for inference acceleration on FPGAs. The dataflow architectures are network dependent, sized according to the user-defined throughput requirements, and constrained by available resources on the user-specified FPGA board. Synthesising large neural network designs with a high degree of configurability leads to large build times, spanning from hours to days, to build an entire network. Thus, the first objective of this thesis is to explore and propose a modified FINN accelerator construction methodology that can substantially reduce the build times. The main idea behind our proposal is to reduce the granularity of the architecture to reduce the size of synthesis jobs and to enable logic reuse within and across neural network layers. Using this method, up to 12× speedup in High-Level Synthesis times and up to 2× speedup in end-to-end build times of accelerator networks are achieved.

The second limitation that this work addresses relates to the performance scalability of FINN generated architectures. There are two modes of parallelism in FINN that currently provide performance scaling in convolution operations. The first factor, which modifies the number of Processing Elements (PEs), parallelises along the input channels of a convolutional layer and the second factor, that modifies the number of Single Instruction Multiple Data (SIMD) lanes present in each PE, parallelises along the number of output channels of the convolution. Computations are currently not parallelisable across the non-depth dimensions of images, i.e., the side containing pixels of images that faces the viewer. This limitation can restrict the achievable performance for networks that contain layers with large image dimensions and shallow depth dimension. The second part of this work leverages the fine-grained construction methodology to augment FINN performance scaling. The proposed approach introduces a generic FINN modification that enables pixel-level parallelism, i.e., multiple output pixels of a convolutional layer can be processed simultaneously by performing Multiple Matrix Vector (MMV) multiplications at the same time. Using this generic method, *MMV* number of pixels can be processed simultaneously, an *MMV* times throughput increase can be obtained at the cost of less than *MMV*× additional resources.

Acknowledgements

This is the final report of a 9-month long project, the longest individual project that I have undertaken. This thesis started in uncertain times and ran through its course in the midst of a pandemic. It is only with the help and support of people, online and in-person, that I was able to bring this project to a logical conclusion. This project was done in collaboration with Xilinx Research Labs.

First, I would like to thank Sorin Cotofana, my supervisor at TU Delft for guiding me towards this research group after knowing about my interest to work in the field of reconfigurable computing. This project would not have started without his support in finding a suitable arrangement when Covid threatened this collaboration. At Xilinx, the project was done under the supervision of Lucian Petrica. I would like to express deep gratitude to Lucian, who has been my primary contact point for this work, and guided me through every step of this project, right from scoping out the topic to organising the final report. From his words of advice and criticism, I learnt how to ask the right questions to solve problems and to use the big picture to make decisions on the details. Very importantly, I would like to thank him for being approachable and being only an e-mail away throughout the course of this project.

I want to thank my parents for their unconditional love, support and encouragement and my sister for always being a cheerleader. I also thank my friends who were on this journey along with me, for inspiring with their passion for work, for entertaining me with their company and for always lending a kind ear.

U.M. Vimal Kumar
Delft, August 2021

Contents

List of Figures

List of Tables

1 Introduction

Neural Networks (NN) have made significant gains in decision making accuracy in the last few decades that have enabled their use in various applications to classify images or make data predictions [28, 33, 36]. NNs can model complex relationships between inputs and outputs and use this model to make predictions on new input data. NNs are able to learn this relationship by observing data and iteratively modifying their model during a process known as *training*. A basic neural network can be modelled as a sequence of *fully connected* layers, each containing multiple neurons, wherein outputs of all neurons in one layer are connected with the inputs of all neurons in the subsequent layer. Each interconnection has an associated weight and each neuron has an activation function. Every neuron multiplies each of its inputs with its associated weight and accumulates these products. The output of each neuron is obtained by applying its activation function to this sum. The weights and coefficients of the activation function are the *parameters* of the neural network. During training, the neural network iteratively adjusts its parameters based on the training data. After training, the NN uses the trained parameters to make predictions on data through a process known as inference.

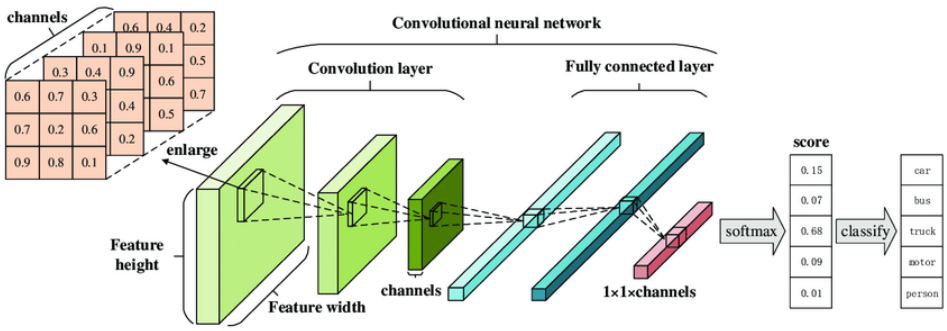


Figure 1.1: General Topology of Convolutional Neural Networks [15]

Convolutional Neural Networks (CNNs) are a type of NNs that are primarily used for image recognition. They are explained in detail in Section 2.2 and a short summary is provided here. CNNs contain convolutional and pooling layers in addition to the general fully connected layers. A general arrangement of layers in a CNN is shown in Figure 1.1. Convolutional layers dominate the total computations. They perform convolution by sliding a kernel or filter over the input image, also called Input Feature Map (IFMs), and perform a dot product between the kernel and the part of the image at each position. A single convolution of a filter on one position of the input image produces one output pixel of the Output Feature Map (OFM). After the kernel slides over the entire image, all output pixels of the OFM are obtained. The filters typically have small frontal dimensions, i.e., the side of the cuboid that faces the next cuboid in Figure 1.1, and a depth dimension equal to the number of channels (depth) in the input IFM. Convolution of a single filter over the entire IFM produces one channel of the OFM. Hence, the depth of the OFM, which is also the number of channels in the OFM, is equal to the number of filters that are convolved with the IFM. Pooling layers in CNNs reduce the frontal dimensions of the image, and serve to reduce the number of computations, and avoid overfitting. State of the art CNNs typically contain a large number of convolutional layers to identify complex features in images.

Hence, they are also called Deep Neural Networks (DNNs). As shown in Figure 1.1, feature maps get deeper, contain more channels as we progress along the CNN, while the number of pixels reduces progressively.

The complex decision making ability of NNs comes with the cost of large number of computations. State-of-the-art image classification networks require millions of floating point operations to classify a single image [18][29]. Recently, various computing paradigms such as using fixed-point arithmetic [23] and stochastic computing [21], have been explored to reduce computational and memory storage complexity [11] of DNNs. It has been shown that quantization, a method to reduce the precision of parameters and computations from floating to fixed point can significantly reduce the NN footprint with only a small cost on accuracy [30]. Such NNs where all computations are quantized are called Quantized Neural Networks (QNNs). Furthermore, time complexity of inference has also been diminished by computing on parallel hardware that can take advantage of the inherent parallelism [4].

Parallel hardware devices like GPUs and FPGAs can exploit the parallel nature of the inference process. FPGAs are well-suited for NN inference, especially with QNNs. Due to a smaller memory footprint, all network parameters of some QNNs can now fit entirely onto memory on the FPGA fabric. In addition to removing the memory bandwidth bottleneck, avoiding off-chip memory accesses also has a positive impact on energy consumption. Moreover, reduced precision arithmetic can be more specifically optimised and mapped efficiently onto FPGA resources. Efficient resource mapping in turn permits more computation to be allocated on-chip and pushes performance capabilities. Advances in FPGA technology and recent trends in NN research have made them well-suited for inference. Research has shown that in some applications FPGAs can offer better performance than GPUs, while being more energy efficient [26].

FINN [19] is a framework developed by Xilinx that generates FPGA-mappable dataflow-style QNN inference accelerators from software descriptions of QNNs. Users of FINN can provide the software description of the model, the required throughput and a specific FPGA board, and FINN can generate the FPGA architecture to meet these requirements. An overview of the end to end flow of the FINN framework is shown in Figure 1.2.

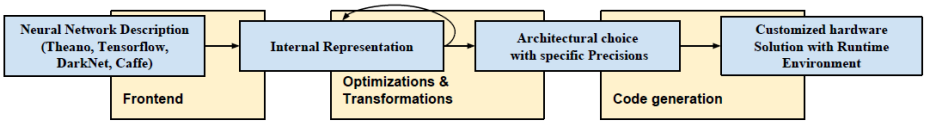


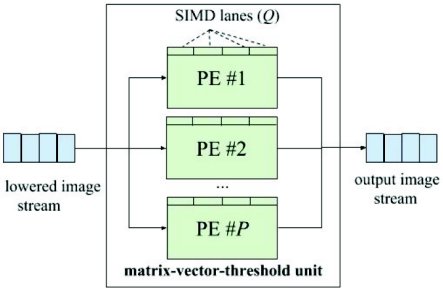
Figure 1.2: Overview of FINN framework [3]

The FINN frontend modifies NN descriptions from frameworks such as Tensorflow or Pytorch into a common internal representation. In the internal representation, an NN is represented as a directed acyclic graph, where each node represents a layer and the edges represent the dataflow direction. Initially, the internal representation is agnostic to the user-specified FPGA board. After a series of transformations, the internal representation contains complete descriptions and details of each layer specific to the particular FPGA model that the user requires. In the intermediate transformations, FINN updates the acyclic graph representation with C++ implementations of each node that encompasses their functional description as well as details necessary to get the required performance. The FINN backend first transforms these high-level descriptions to RTL descriptions through High Level Synthesis. Each node or layer of the neural network is synthesised independently of the others and the entire network is stitched together in subsequent steps. Then, the RTL descriptions are synthesised into a netlist and subsequently placed and routed on the FPGA target.

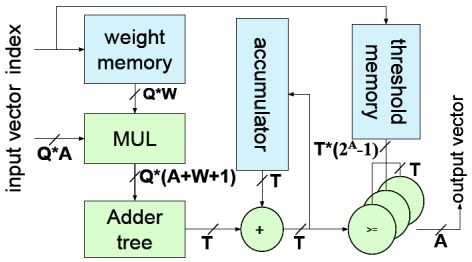
Inference accelerators generated by FINN are scaled to meet user defined performance requirements and have ultra-low classification latencies. Every layer is always active on the FPGA and the architecture is pipelined. So, each layer can begin computation as soon as the previous layer starts producing outputs. This also means that a new input image can be processed even before the previous input has exited the pipeline. The architecture thus mirrors the streaming nature of NN inference process. Each layer in an NN is individually sized by FINN to meet the user provided throughput requirements. The number of cycles that each layer takes to complete data processing for an image depends on the amount of FPGA resources FINN allocates to that layer for processing. In FINN-generated architectures, the throughput of a layer is improved by allocating more processing elements for its implementation. Computations of all layers are rate-balanced to maximise performance with the available resources. The need for rate-balancing can be illustrated with a simple example of an NN containing two layers. Consider that the first layer is sized such that it takes 100 cycles to finish computation for a single image and the second layer takes 1000 cycles. The overall network throughput would still be limited by the layer that has a latency of 1000 cycles. Hence, some of the parallel

processing elements allocated for the first layer would be wasted as they do not contribute to extra performance of the network, as the second layer would bottleneck the performance. Sizing of each layer in FINN is also referred to as *folding* in FINN terminology, and is an important transformational step that is performed on the internal representation.

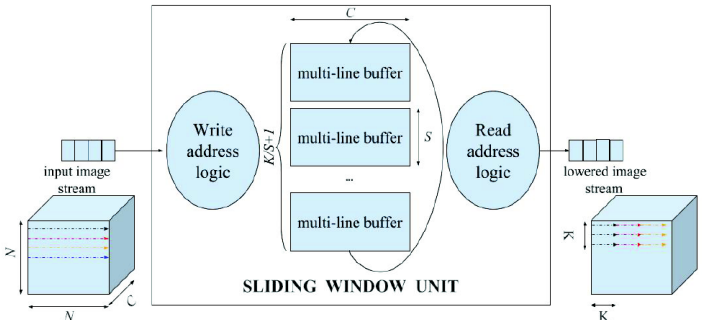
In this work, the term *FINN* is used to represent both the compiler and the architectures generated by the compiler.



(a) Matrix Vector Activate Unit - performs matrix multiplication operations



(b) Basic Processing Element - performs the basic Multiply Accumulate operation



(c) Sliding Window Unit - the structure that is used to *lower* convolutions to a general matrix multiply operation

Figure 1.3: Architectural implementation of relevant layers in FINN [3]

Architectures of the most common NN layers as implemented by FINN are shown in Figure 1.3. The Matrix Vector Activate Unit (MVAU) or Matrix Vector Threshold Unit (MVTU) (Figure 1.3a) is the primary computational unit in FINN. It implements matrix multiplication for both fully connected layers and convolutional layers. It performs multiply-accumulate (MAC) operations and applies the activation function, that can be implemented as a thresholding operation. To speed up matrix multiplication, the MVAUs have two methods of performance scaling that can exploit the parallel nature of the multiplication, namely, PE and SIMD parallelism. Firstly, PE parallelism refers to the number of Processing Elements (PEs) that can simultaneously perform MACs. Secondly, SIMD parallelism refers to the number of Single Instruction Multiple Data (SIMD) lanes that are present in each PE, that can perform SIMD multiplications simultaneously. Figure 1.3b shows the structure of each processing element. *W, A* and *T* refer to the bit-widths of weights, activations and accumulator outputs respectively. The Sliding Window Unit (SWU) (Figure 1.3c) in FINN is used in representations of convolutional layers and is used for lowering convolutions, i.e., to convert the input image into a suitable form, so that convolutions can be implemented as a matrix multiply operation. Hence, in FINN convolutions are represented as an SWU followed by an MVAU. In the FINN dataflow model, the IFM arrives pixel by pixel and is stored in the multi-line buffer of the SWU. The IFM arrives in raster order - first, all channels of a pixel, followed by all pixels in a row of the image and then subsequent rows. The buffers in the SWU store the incoming IFM and output the pixels in an order that is suitable for the MVAU. The SWU contains read and write control logic that provides the address of the buffer which has to be read from or written into. The MVAU and SWU modules in Figure 1.3 are the focus of this thesis.

Currently, in FINN, NN dataflow accelerators are constructed by assembling appropriately sized and self-contained compute engines representing each NN layer. Each layer contains its own computational logic and its parameters. FINN generates the C++ definition of each layer, invokes Vivado HLS to compile this

description into Verilog, and finally turns to Vivado to synthesise an FPGA bitstream from this description. The accelerator is thus a *coarse-grained* pipeline of such logic blocks. Each block is individually sized by FINN to meet user-specified performance requirements. Such a high degree of configurability translates to large synthesis and implementation times associated with building the FPGA configuration. FINN build times need to be diminished in order to speed up the design process and allow for faster architectural design space exploration.

Another limitation to FINN relates to the limited performance scalability of the DNN architectures it generates. In FINN, convolutions, which are the primary computation in DNN inference [6], are lowered to matrix multiplications [5]. They constitute the most significant portion of end to end computation cycles in CNNS. As described previously, convolutions in FINN are modelled as a SWU, that lowers convolutions to matrix multiplications, followed by an MVAU, that performs the matrix multiplication. Currently, FINN provides two modes of parallelism to speed up performance - PE parallelism and SIMD. In the context of convolution, PE parallelism translates to parallel computation across multiple channels of a *single* pixel in the OFM, and SIMD parallelism allows for parallel processing across multiple IFM channels for a single output pixel. However, there is no provision for parallelising computations of multiple output pixels simultaneously. Therefore, the achievable speed up is limited by the frontal dimensions of the output, that depend on the total number of pixels. This can pose a limitation to the performance scalability of initial DNN layers, because feature maps in initial DNN layers are typically characterised by large frontal dimensions and a shallow depth dimension, similar to input images. FINN is able to obtain maximum performance with the available resources by keeping the dataflow rate uniform for each layer. Hence, when the initial layers cannot exploit parallelisation methods that FINN currently provides due to their shallow channel dimensions, this limitation can curtail overall performance in some networks.

1.1. Research Questions and Approach

This work is centred around answering the following research questions, which are formulated in view of the discussion above as:

- How can the construction methodology of FINN architectures be modified to improve their build times? How large is the resource overhead induced by this approach and how can it be minimised?
- How can FINN-generated dataflow architectures be modified to introduce an additional level of performance scaling?

This thesis consists of two parts: the first part aims at implementing and evaluating a modified FINN workflow that improves its ease of use. The focus of the second part is to introduce an additional level of parallelism to FINN, to improve its scalability.

The initial objective of this thesis is to reduce the overall build times of FINN networks by implementing fine-grained MVAUs. It was observed that MVAU nodes that have smaller degrees of PE parallelism synthesise faster than MVAU nodes that have larger PE values. Hence, the idea is to replicate the same functionality of the existing large monolithic MVAU blocks with a fine-grained reconstruction containing smaller and repeatable PEs that can synthesise faster. Parameter storage is separated from computational logic used to implement the MVAUs, and computational logic is assembled from smaller processing elements that implement MAC operations. Fewer simple identical PEs can then be repeated within and across computational layers that implement the fully connected or convolutional layers. Effectively, this work modifies the part of the FINN framework that generates C++ code for the MVAUs such that it generates finer processing elements. Such a fine-grained structure that enables the reuse of identical components reduces the synthesis workload of Vivado HLS and Vivado, running in the FINN backend. Post-synthesis, these units can be assembled along with data movement infrastructure to reproduce the same functionality as the existing coarse pipeline. Using this approach, up to 50× speed up was obtained on HLS synthesis times of standalone computational nodes, 1.6× speed up was obtained on Vivado synthesis times and upto 12× speed up on total build times of standalone nodes.

The second objective of this thesis is to leverage this fine-grained construction methodology to provide an additional level of performance scaling in FINN designs, particularly in convolutions. Currently, the two methods of parallelism are able to speed up a single Matrix Vector multiply operation, which is able to produce just one output pixel in the current FINN dataflow method. This thesis provides an organisation method to perform Multiple Matrix Vector (MMV) multiplications and compute multiple output pixels of a feature

map simultaneously. This work modifies producer and consumer blocks of convolutional layers to be compatible with the modified dataflow and introduces a SWU block with new functionality. The value of MMV, i.e., the number of output pixels to be calculated parallelly can be used by FINN as an additional parameter that can be modified to meet throughput requirements when it is not able to meet this requirement with the existing PE and SIMD parallelism. With this generic approach, it is possible to compute *MMV* number of output pixels simultaneously and effectively reduce the number of computation cycles taken to classify an entire image by a factor of MMV.

The contributions of this thesis are as follows:

- Implement FINN-Finegrained - a generic modified construction methodology that enables reuse of its core computational components and evaluate the impact of this method on build time in various design stages and measure related overheads in area.
- Introduce a modification to the FINN dataflow that enables further performance scaling in FINN to overcome current throughput limitations.
- Introduce a generic RTL implementation of a sliding window unit which adds extra capabilities to handle multiple pixel data simultaneously. Further, it integrates multiple existing HLS implementations of the unit. It is also more resource efficient than the HLS implementations and can reduce memory usage by 25%.
- Introduce a resource efficient method of implementing MVAUs that are not time multiplexed and contain sparse weights.

1.2. Outline

The rest of this thesis is structured as follows. In Chapter 2, background about general neural network concepts is provided, need for hardware accelerators is motivated, some common strategies for efficient implementations of NN inference are described, end-to-end FINN flow is introduced and architectural implementation of the algorithm is detailed. In Chapter 3, fine-grain construction method is derived from the existing algorithm of the monolithic structure and modifications to reduce resource overheads from this method are provided. In Chapter 4, MMV methodology is incrementally derived and the design for a new sliding window unit that facilitates performance scaling is presented. In Chapter 5, fine-grained implementation is evaluated on a standalone node and impact on build times and resource utilisations are measured for varying number of processing elements. Some networks supported by FINN are also implemented and evaluated using the modified method, and the results are analysed based on the results on standalone nodes. In Chapter 6, the new SWU implementation is compared with the existing SWU in terms of resource improvements and the differences in scheduling efficiency are tested on a FINN network and reported. Then, effectiveness of the proposed MMV modification is evaluated.

2
Background

This chapter provides an introduction to the field of neural networks and describes their common implementations. It describes the problem associated with real-time implementations of neural networks in hardware and introduces quantization, a commonly used solution to solve practical problems associated with using them. Section 2.4 discusses why FPGAs are suitable for implementing neural networks and some common architectures. Section 2.6 introduces FINN, a framework for fast neural network inference on FPGAs and details its features and steps of flow. Section 2.5 outlines the backend steps in developing an implementation of a design on an FPGA. Finally, in Section 2.6.3, the two limitations in FINN that this thesis targets to improve are defined.

2.1. Neural Networks

Neural Networks (NNs) are a class of decision making algorithms that are structured based on a simplification of decision making neurons in a brain. NN algorithms are modeled as an interconnection of neurons which modify incoming signals based on their characteristic properties similar to how electrical signals are propagated in the human brain. Inputs to the network are obtained from datasets or measured in real time and propagated as signals through the NN architecture. The structure modifies the incoming signals to extract important properties to obtain the final output.

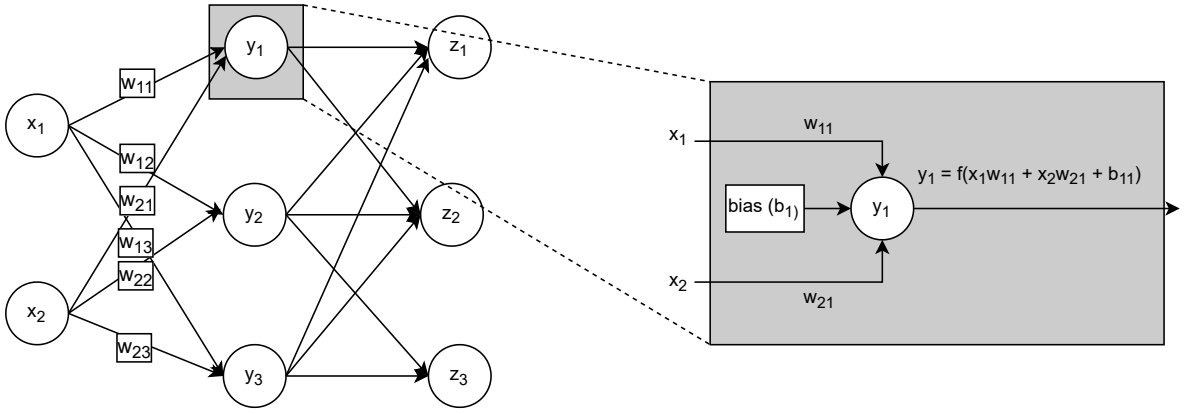


Figure 2.1: Basic Structure of a Neural Network

NNs are generally represented as an acyclic graph structure. Nodes in the graph correspond to neurons and are arranged in a layered fashion where outputs of one layer are inputs to the subsequent layer. Figure 2.1 shows a basic structure of a Multi-Layer Perceptron (MLP) NN, and a neuron, which represents the basic building block in a NN. An MLP is a neural network in which all nodes in a layer are connected to all nodes in the subsequent layer. A neuron first performs a linear operation that computes the sum of inputs from the previous layer multiplied with their respective fixed coefficients. Hence, computations are dominated by Multi-Accumulate (MAC) operations. An activation function is applied to the sum to introduce

non-linearity into the computation. The result of the activation function serves as the input to the subsequent layer. Each NN is characterised by the structural arrangement of its nodes, i.e., the number of nodes in each layer and the number of layers. Each node is characterised by its coefficients for the multiply accumulate (MAC) operations and the activation function that is applied. For a given NN, the structure is fixed, the number and type of computations are fixed, and exact coefficients of computation can be adapted for the specific application. Before the neural network can be used for a specific task, it learns the parameters of computation using a process called *training*. Training involves multiple iterations of forward pass and backpropagation through the structure to learn parameters by optimising a cost function. After training, neural networks can be used for *inference*, where it operates on new data to predict outputs with an accuracy defined by its classification accuracy. This work focusses on the inference process only since the current trend is to perform training in servers online and then deploy networks in real-time applications where only inference needs to be performed.

2.2. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a variation of the general NN structure (Figure 2.1) in which only a subset of nodes in a layer is connected to a node in the subsequent layer. They are specifically used in image processing applications due to their ability to capture features in an image with fewer parameters than an MLP would require.

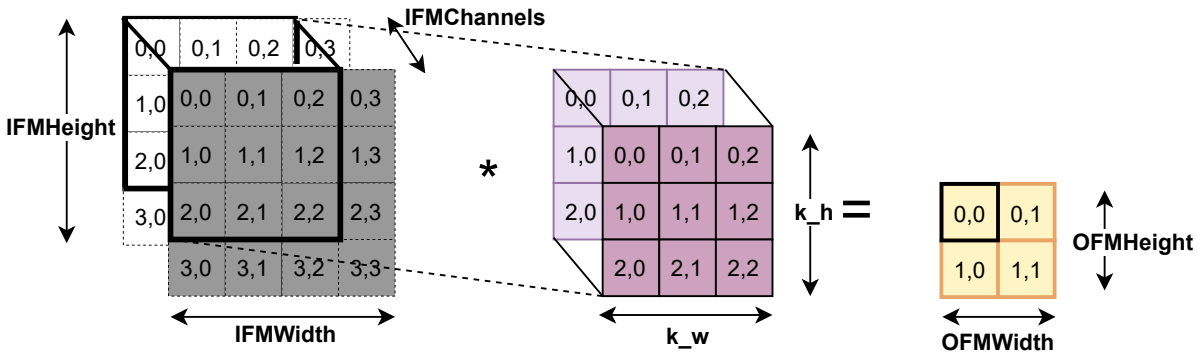


Figure 2.2: Convolution Operation

CNNs primarily contain convolutional layers which perform convolutions of ($k\_h \times k\_w \times IFMChannels$) sized kernels on input images, also referred to as Input Feature Maps (IFM). A sliding kernel that steps over stride, *S (S_h, S_w)*, pixels every time is convolved with the input image. Outputs from a single convolution correspond to one channel of a pixel in the output feature map (OFM). The output from a CNN layer consists of stacked outputs obtained from convolutions of multiple kernels on the input feature map. Different kernels are specialised to identify specific low level features. Kernel dimensions are typically small, ranging from 1 to 7 in most modern neural networks, so OFMs contain information about localised features of images. After multiple convolutional layers, the network is able to recognise high level features.

Additionally, CNNs contain Pooling layers to downsize image dimensions by subsampling. They serve to reduce the number of computations, to suppress noise in the network and to avoid overfitting. CNNs also contain fully connected layers at the end to aid in the final classification based on high level features returned by previous layers. Computational time in the forward pass of a CNN is dominated by convolutional layers. Modern CNNs typically have a large number of convolutional layers and are also referred to as *Deep Neural Networks* (DNNs). The number of computations (N) in a single convolutional layer depends on

- OFMWidth, OFMHeight : Horizontal and Vertical dimensions of the output feature map (OFM)
- IFMChannels : Depth of the input feature map
- OFMChannels : Depth of OFM (number of convolution kernels)
- k_h, k_w : Horizontal and Vertical dimensions of the convolution kernel

The OFMWidth and OFMHeight in turn depend on

- IFMWidth, IFMHeight : Horizontal and Vertical dimensions of the input feature map (IFM)
- S_h, S_w : Number of pixels that the kernel strides over in 1 step
- Pad_h, Pad_w : Number of padding pixels applied to IFM before convolution.

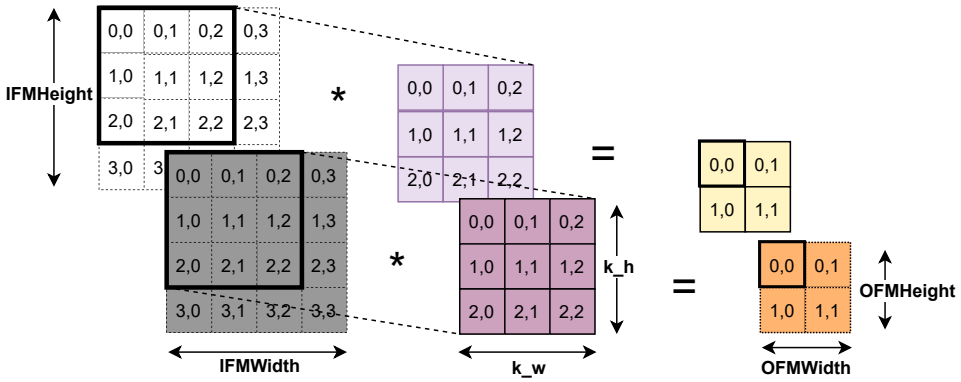
The relationship is given by

$$N = OFMWidth \times OFMHeight \times k\_w \times k\_h \times IFMChannels \times OFMChannels \tag{2.1}$$

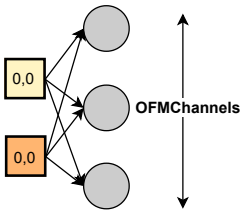
$$OFMWidth = \frac{IFMWidth - k\_w + Pad\_w + S\_w}{S\_w} \tag{2.2}$$

$$OFMHeight = \frac{IFMHeight - k\_h + Pad\_h + S\_h}{S\_h} \tag{2.3}$$

Depthwise Separable Convolutions Depthwise separable convolution is a light-weight convolution method present in state of the art DNNs such as Mobilenet [12] and QuartzNet [17]. It operates in two stages, depthwise convolution followed by pointwise convolutions, which when applied sequentially to an input image achieve the same effect as a normal convolution, with lesser computations.



(a) Depthwise Convolution



(b) Pointwise Convolution

Figure 2.3: Depthwise Separable Convolutions

In depthwise convolutions, IFMChannels kernels of dimension $(k\_h \times k\_w \times 1)$ are separately convolved with each channel of the input feature map. Each kernel is applied over a single channel of the IFM only. Number of operations in depthwise convolutions of a single image is given in Equation (2.8). This step does not modify the depth of the image. The subsequent pointwise convolution step has an impact on the number of output channels. A 1×1 kernel with a depth equal to $IFMChannels$ convolves over every pixel of the intermediate output to produce a single output channel of the OFM. Convolutions of $OFMChannels$ number of $1 \times 1 \times IFMChannels$ sized kernels produce the entire output feature map after depthwise separable convolution. The number of operations in pointwise convolution is given in Equation (2.9). The total number of operations is thus greatly reduced with this method while achieving the same effect as normal convolutions.

$$DepthwiseConvolutionOperations = OFMHeight \times OFMWidth \times k\_h \times k\_w \times IFMChannels \tag{2.4}$$

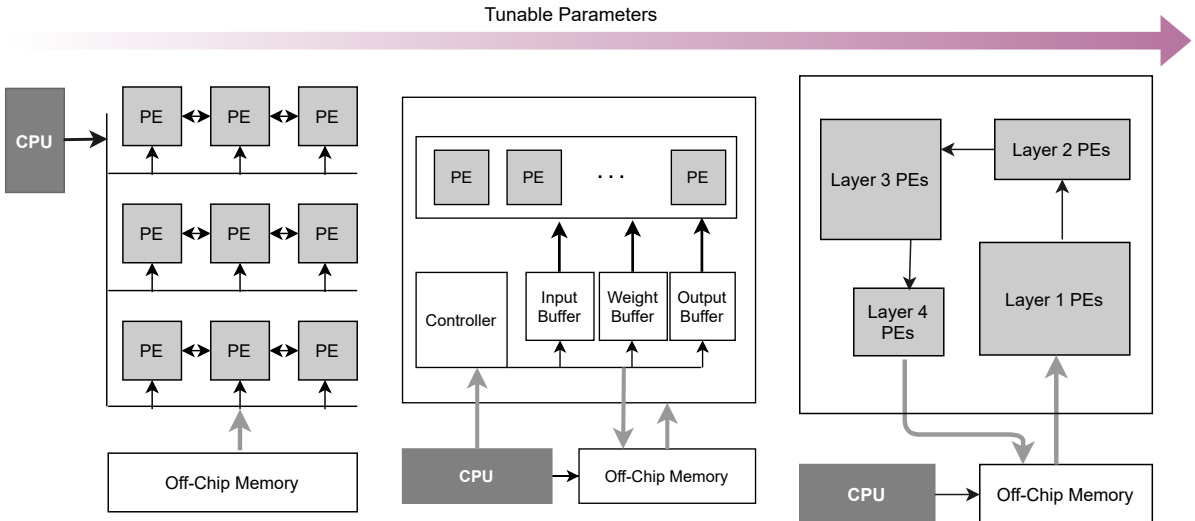
$$\text{PointwiseConvolutionOperations} = \text{OFMHeight} \times \text{OFMWidth} \times \frac{\text{OFMChannels}}{\text{PE}} \times \frac{\text{IFMChannels}}{\text{PE}} \tag{2.5}$$

Due to large numbers of computations, CNN inference has high time complexity and large energy requirements that are prohibitively large for use in embedded systems that have limited hardware capabilities. Hence, various implementation methods have been explored to reduce the memory and computational footprint of CNNs.

2.3. Quantized Neural Networks

Quantized neural networks (QNNs) are a type of NNs which perform computations on stored parameters at reduced precisions. The quantized parameters can be obtained either during training itself (Quantization Aware Training) [13] or by quantizing high-precision values obtained after training. The advantanges of quantized networks are two fold. Firstly, memory requirements drastically decrease when we move from floating point to few-bit representations. This makes it easier to store network parameters on resource constrained edge devices. Secondly, computations are greatly simplified for reduced precision networks. The primary MAC operation can be replaced with easier bitwise operations that significantly reduces complexity of the forward pass in exchange for a small accuracy reduction. Research has shown that considerable accuracy can be obtained even with extremely low precision (1-bit) quantised networks (Binarized Neural Networks) [7].

2.4. High Speed Neural Network Inference



(a) Systolic Array (b) Generic SIMD Architectures (c) Dataflow Architectures

Figure 2.4: FPGA NN Architectures

Deep neural networks require millions of operations to classify a single frame. For real time applications such as machine vision or cybersecurity, dedicated hardware that can speed up inference is required to meet the strict timing requirements. NN algorithms exhibhit multiple avenues of parallelism that can be exploited using high speed parallel processors. The sources of concurrency in the algorithm are [1]:

- batch level parallelism : computation for each image is completely independent of other images
- pipeline parallelism : the feedforward structure of CNNs means that the computations of successive data dependent layers can be pipelined. Subsequent layers can start processing and producing outputs immediately when they start receiving inputs from the prevous layer.
- inter-channel parallelism : calculation of each output channel of a feature map is independent of other output channels.

- multi-pixel parallelism : each output pixel of an output feature map can be calculated independent of the other pixels.
- intra-convolution parallelism : multiply operations within a MAC operation can be executed simultaneously.

Hence, inference can be greatly sped up using computer architectures that perform multiple computations simultaneously. Due to the inherently streaming nature of a NN forward pass, inference is suitable for implementation on FPGAs [26]. QNNs can also be efficiently implemented in FPGAs since processing elements can be specifically tailored for specific bit-widths. Further, current trends in Deep Neural Networks such as pruning introduce irregular parallelism in the network. Algorithm level optimisations that can take advantage of such specific low level features are easier to implement in FPGAs. Moreover, FPGAs offer higher performance per Watt [27] so they are suitable for such power hungry applications in power constrained embedded environments. Owing to these reasons many FPGA DNN architectures have come up in research recently [24]. This thesis specifically focusses on FPGA accelerators.

FPGA inference accelerators in literature are summarised as [2]:

- Systolic Arrays : The general design of systolic array architectures is shown in Figure 2.4a. Generally, it consists of a static collection of processing elements (PEs). The architecture is fixed regardless of the CNN topology implemented. Weights and intermediate results are transferred to and from off-chip memory. Since there is no data caching and the CPU controls data transfer through DMAs, memory bandwidth easily becomes the bottleneck in this type of architectures. The maximum supported kernel size k_m is a hyperparameter in the design and can lead to underutilisation of resources when kernel size $k < k_m$ [22].
- SIMD Style Accelerators : These architectures try to optimise the PE size by appropriately sizing the SIMD lines for every DNN. This is done by performing design space exploration of the CNN algorithm, where the optimal level of unrolling of the various parallelism parameters are identified. Here as well, a subset of layers is computed at a time, but, intermediate values are stored in buffers on chip rather than DRAMs. In [37], an efficient way of implementation is proposed by double buffering input and output buffers to be able to perform computation and communication simultaneously. Various optimisations such as loop tiling, to fit data onto on-chip buffers [8], are performed to reduce DRAM accesses. Using the roofline model [34], design space exploration is performed to find the combination of parameters that offers good performance for a given FPGA board.
- Dataflow Architectures: In this, all computations are data driven, all layers are present on the reconfigurabe fabric and the flow mirrors the actual data driven flow of DNNs. It is best described as a (MIMD) multiple instruction multiple data topology, multiple parts of the network can simulataneously act on data whenever it becomes available in that point of flow. Hence, this method can theoretically exploit all levels of parallelism inherent by CNN computations. All computations including Pooling and Batch Normalisation are optimised and performed in the fabric and each layer can be appropriately sized to meet throughput requirements. Naturally, the bottleneck is resource utilisation, since all layers are simultaneously active on the programmable fabric. Hence, by tuning the tradeoff between resource utilisation and latency, it is possible to obtain ultra-low latency using this approach.

Hence, FPGA NN inference accelerators differ in design about the degree of fine-tuning possible, how much control is still in CPU, location of network parameters, frequency of data transfer between on chip and off chip memory, and homogeinity of processing elements in each of the computational layers.

2.5. FPGA Acceleration

An Field Programmable Gate Array (FPGA) is a semiconductor device that is programmable on the field after manufacturing. It consists of an array of configurable logic blocks (CLBs) that communicate using programmable interconnect (Figure 2.5. Each logic block contains smaller components, primarily, look-up tables (LUTs) that implement combinational logic, and flip-flops (FFs), that can store the results from LUTs. Data is transferred into and out of the FPGA through I/O buffers. FPGAs map computation spatially onto these logic blocks. They typically contain large number of logic elements and are able to massively parallelise computation and communication. FPGAs are well suited for streaming applications that require high-speed data processing.

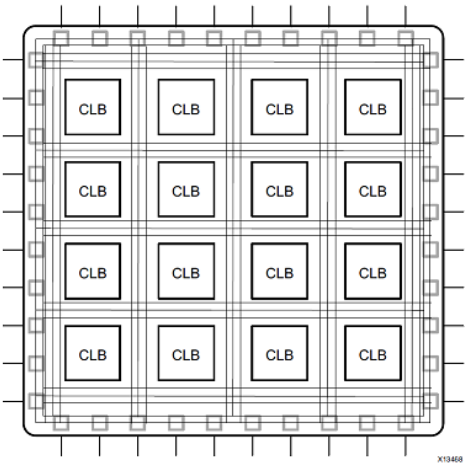


Figure 2.5: Basic FPGA Architecture [35]

2.5.1. FPGA Build Flow

Major steps in the FPGA design flow are described in this section. Synthesis and implementation are always part of the build flow. High level synthesis is a step that is performed before synthesis for parts of the design which are complex to express directly in Register Transfer Level (RTL) descriptions that are needed for synthesis and are instead described using a high-level language.

High Level Synthesis:
High-Level Synthesis (HLS) is a process which converts high level programming languages to RTL level hardware descriptions. Using HLS, development times can be significantly reduced as much of the implementation details are handled by the compilation software. Using HLS, it is easy to make generic code rapidly that aids in faster design space exploration. It is also possible to automate the pipelining process to obtain the required frequency of operation. However, there is little control over the type of hardware that is inferred by HLS, and there is a good possibility of generating inefficient hardware. The development time is saved at the expense of extra resources.

Synthesis:
In this step, RTL description of the design is transformed into a list of flip-flops, registers and multipliers which can map directly onto FPGA primitives. This step depends on the particular FPGA part that the design will be mapped on. Physical constraints that can restrict the placement of cells or pin configurations and timing constraints that define the expected clock frequency for the design can be applied during this phase.

Implementation:
Implementation process involves a sequence of steps that place the synthesised netlist onto the specific FPGA board and route them to meet the user-specified resource and timing constraints. The major steps in implementation are

- Link Design: In this step, components designed using the bottom-up synthesis flow are stitched together to form the unified total design.
- Opt Design : Now that information about the overall design has been obtained, Vivado is able to optimise the design through a series of steps like constant folding, sweeping all unnecessary components from the generated hardware design, and remapping logic to simpler components, wherever possible.
- Place Design : During the placement process, components from the netlist are assigned their specific spots in the targetted FPGA device. At the end of this phase, resource availability on the board for the design can be checked.
- Route Design : In this step, all components in the placed design are interconnected and the frequency requirements can be checked.

Bitstream Generation:
Finally, all the information required for the FPGA to behave as an embedded hardware device for the given application is packed into a bitfile which will be copied and programmed onto the FPGA device.

2.6. FINN

FINN is a framework that generates high speed dataflow inference accelerators for Quantized Deep Neural Networks [32]. The framework is able to tailor architectures for each neural network model depending on latency required by the application and practical resource constraints. Being a dataflow architecture, it has the potential to exploit all levels of parallelism possible in DNNs including pipeline parallelism. Topologies of FINN networks reflect the acyclic graph structure representative of CNNs. Each node is implemented using specific configuration of FPGA resources described by High Level descriptions created by FINN. Hence, it is possible to customize each layer's computational resources according to each network's requirements. Successive nodes are interconnected using the AXI Stream interface. FINN is beneficial as it helps to cater to each network specifically; it falls in the far right of the spectrum in Figure 2.4. Therefore, it requires a new bitfile to be created whenever a parameter changes. The following section describes the flow of a software to bitstream representation of an NN in FINN. Then, it provides implementation details of how some important types of computation in FINN.

2.6.1. End-To-End flow

The FINN framework provides a series of transformations that take a description of a QNN as input and provides a bitfile that can be used for high performance inference on FPGAs. This section briefly introduces the end to end flow and then presents a more detailed sequence of operations.

- Frontend : The FINN frontend is responsible for converting trained networks from different frameworks like Tensorflow, Caffe or Pytorch into a common intermediate representation that can be used for further processing.
- Intermediate Representation : In its intermediate representation, the network is represented as a directed acyclic graph. Each computational layer is represented with generic and custom FINN attributes encapsulated in a node of the graph. Edges in the graph represent input and output tensors of each computational layer and contain information about their dimensions and level of quantization. After multiple transformation passes over the network, the graph contains FINN-specific representations of each layer that the FINN back-end can process.
- Backend : Vivado HLS, Vivado, and Vitis are part of the FINN backend and are used to transform the final intermediate descriptions into a bitfile.

The following subsection describes a sequence of operations on a network description.

- tidy up : This transformation infers the dimensions of input and output tensors of each node, performs constant folding, and gives nodes and tensors descriptive names.
- streamline : Even in QNNs, some operations in the forward pass like batch normalisation, α scaling may remain in floating point for the sake of accuracy. This increases both computational complexity and memory requirements in their FPGA implementation. The streamlining transformaton is performed to collapse multiple linear floating point operations into a single integer operation, which is then absorbed into the node of FINN that performs the activation function. [31]
- convert_to_hls : Computational nodes in the network are converted into custom FINN nodes that have specific functions in the FINN-HLS library [1] for their FPGA representations. The custom FINN nodes are described in detail in Section 2.6.2
- create_dataflow_partition : In this transformation, the entire dataflow graph is split into partitions that are composed completely of HLS nodes that can be processed by the FINN back-end and non-HLS nodes.

[1] https://github.com/Xilinx/finn-hlslib

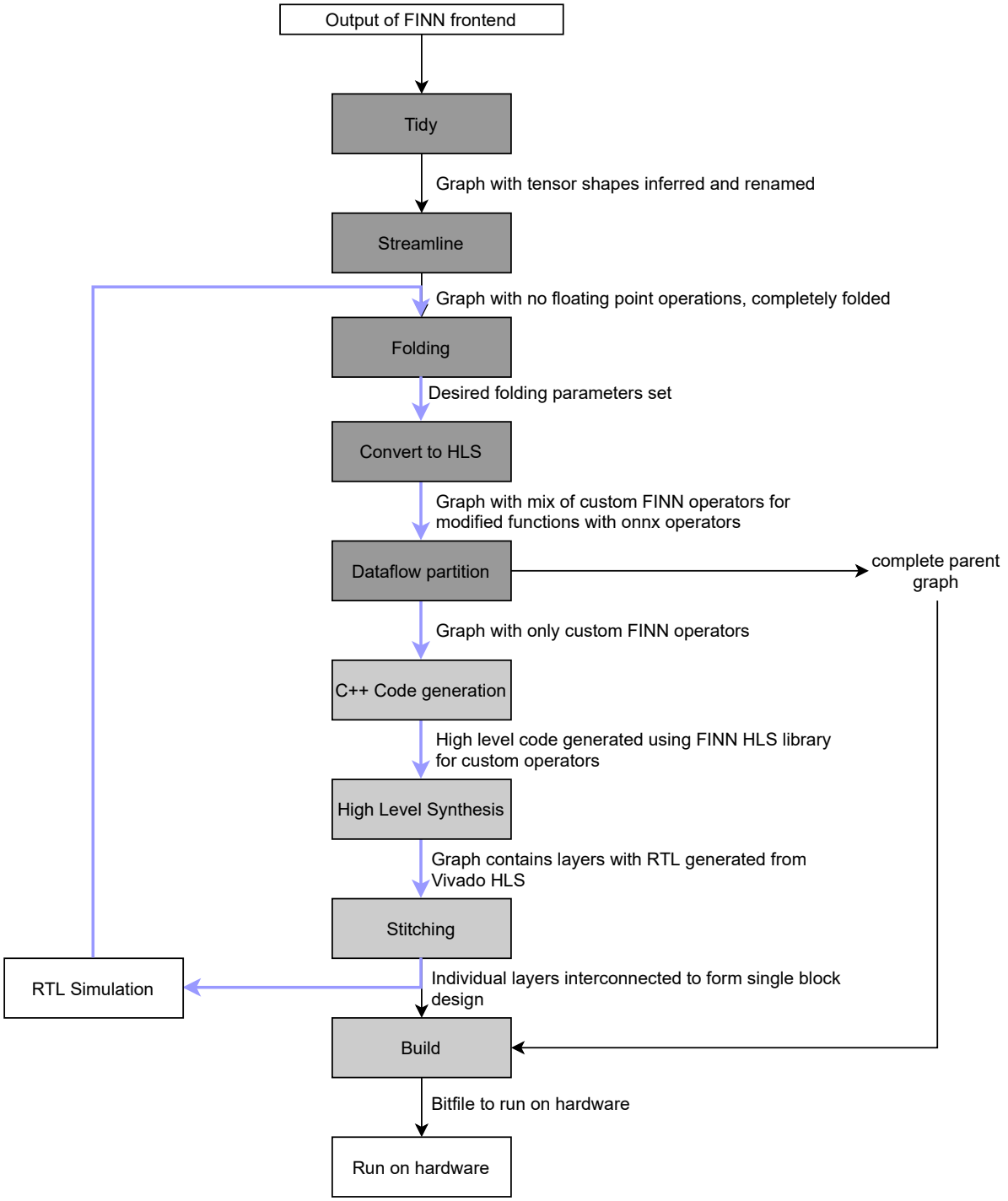


Figure 2.6: FINN End To End Flow

- setFolding : This transformation sets the degree of parallellism for each node based on throughput requirements set by the user. In other words, this transformation modifies the degree of computation in time with respect to computation in space in the final architecture.
- Code generation: At this stage all the data required to design each block is obtained. Using this, FINN generates C++ code in this step (codegen) along with files that store associated parameter data for the network such as weights and thresholds.
- IP Generation : In this stage, FINN creates multiple parallel threads, each of which is responsible for

High Level Synthesis (HLS) of a single node. Vivado HLS is used to this end, It can take the C++ description of a node provided by codegen stage and generate the corresponding RTL description.

- CreateStitchedIP: In this step, Vivado IP Integrator stitches together all the nodes to form the end to end dataflow architecture in Vivado. Currently FINN uses the standard AXI-Stream protocol to interface between individual blocks. Stitched block design from this stage can be used for RTL Simulation to verify functional correctness and throughput.

  Details of AXI Stream Handshake: The AXI stream protocol contains the *ready* and *valid* handshake signals along with the data to be transferred. Master (source module) asserts *valid* and places *data* on the bus; these must remain stable until the handshake is completed. Slave (destination module) asserts *ready* whenever it is ready to receive data (this can be conditional on *valid* being asserted, but it can also be independent). When *ready* and *valid* are asserted simultaneously, a handshake is completed. The data needs to be registered on the slave side in that cycle. The master is allowed to change the value of *data* in the next cycle.

- Build: In this step, we run synthesis, implementation and generate the bitfile for the overall network.

2.6.2. Important customOps

This section describes the structure and operations of the important architectural design blocks of FINN. It also introduces various terminologies used in the FINN compiler that are referred in the rest of the report.

Matrix Vector Activate Unit

The Matrix Vector Activate Unit is reponsible for almost all of the multiply-accumulate capabilities in FINN. In addition to multiply accumulate, it also applies an activation function using successive thresholding. Hence, this block is responsible for almost all of the computation during inference, has the highest impact on latency in the forward pass, takes the longest to synthesise, and is responsible for most of the resource utilisation in the network. It performs computation for fully connected layers in MLPs and CNNs and convolution operation in CNNs. First, we describe how MVAU performs computation for a fully connected layer with complete folding, i.e., when all operations are completely time multiplexed.

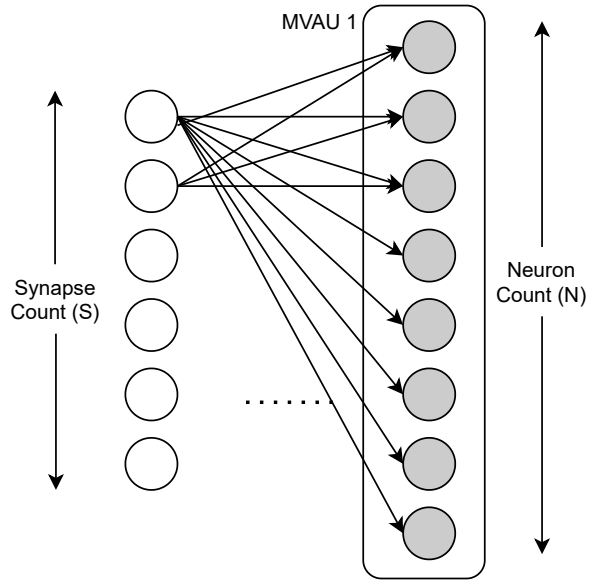
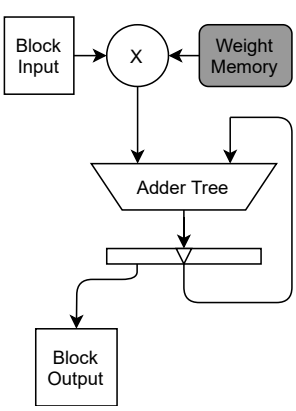
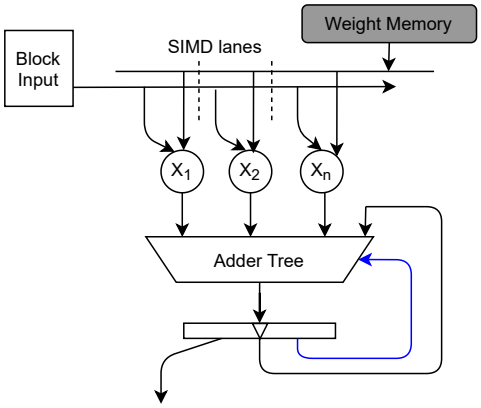


Figure 2.7: General MVAU FC Layer

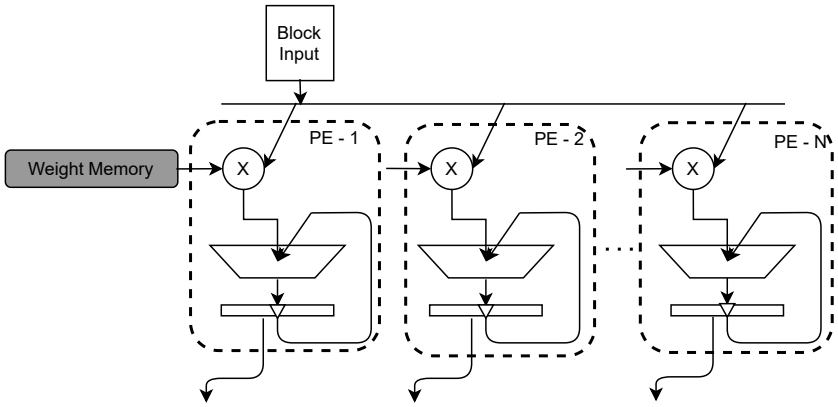
Consider the MVAU for the highlighted layer in Figure 2.7. The general completely folded structure of the MVAU is shown in Figure2.8a. FINN provides two levels of parallel processing in the MVAU, *PE* and SIMD. Increasing PE translates to unfolding across the neuron dimension and increasing SIMD translates to unfolding across the synapse dimension. Correspondingly, neuron fold refers to how many neurons' outputs each PE calculates and synapse fold refers to how many cycles of MAC operations are required by a singe PE to



(a) MVAU : PE = 1, SIMD = 1



(b) MVAU : PE = 1, SIMD = N



(c) MVAU : PE = N, SIMD = 1

Figure 2.8: Modes of parallellism in MVAU

produce the output for a single neuron. Each MVAU has a $[N \times S]$ weight memory associated with it. In FINN, there are 2 modes in which this $[MH \times MW]$ sized memory can be synthesised :

- *const* : Weight memory is synthesised inside the MVAU. The resulting block after HLS Synthesis stage has the weights embedded with the compute unit.
- *decoupled* : Weight memory is synthesised in a separate *memstream* block and is streamed into the computational unit as needed along with inputs from the previous block. Hence, during HLS Synthesis, the weight memory is not synthesised. Rather, it is synthesised separately from a Verilog description of a memory streamer.

For a complete iteration for a single image, the MVAU has to perform $MH \times MW$ computations. Hence, the total number of cycles required for the overall computation is given in equation.2.6

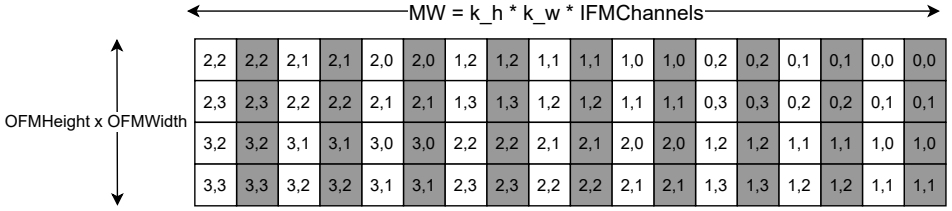
$$Cycles = \frac{MW}{SIMD} \times \frac{MW}{PE} \tag{2.6}$$

For a fully unfolded linear layer, where PE = MH and SIMD = MW, the layer can perform one complete iteration per cycle. Using the same principles, the MVAU is also used for performing convolutions. For this purpose, inputs to the convolution need to be reordered accordingly. The sliding window unit is responsible for this conversion.

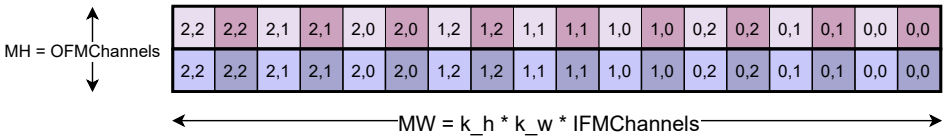
Sliding Window Unit

All convolutions in FINN are modelled as a sliding window unit followed by the MVAU. Hence, the SWU is reponsible for converting the input feature map from the previous layer to a form that is suitable for computation using the same MVAU principles. This is done by storing incoming data in line buffers and reading in

a pattern dictated by the attributes of the SWU. Figures 2.9 demonstrate how the convolution operation in Figure 2.2, , reproduced in



(a) Image Transformation for GEMM - Normal Convolution



(b) Kernel Transformation for GEMM - Normal Convolution

Figure 2.9: GEMM Transformation of Convolutiom

Hence, the total number of cycles for one complete iteration of a convolutional layer is given in equation.2.7.

$$\text{Cycles} = \text{OFMHeight} \times \text{OFMWidth} \times \text{k\_h} \times \text{k\_w} \times \frac{\text{IFMChannels}}{\text{SIMD}} \times \frac{\text{OFMChannels}}{\text{PE}} \tag{2.7}$$

$$\text{DepthwiseConvolutionCycles} = \text{OFMHeight} \times \text{OFMWidth} \times \text{k} \times \text{k} \times \frac{\text{IFMChannels}}{\text{PE}} \tag{2.8}$$

$$\text{PointwiseConvolutionCycles} = \text{OFMHeight} \times \text{OFMWidth} \times \frac{\text{OFMChannels}}{\text{PE}} \times \frac{\text{IFMChannels}}{\text{SIMD}} \tag{2.9}$$

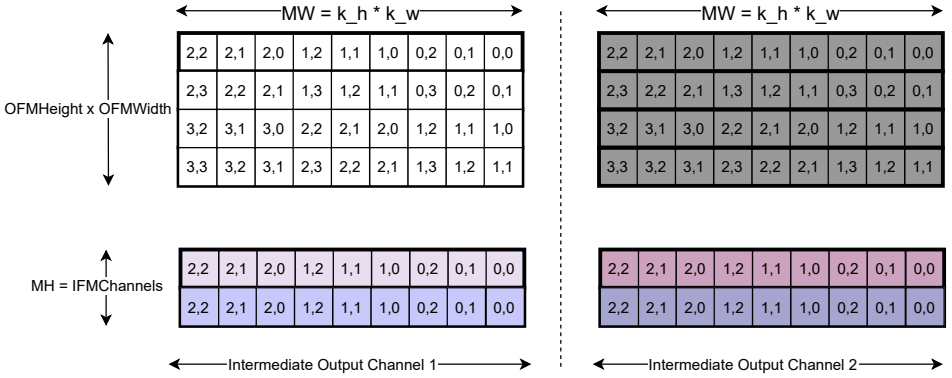


Figure 2.10: Image and Kernel Transformation for GEMM - Depthwise Convolution

The sliding window can also be configured to emit pixels in a form suitable for depthwise separable convolutions. In FINN, depthwise convolutions are modelled as a convolution layer which perform depthwise separable convolutions followed by a fully connected linear layer that performs pointwise convolutions. Figure 2.10 shows how the depthwise convolution in Figure 2.3 can be transformed into a general matrix mutiply operation.

In both types of convolution, PE signifies parallellism across the output channel dimension and SIMD denotes parallellism across the input channel dimension.

The difference between the performance of an MVAU between fully connected and convolutional layer arises because of the levels of parallellism present in FINN. The maximum throughput in convolutional layers is limited by the dimensions of the image that the layer operates on. For typical input image sizes, this can lead to a large number of cycles for a complete iteration of initial layers of DNNs and can bottleneck the maximum throughput of the entire network.

2.6.3. Limitations

The FINN framework has found its use in multiple low latency applications and provides an easy and portable way of customizing NN classifiers for specific applications [9] [16] [10] [14]. To choose an ideal tradeoff between resource usage and latency, a design space exploration between various parallelisation metrics is required. Since FINN falls on the right of the spectrum in Figure 2.4, there is high design complexity and the bitfile has to be changed for every new configuration. Hence, build times for large DNNs are proportionately large in FINN.

We define two checkpoints in the end to end flow that are useful for users of FINN.

- Stage 1: The RTL Simulation checkpoint where RTL for every sub-component of the block design is obtained after HLS synthesis. At this point it is possible to perform cycle-by-cycle analysis of throughput and functional verification of new features in FINN. A user may go through multiple iterations of builds till Stage 1 before proceeding to stage 2.
- Stage 2: Final build checkpoint where the design is implemented, final resource utilisation can be obtained and the design can be verified on an FPGA.

Table 2.1: Build Times of FINN networks

Network	Build Time (hours)
CNV-w1-a1	1 h 46 m
CNV-w1-a2	1 h 44 m
CNV-w2-a2	1 h 49 m
TFC-w1-a1	1 h
TFC-w1-a2	48 m
TFC-w2-a2	52 m
Mobilenet	17 h 30 m
Resnet	82 h 18 m

Table 2.1 shows the build times in FINN for some NN architectures. It is evident that FINN needs to compile faster to be more usable. This thesis first investigates a Fine-grained method of implementation of the FINN pipeline as a method to reduce build times.

The maximum throughput in FINN-based DNNs is limited by dimensions of the input and output feature maps. Hence, by adding another level of performance scaling, FINN can overcome the current limits to parallelism. This thesis also aims to provide a general method to implement Multiple Matrix Vector (MMV) product calculation capability to FINN that can be used where the existing parallelism methods are not sufficient to meet the throughput requirements of the user.

3

Fine-Grained Build Flow for FINN

This chapter identifies the process that bottlenecks FINN build times and the specific part of the FINN structure that contributes to the most significant chunk of this process. Naturally, it is expected to reap most benefits by performing architectural modifications to this bottleneck structure. The internal organisation of this structure was derived from its description and its construction method was modified to reduce its granularity. Using a fine-grain approach, the architecture is broken into smaller and simpler components that are easier for synthesis tools to build and replicate, instead of monolithic modules. Resource overheads contributed by this method are minimized by further simplifications.

3.1. Profiling FINN Build Times

It is important to know the division of total build time that each step of the FINN backend takes to run to completion to identify scope for timing improvement. Some example DNNs were taken through the FINN end-to-end flow. A breakup of their build times is given in Table 3.1. Synthesis (HLS + Vivado RTL synthesis) dominates the runtime for all cases. A general workflow could involve multiple cycles of HLS synthesis before the final build process as it is possible to assess and obtain performance data points for design space exploration after this stage (Stage 1) itself. Therefore, HLS Synthesis was identified as the focus of the modification, as in addition to reaching Stage 1 faster, any structural simplifications made to speed up HLS synthesis can offer advantages in RTL Synthesis times as well.

Table 3.1: Division of Runtime

Network	HLS	Vivado Synthesis	Total Synthesis (HLS + Vivado Synthesis))	Place and Route
CNV-w1-a1	31.25%	45.98%	77.23%	22.77%
CNV-w1-a2	25.10%	51.78%	76.88%	23.12%
CNV-w2-a2	24.33%	49.78%	74.11%	25.89%
TFC-w1-a1	20.60%	45.59%	66.19%	33.81%
TFC-w1-a2	19.63%	37.30%	56.93%	43.07%
TFC-w2-a2	17.30%	39.61%	56.90%	43.10%
Mobilenet	24.73%	36.30%	61.03%	38.97%
Resnet	26.37%	34.16%	60.53%	39.47%

A further breakdown of the contribution of each FINN custom operation to HLS Synthesis times is presented in Table 3.2. As expected, it is MVAU, the primary computational unit in FINN that is responsible for 54% to 88% of total synthesis times. On that account, the algorithm of MVAU was investigated to identify sources of complexity that lead to the large build times.

Table 3.2: Contribution of each custom operation to total HLS synthesis time

Network	SWU	MVAU	MaxPool	Thresholding
cnv11	20.09%	63.24%	6.83%	6.43%
cnv21	27.81%	60.89%	11.69%	8.05%
cnv22	24.71%	53.87%	4.29%	7.25%
tfc11	-	59.50%	8.03%	32.47%
tfc21	-	79.35%	9.77%	10.88%
tfc22	-	78.12%	10.25%	11.63%
mobilenet	10.46%	88.40%	0.37%	-
resnet	2.79%	82.20%	0.18%	9.98%

3.2. MVAU - Details

Algorithm 2 describes the C++ code that implements this MAC block in FINN.

Algorithm 2 Matrix Vector Activate Unit Pseudocode

```
nf = 0
sf = 0
Total_Fold = Neuron_Fold × Synapse_Fold
for i in 0 to Total_Fold - 1 do
  if nf = 0 then
    InputBuffer[sf] = inputport.read()
    inElem = inputport.read()
  else
    inElem = InputBuffer[sf]
  end if
  W = Datafromweightport
  if sf = 0 then
    acc = 0
    for p in 0 to PE do
      act = inElem
      wgt = w[p]
      acc = mac(acc, wgt, act)
    end for
    sf = sf + 1
    for p in 0 to PE do
      outElem = apply_activation(acc)
    end for
    outputport = outElem
    sf = 0
    nf = nf + 1
    if nf = NF then
      nf = 0
    end if
  end if
end for
```

Based on the HLS code, the physical organisation of the MVAU can be visualised as in Figure 3.1. It contains an input buffer component that enables input reuse across time multiplexed operations of the multiply accumulate block. Since each PE performs computation with the same input for different rows of the weight matrix, outputs from the buffer are shared with all the PEs. Each PE also receives its corresponding subset of weights from the weight memory as another input. After computation in the MAC block, the output is thresholded, which is equivalent to applying an activation function. The entire block inside the highlighted grey area of Figure 3.1 forms a single MVAU unit and is monolithically synthesised by Vivado HLS. Apart from the folding factors, each MVAU is uniquely characterised by the thresholding function and parameters

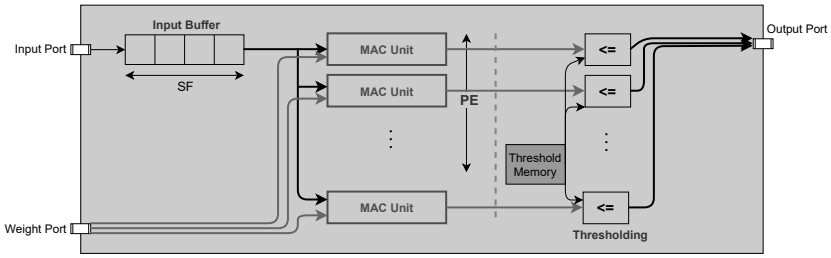


Figure 3.1: Internal MVAU Structure

used for thresholding. Another characteristic of the structure, *'memmode'* defines how the MVAU accesses its weight memory. It can be either in *const* mode or *decoupled* mode (refer to Section 2.6.2).

3.3. Fine-grained methodology

From this structure, it was observed that the design synthesises an array of PE processing elements. The initial solution proposed to reduce the granularity is shown in Figure 3.2. The design can be separated into individual repetitive components and only a single component can be synthesised. The repeated components can be copied at the time of creation of the overall structure.

Hence, the same functionality can be replicated by synthesising the MVAU that is one processing element (PE) long and using Vivado IPI to stitch multiple PEs together with Vivado AXI Stream blocks to direct input, weight and output streams in the order required. Data movement infrastructure is thus removed from High-Level Synthesis. This way HLS synthesis will bear the load of synthesising one PE irrespective of the number of PEs necessary for the amount of parallelism required. Vivado IPI can replicate the synthesised PE by copying it from the cache. Since we need to be able to reuse the same synthesised PE, the PEs need to be made completely identical and all unique parameters need to be separated from the MVAU. Hence, the characteristic weight memory and thresholding unit always need to be separated for the FG implementation. Only the highlighted area in Figure 3.2 would be need to be synthesised. This method enables logic reuse both within and across computational layers.

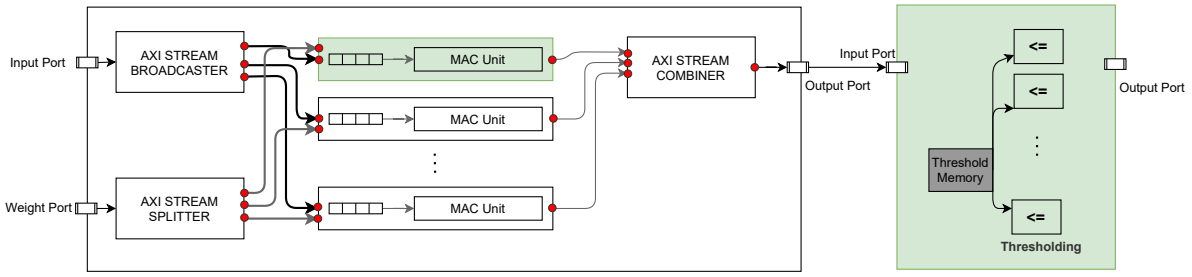


Figure 3.2: Fine Grained MVAU - Naive Approach

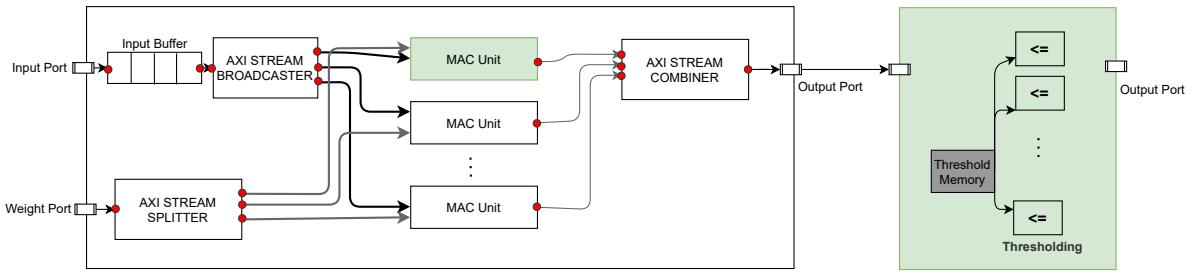
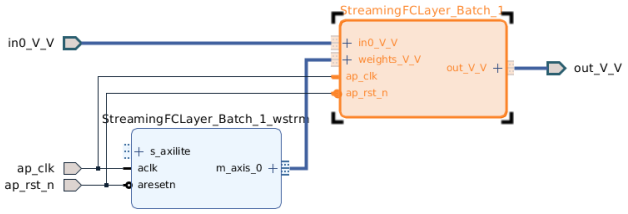


Figure 3.3: Fine Grained MVAU - Optimised Approach

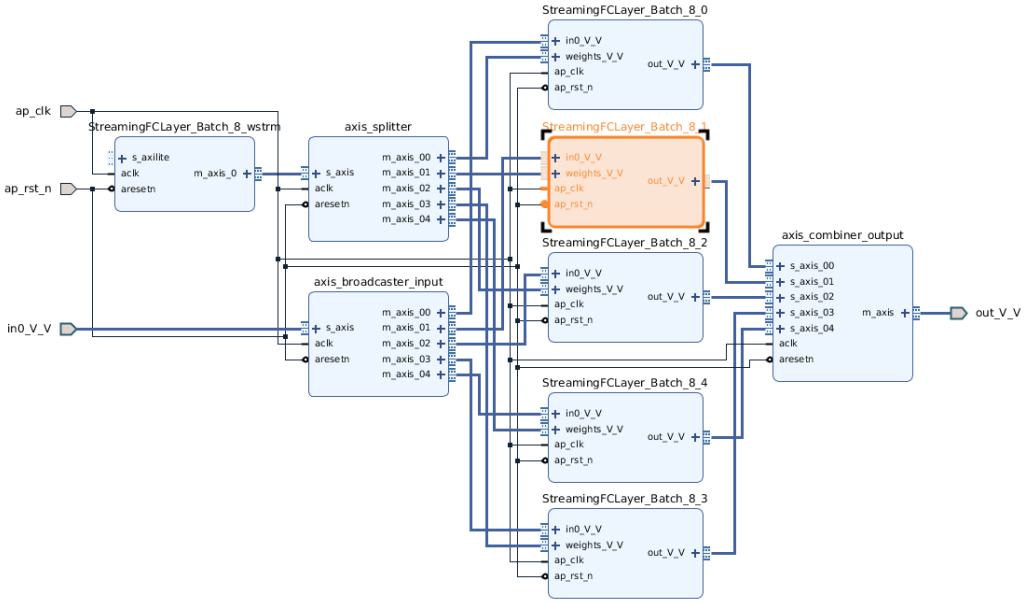
The proposed methodology was integrated into the FINN build flow and its functional correctness was verified on the BNN-PYNQ prototypes. Further, the following architectural modifications were made to improve resource efficiency of the design:

- This structure contains the input buffer repeated across PEs, that would contribute to unnecessary resource utilisation. The HLS description of the MVAU was modified to describe only the MAC operation of the processing element architecture, the input buffer was removed from the design and implemented separately using Verilog. The modified structure is in Figure 3.3.
- This method is also expected to have utilisation overheads because of externally exposed interfaces, that are marked in red. In the proposed approach, each input and output of processing elements will be exposed externally as AXI-Stream interfaces which are also replicated as many times as the number of PEs, whereas these were present as internal connections in the initial coarse-grained (CG) approach. All AXI stream infrastructure paths are double registered, at the source and destination side of the interface. Since inputs and outputs to these PEs are also from Vivado AXI infrastructure with registered interfaces, some of these registers can be removed without affecting the function or attainable frequency of the design. The input stream registers were targeted for removal since the width of the input stream interface is an additional repetition factor present only in FG. Registers on the weight side were also removed to keep identical delays to both inputs of the MAC operation.

This method moves the design complexity to the stitching of components of the block design in Vivado. The differences in the stitched block design between the coarse-grained structure (`CG`) and fine-grained structure(`FG`) produced by Vivado is shown in Figure 3.4. Both structures perform identical functionalities. In both Figure 3.4a and Figure 3.4b, the highlighted part show the portions that are HLS synthesised.



(a) Coarse-Grained Structure of MVAU



(b) Fine-Grained Structure of MVAU

Figure 3.4: Vivado Stitched Block design

3.4. Fine grain VVAU

This section describes briefly the structure of the Vector Vector Activate Unit (VVAU), which is used for depthwise separable convolutions in FINN. Together, the MVAU and VVAU are responsible for all of the MAC functionality in FINN. From Algorithm 4, a subtle difference in the VVAU structure compared to MVAU was observed. The derived structure is shown in Figure 3.5. The modified fine-grained structure is obtained with identical core components as the MVAU, and with different data movement infrastructure and is shown in Figure 3.6. This method thus unifies the primary computational unit for all FINN blocks that perform the MAC operation. This further increases the possibility of logic reuse among different layers of the DNN implementation.

Algorithm 4 Vector Vector Activate Unit Pseudocode

```
nf = 0
sf = 0
Total_Fold = Neuron_Fold × Synapse_Fold
for i in 0 to Total_Fold - 1 do
    inElem = InputBuffer[sf]
    W = Datafromweightport
    if sf = 0 then
        acc = 0
        for p in 0 to PE do
            act = inElem[p]
            wgt = w[p]
            acc = mac(acc, wgt, act)
        end for
        sf = sf + 1
        for p in 0 to PE do
            outElem = apply_activation(acc)
        end for
        outputport = outElem
        sf = 0
        nf = nf + 1
        if nf = NF then
            nf = 0
        end if
    end if
end for
```

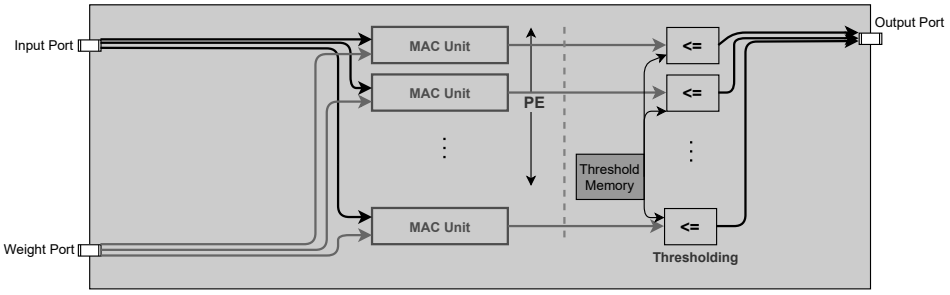


Figure 3.5: Internal VVAU Structure

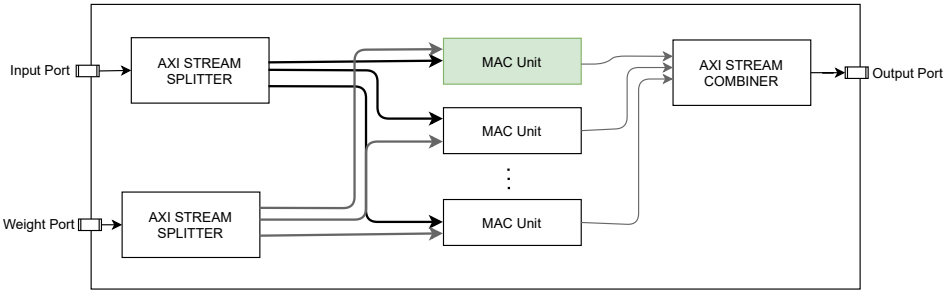


Figure 3.6: Fine-Grained Implementation of VVAU

ADVANTAGES AND DISADVANTAGES OF FINE-GRAINED METHODOLOGY

- The advantage of this method is that it can be directly extended and scaled to any network supported by FINN without any modification.
- Using this method, the computational unit is agnostic to the PE configuration, hence, it enables logic reuse not just within a layer but also across computational layers, as long as they have identical SIMD and MW values.
- An improvement is expected not only in HLS Synthesis time but also in RTL synthesis time.
- There may be utilisation overheads that need to be minimised.
- This method needs to always decouple the threshold blocks and the memory streamer. Synthesising these additional components may lead to build time as well as utilization overheads.

3.5. Example Accelerator Designs

This section will discuss the structures of two example networks supported by FINN that are used in this work to evaluate the proposed modifications to the FINN framework.

3.5.1. BNN-PYNQ Networks

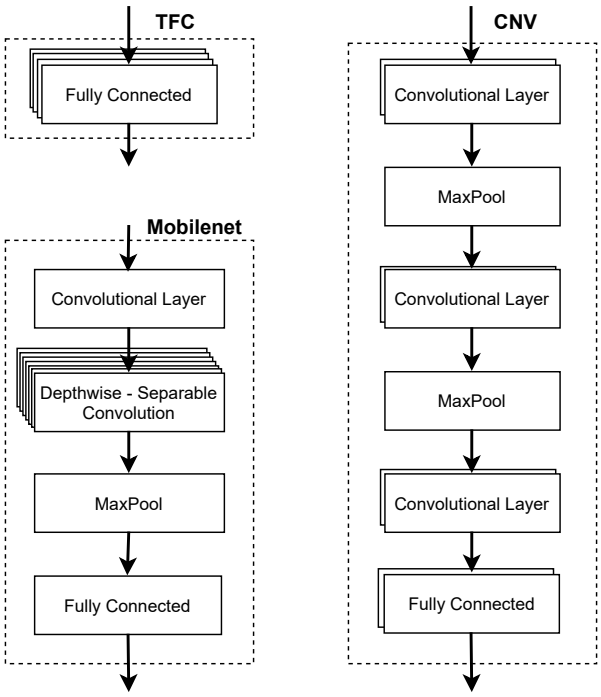


Figure 3.7: Topologies of Example FINN Networks

This class contains a set of Binarized Neural Networks (BNNs) which are ultra-quantized networks with 1 or 2-bit precision of weights and activations. It consists of *TFCs*, a set of MLP networks, and *CNVs*, a set of CNNs. Each set has the same topology with different combinations of bit widths to represent their parameters and inputs. TFCs are built for classifying the MNIST dataset, they operate on 28×28 input images and emit a 10×1 one-hot encoded vector to identify the digit. Their topology consists of four fully connected layers as shown in Figure 3.7. CNVs are designed for classifying both the SVHN and CIFAR-10 datasets, and take 32×32 sized images as input and output a 10×1 one-hot encoded vector. Their topology contains three repetitions of a pair of convolution layers and a MaxPool layer followed by three fully connected layers.

3.5.2. Mobilenet

The Mobilenet accelerator operates with four-bit weights and activations. It consists of one convolutional layer followed by 13 depthwise-separable convolutions, a MaxPool layer and a fully connected layer. It is used to classify the ImageNet dataset and gives a top-1 accuracy of 68%.

4

Evaluation of Fine-Grained flow

In order to understand effects of the fine-grained approach, and advantages obtainable, an initial analysis is performed on a standalone MVAU node. The parallelism factor, PE, is varied across experiments and the impact of this method on both build times and resource utilisation is traced. Further, the number of CPUs used by the build jobs is also varied to note the advantage of parallel processing in synthesis and implementation. Then, an analysis of speedups is provided and overheads are broken down and explained. Speedups can be expected in HLS Synthesis as well as Vivado synthesis because the Fine-Grained (FG) flow works on simpler components than the coarse-grained baseline (CG). FG is expected to have overheads in stitching time due to the presence of finer interconnections and also in resource utilisation.

Two points of the build flow that produce products that are useful to developers and users are identified and analysis is done for each stage separately and then cumulatively. First, it is the RTL Simulation stage where it is possible to completely test, verify and perform cycle-by-cycle analysis of functionality and performance. The second checkpoint is the final build stage when implementation is complete and the design can be used and tested on hardware.

Subsequently, the proposed modifications were implemented and tested on some accelerator networks supported by FINN and the effect of FG was evaluated. All measurements were made on Vivado HLS 2020.1, Vivado 2020.1 and Vitis 2020.1 on the *xirxlabs* servers provided by Xilinx Research Labs.

4.1. Profiling Build Times for Standalone Nodes

A standalone MVAU node with MH = 256 and MW = 1304 was synthesised and implemented with both CG and FG transformation applied with varying folding levels (PE count) to understand the scope of improvement obtainable using this approach. MH and MW refer to the height and width of the weight matrix, that stores the parameters for a given layer (refer to Section 2.6.2). The CG implementation can be used with both internal and external thresholds, and FG can be used only with external thresholds. Hence, the baseline results are reported for the following congifurations : CG with embedded thresholds `CG1`, CG with external thresholds `CG2` and FG. In addition to the number of PEs, the number of CPUs that could parallelly perform synthesis was also varied. The second parameter is important because `FG` breaks the process down into smaller repeated blocks and the improvement can be expected to scale with the number of CPUs running parallelly, constrained by Vivado's memory usage. Synthesis times reported also include times for synthesising the external thresholding unit to ensure comparison of structures with identical functionalities.

First, benefits in timing to reach the RTL simulation stage are analysed (Figure 4.1 and Figure 4.2). The flow involves HLS Synthesis followed by stitching of synthesised components to obtain the overall block design for simulation. For CG, HLS Synthesis time scales with PE while the stitching time remains constant. This is expected since the size of block to be synthesised increases with PE, but the number of blocks to be assembled is not dependent on PE. Contrastingly, for FG, HLS Synth time is constant and stitching time scales with PE. FG synthesises only a 1-PE block irrespective of PE value, and needs to assemble larger numbers of the 1-PE blocks for higher values of PE.

In Figure 4.1, note that there is an overhead in `CG2` compared to `CG1` that arises because it synthesises a separate threshold unit. Despite having identical functionalities, this overhead arises because CG2 externalises some interconnects. Hence, it needs to synthesise more external ports. Further, the MVAU outputs

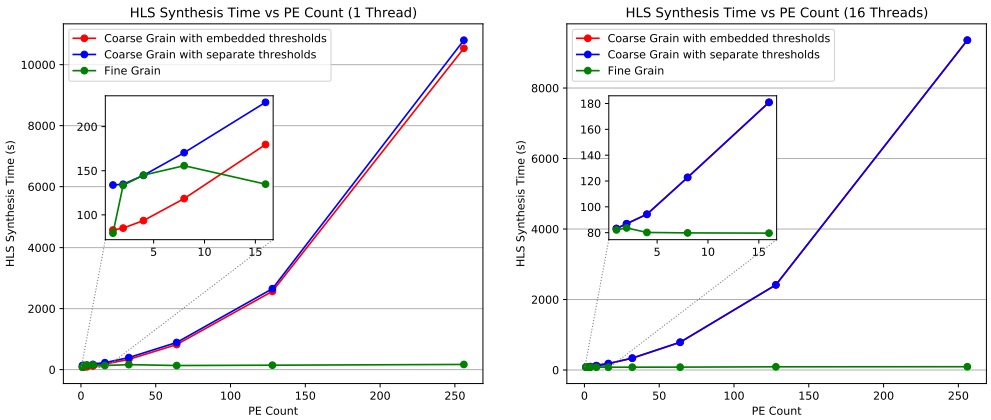


Figure 4.1: HLS Synthesis Time vs PE

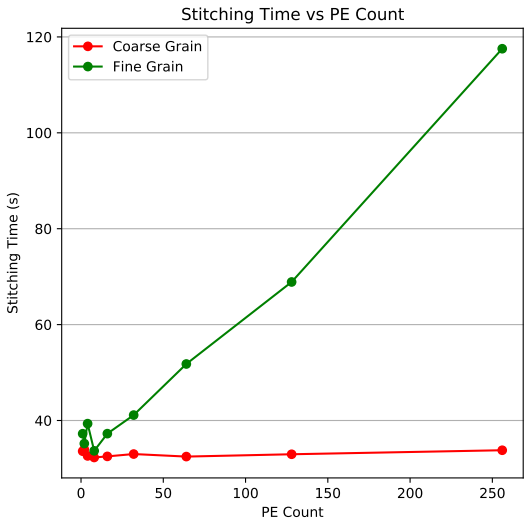


Figure 4.2: Stitching Time vs PE

would not be thresholded values, which have higher bit widths than thresholded outputs in CG1. So, CG1 has more number of wider external ports. For FG, which can operate only with separate thresholds, to provide timing gains, its benefits must be able to compensate for this slowdown. We can see that for smaller values of PE (in the inset region of Figure 4.1a), there is a slowdown in HLS Synthesis due to FG indicating that FG does not provide compensatory benefits for these values. However, removing this overhead is as simple as synthesising the threshold block parallelly on another CPU.

From Figure 4.1.b that performs the same comparison, but on 16 CPUs, it is observed that this overhead can be alleviated. Benefits of FG scale proportionally with increasing PE values from 11% improvement for PE = 1 to 95% for PE = 256. This is expected, since the workload of CG increases proportionally with increasing PEs, while FG always has to synthesise a block of the same size. Though stitching times increase proportionately with PE for FG, FG is able to give improved runtimes cumulatively for a design that can be simulated (Figure 4.3). This concludes the observations for the first part of the flow.

The following steps have to be performed after HLS Synthesis to get the final implemented design:

- Vivado Synthesis : In this step, all RTL designed components and HLS generated RTL for custom nodes are synthesised into a netlist. FINN uses bottom-up synthesis flow and a netlist for each component is obtained separately at the end of this stage. The size of the synthesis job is also different between

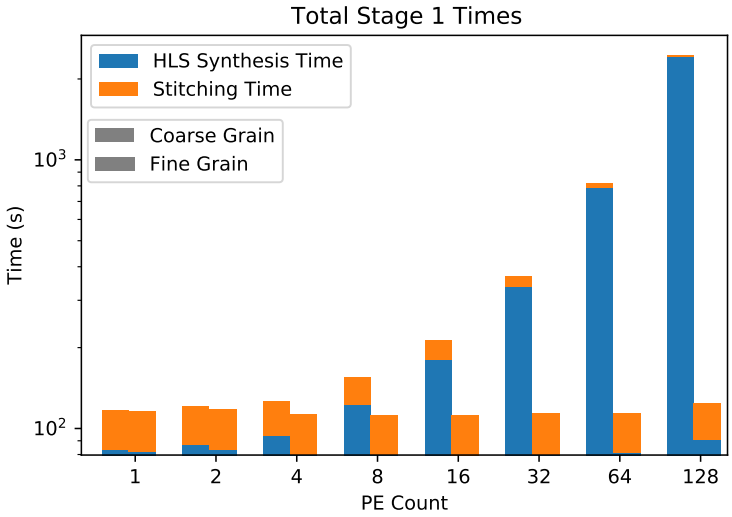
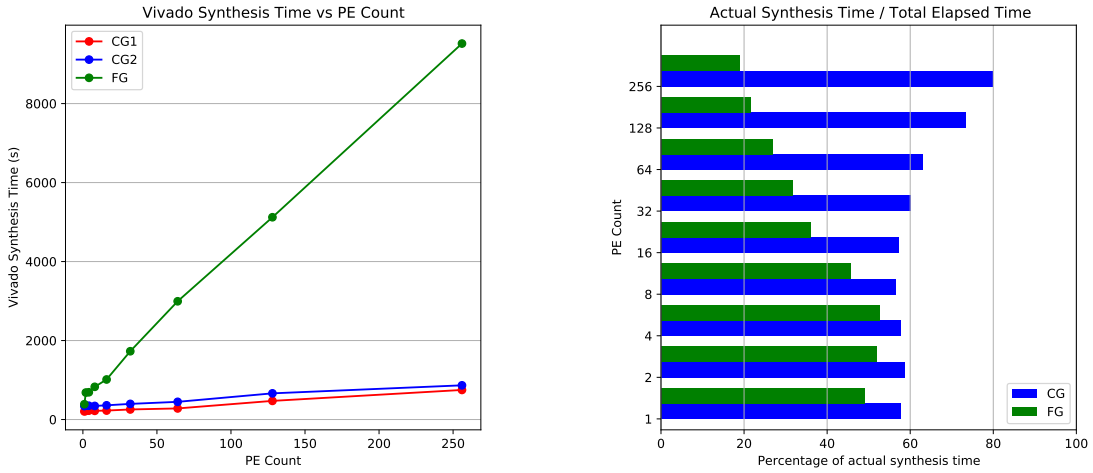


Figure 4.3: Stage 1 Cumulative Times



(a) Standalone Vivado Synthesis Times on 1 CPU

(b) Ratio of Actual to Elapsed Synthesis Times

Figure 4.4: Vivado Synthesis Times and Overheads

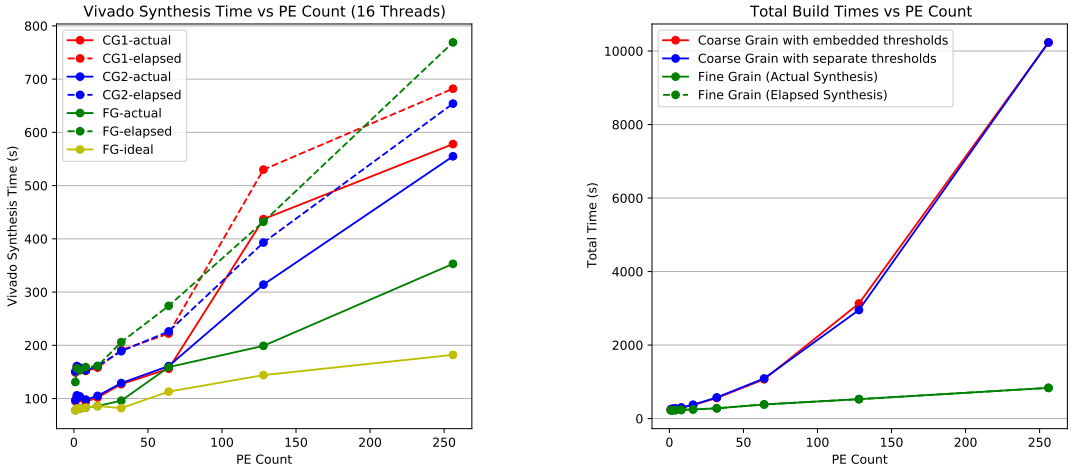
CG and FG. For CG, Vivado needs to synthesise an RTL description of PE-sized MVAU, whereas for FG, Vivado would synthesise the RTL of a single PE only. For other PEs, it can reuse this synthesised design from the cache as they would be completely identical.

- Implementation : A routed design for the required FPGA part is obtained from the synthesised netlist.

FG is expected to have an advantage in Vivado synthesis since RTL generated by HLS Synthesis for a 1-PE MVAU would be less complex than a larger MVAU. Replication of finer components would just involve retreiving the synthesised design from the cache, rather than synthesising a multi-PE MVAU.

Results of Vivado synthesis performed on 1 CPU are shown in Figure 4.4a. While this does not demonstrate the speed-up potential of FG, it gives an indication of additional synthesis workload introduced by FG due to the extracted components (Figure 3.3). Contrary to expectations, a slowdown in synthesis times is observed. The slowdown also increases with increasing PEs. These results are further broken down to identify sources of overheads:

- PE = 1 : First, the case of PE = 1 is observed. Here, the overhead is only because Vivado needs to syn-



(a) Synthesis Times with parallel synthesis on 16 threads

(b) Total Implementation Times with parallel synthesis on 16 threads

Figure 4.5: Vivado Synthesis Times and Total Implementation Times

thesise an additional thresholding unit and its threshold memory. This problem can also be alleviated using parallel processing.

- PE > 2 : The overhead is due to 2 factors. Firstly, additional AXI infrastructure blocks need to be synthesised for movement of input, weight and output streams. Secondly, there is an overhead that scales which is accounted to Vivado's inefficieny in invoking the cached synthesis jobs (Figure 4.4b). For each PE, Vivado requires to start new jobs and perform housekeeping tasks such as loading project part and timing information. This takes up almost 30 seconds even for a single cache retrieval for a replicated PE. The cache retrieval time is almost equal to synthesis time for a single PE. With increasing PEs, the number of cache retrievals increases. Hence, by comparing total elapsed synthesis times, it is not possible to evaluate the impact of hardware modifications introduced by FG, due to bottlenecks introduced by the software that implements the hardware. However, it is possible to obtain information about the *actual* time that Vivado spends on synthesis. Using this, it is possible to isolate the impact of hardware modifications without the influence of Vivado. Henceforth, all analysis will be done on both the actual synthesis times reported by Vivado and the total elapsed time.

Next, build times are measured after running synthesis on 16 CPUs. The number of threads was chosen arbitrarily. In Figure 4.5a, dashed lines refer to the measured total elapsed time and solid lines indicate the actual synthesis times. Speed-up is obtained in the FG method for all values of PE, though it is not as significant as the improvement in HLS Synthesis. The primary reason is that Vivado synthesis times in the baseline method itself do not scale as much with PEs as HLS Synthesis. Hence, the scope for improvement is limited. The overhead is also due to a limitation of Vivado AXI infrastructure blocks. They directly support only upto 16 Master or Slave AXIS ports. When the broadcast value is higher than 16, AXI blocks that perform identical functionalities need to be replicated. Since Vivado does not cache AXI infrastructure blocks and repeats synthesis of even identical blocks, this leads to some overheads. The *ideal* line in Figure 4.5a shows the speed-up that can be obtained by creating custom AXI stream infratructure without port limitations.

Implementation:

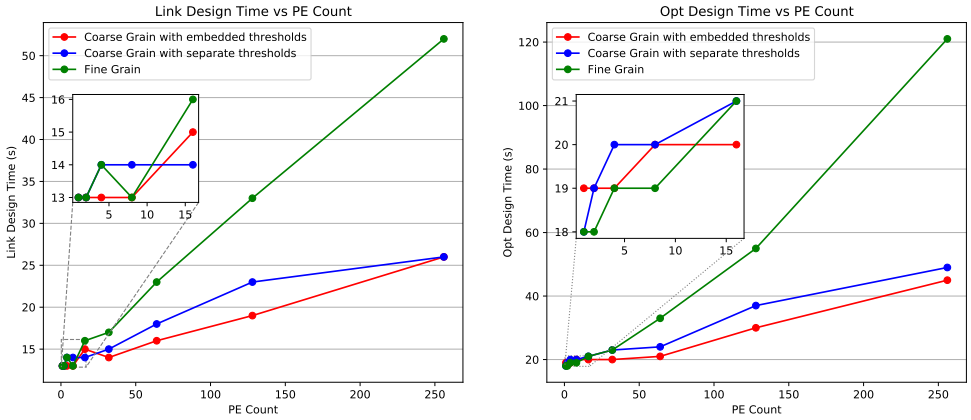


Figure 4.6: Impact of Fine-Grain on Implementation Steps

The impact of FG on the affected sub parts of implementation times (refer Section 2.5.1) is shown in Figure 4.6. There is an overhead in the opt and link design stage for FG. This is because, for FG, Vivado obtains complete information about the entire block design only in this phase and makes high level optimisations, whereas this information is already available during HLS Synthesis for the CG approach. During place and route phase, the cumulative netlist is already generated, so FG exhibits identical timing as CG.

The overall impact of the finegrain method is shown in Figure 4.5b. Hence, it can be concluded that FG has a positive impact on build time and the magnitude of improvement is directly related to the value of PE.

4.2. Resoure Utilisation - Standalone

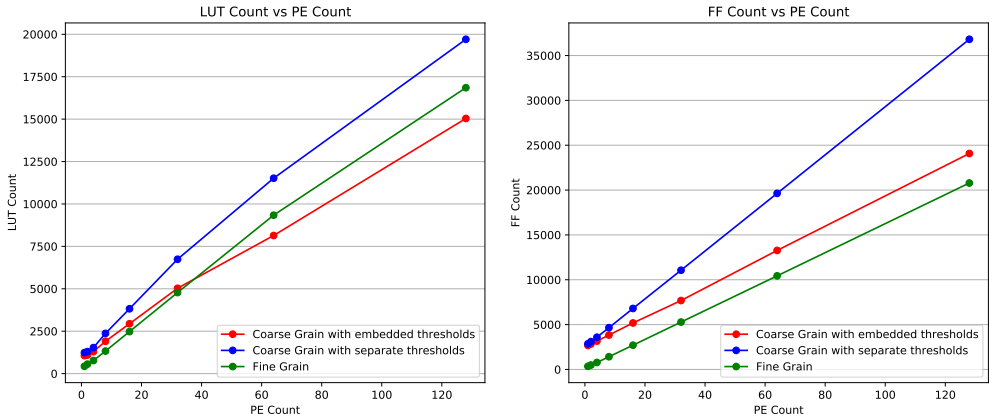


Figure 4.7: Resource Utilisation

In this section, utilisation overheads are measured. FG and CG utilisations for each PE value is shown in Figure 4.7. It is interesting to note that there is in fact an improvement in utilisation with the FG implementation for PE values smaller than 64 and a maximum overhead of 10%. This is due to the removal of registers from the AXI infrastructure of weight stream ports in FG. An overhead that steadily increases with PE count and compensates for this improvement is because of the presence of wide pre-thresholded output stream AXI infrastructure of each PE which is repeated in FG.

In the next section, overall impact of FG on end to end networks are measured and explained based on the above analysis of its effects on a single node.

4.3. Build Time Analysis on Networks

The fine-grained methodology was implemented and verified on hardware for various networks that FINN currently supports. The timing results for each stage of the flow are reported for synthesis with both 1 thread and 16 threads. Contrary to the standalone case, parallelising the synthesis process of networks will be advantageous to the coarse-grained approach as well, since it will now be able to process multiple monolithic layers simultaneously.

Table 4.1: Time taken for flow upto RTL Simulation Stage for Different Networks (1 thread)

Network	CG			FG		
	HLS Synthesis Time	Stitching Time	Total Time	HLS Synthesis Time	Stitching Time	Total Time
CNV11	33 m 2 s	59 s	34 m 1 s	29 m 59 s	61 s	31 m 1 s
CNV21	25 m 59 s	63 s	27 m 2 s	30 m 28 s	68 s	31 m 33 s
CNV22	26 m 37 s	63 s	27 m 40 s	30 m 26 s	67 s	31 m 36 s
TFC11	12 m 16 s	39 s	12 m 55 s	9 m 7 s	44 s	9 m 51 s
TFC21	9 m 25 s	42 s	10 m 7 s	9 m 51 s	49 s	10 m 40 s
TFC22	8 m 56 s	39 s	9 m 35 s	9 m 19 s	47 s	10 m 6 s
Mobilenet	259 m 34 s	2 m	261 m 34 s	82 m 35 s	15 m	97 m 35 s

Table 4.2: Time taken for flow upto RTL Simulation Stage for Different Networks (16 threads)

Network	CG			FG		
	HLS Synthesis Time	Stitching Time	Total Time	HLS Synthesis Time	Stitching Time	Total Time
CNV11	8 m 1 s	59 s	9 m 2 s	2 m 38 s	61 s	3 m 39 s
CNV21	3 m 49 s	63 s	4 m 53 s	2 m 38 s	68 s	3 m 46 s
CNV22	3 m 55 s	63 s	4 m 59 s	2 m 34 s	67 s	3 m 42 s
TFC11	3 m 54 s	39 s	4 m 34 s	1 m 2 s	44 s	1 m 47 s
TFC21	4 m 8 s	42 s	4 m 50 s	1 m 11 s	49 s	2 m 0 s
TFC22	3 m 45 s	39 s	4 m 25 s	1 m 6 s	47 s	1 m 54 s
Mobilenet	70 m 15 s	2 m	72 m 15 s	5 m 58 s	15 m	20 m 58 s

Table 4.3: Time taken for flow upto Final Implementation Step for Different Networks (1 thread)

Network	CG			FG		
	HLS Synthesis	Vivado Synthesis	Total Synthesis	HLS Synthesis	Vivado Synthesis	Total Synthesis
CNV11	33 m	48 m	81 m	31 m	89 m	118 m
CNV21	25 m	53 m	79 m	31 m	94 m	124 m
CNV22	26 m	54 m	81 m	31 m	104 m	134 m
TFC11	12 m	27 m	39 m	9 m	33 m	42 m
TFC21	9 m	17 m	27 m	10 m	34 m	43 m
TFC22	8 m	20 m	29 m	10 m	36 m	45 m
Mobilenet	259 m	381 m	640 m	97 m	892 m	975 m

Table 4.5: Network Nodes Summary

Network	Number of FC Layer Nodes	PE Values
CNV11	9	[16, 32, 16, 16, 4, 1, 1, 1, 5]
CNV21 & CNV22	9	[8, 16, 8, 8, 4, 1, 1, 2, 5]
TFC	4	[16, 8, 8, 8]
Mobilenet	28	[32, 32, 16, 32, 16, 64, 32, 16, 16, 32, 32, 8, 16, 16, 32, 16, 32, 16, 32, 16, 32, 16, 32, 4, 16, 8, 32, 4]

Table 4.4: Time taken for flow upto Final Implementation Step for Different Networks (16 threads)

Network	CG			FG		
	HLS Synthesis	Vivado Synthesis	Total Synthesis	HLS Synthesis	Vivado Synthesis	Total Synthesis
CNV11	9 m 1 s	4 m 2 s	12 m 3 s	2 m 39 s	6 m 14 s	8 m 52 s
CNV21	3 m 53 s	5 m 6 s	8 m 56 s	2 m 46 s	7 m 57 s	10 m 35 s
CNV22	3 m 59 s	5 m 0 s	8 m 56 s	2 m 42 s	8 m 36 s	11 m 11 s
TFC11	3 m 54 s	3 m 43 s	7 m 38 s	1 m 47 s	2 m 26 s	3 m 29 s
TFC21	4 m 50 s	2 m 58 s	7 m 6 s	1 m 0 s	2 m 31 s	3 m 42 s
TFC22	3 m 25 s	5 m 4 s	8 m 50 s	1 m 54 s	2 m 45 s	3 m 52 s
Mobilenet	70 m 15 s	31 m 33 s	101 m 48 s	5 m 58 s	55 m 32 s	61 m 30 s

4.3.1. Effects on HLS Synthesis

The results of the first step of flow after running on 1 CPU and 16 CPUs are summarised in Table 4.1 and Table 4.2 respectively. *CNVs* and *TFCs* are part of the BNN-PYNQ prototypes. The numbers on their label denote the bit-widths of weights and activations. For *tfc11*, the baseline network is implemented with separate thresholds to understand improvements with this variation as well. For each network, results for synthesis with 1 thread is discussed first, followed by synthesis with 16 threads.

The standard configuration of TFC networks have 4 MVAU nodes with PE values shown in Table 4.5. For *TFC11*, the baseline design is implemented with separate thresholds. Based on our analysis for standalone nodes, there is an expected improvement with FG if PE >= 8 and no significant impact for nodes with PE < 8 when CG is also implemented with separate thresholds. Hence, all 4 nodes are expected to individually gain times with improvements ranging from 8% to 40%. The measured cumulative speed-up in HLS Synthesis for this network is 26% (Table 4.1). For TFC21 and TFC22, the baseline design is implemented with embedded thresholds. Hence, a slowdown of 30% is expected for nodes with PE = 8, and a 25% speed-up is expected for the single node with PE = 16. Cumulatively, HLS Synthesis would have to do more work with FG, hence, we do not obtain a speed-up in FG for both these networks. This is also corroborated by the collected timing information in Table 4.1.

Next, we analyze the effect of parallel synthesis of TFC networks. An advantage is obtained here even for *TFC12* and *TFC22* because the increase in workload due to synthesis of separate thresholds in fine-grain is alleviated by distributing the work among multiple threads. Since the number of additional jobs (4) is lesser than the number of threads (16) allowed for synthesis in this experiment, the additional workload is completely hidden by parallelisation. All TFC networks are able to show benefits of upto 70% HLS Synthesis time improvement.

Results obtained in Table 4.5 for the CNV networks are reasoned below: *CNV11* has different folding factors from *CNV21* and *CNV22* and will be reasoned separately.

- *CNV11* contains four nodes with PE > 16 and five nodes with PE < 16. The speed-up obtained by FG in the larger nodes can be expected to be compensated by the small PE nodes based on analysis of the standalone node. Therefore, no speed-up is expected for the sequential case, when synthesis is performed on just one CPU.

  With parallelization, an improvement can be expected inspite of the increased workload. Eight new jobs are introduced by FG in addition to the 19 jobs already present in CG. Since 16 threads are present for parallelisation, CG already requires two rounds of parallel threads. Since the number of additional jobs introduced by FG can be scheduled to synthesise within the second group of 16 jobs, the overheads are hidden and an improvement is observed.

- Based on their default folding, *CNV21* and *CNV22* have four nodes with PE > 8 and 5 nodes with PE < 8. So cumulatively, more work would done by the FG method and again, no speed-up is expected for the sequential case. With parallel scheduling, there is an improvement in times, however, the improvement is not as large as *CNV11* since the default folding factors, that determine the value of PE are smaller in these networks.

It was noted that, as expected, FG times were the same for HLS synthesis of all three CNV networks despite having different folding factors.

The Mobilenet-v1 is much larger than the BNNs. 24 out of 28 of its computational nodes have PE > 16. Almost 50% of its nodes have PE > 32 as well. So, more benefits can be extracted on this network using FG. We observe a 3× build time improvement due to the FG approach. With parallel processing, FG is able to provide 11 times faster HLS Synthesis than CG.

4.3.2. Effects on Vivado Synthesis

Stage 2 results involves the synthesis of all structures that were separated to synthesise externally in the fine-grain method in addition to the components that were HLS synthesised. For each MVAU node, this involves additional synthesis of the following five components for each MVAU node of the network for FG:

- Input Buffer
- AXI Stream Broadcaster
- AXI Stream Scatter
- AXI Stream Commbiner
- Thresholding Unit

Hence, the number of additional jobs due to FG scales proportionally with the number of modified MVAU nodes that need to be synthesised. For CNV and TFC networks, the number of additional jobs is approximately equal to the baseline number of jobs, so, the total number of jobs almost doubles due to FG. Further, despite the simplicity of the additionally introduced structures, their synthesis times are roughly equal to that for a single PE which performs actual computation. This indicates that Vivado synthesis itself involves some housekeeping processes that have fixed time requirements irrespective of the complexity or size of the RTL and always takes up a certain minimum amount of time. This fixed time also scales linearly with the number of additional jobs.

Moreover, Vivado Synthesis times of MVAUs also do not scale with PE count as much as HLS Synthesis. Hence, the benefits provided by FG in RTL Synthesis are small while the overheads are constant and large. Hence, synthesis times with FG are always greater than CG (Table 4.3). The severity of these overheads are allayed when using higher number of threads (Table 4.4), but are not compensated.

Thus FG is seen to perform poorly in Vivado Synthesis. The implementation time was not significantly affected by the fine-grain method and hence, it is not reported. The overall impact is dependent on whether the improvement in HLS Synthesis times is able to compensate for the disadvantage in Vivado Synthesis times.

Table 4.6: Resource Utilization of FINN Networks

	Primitives	TFC11	TFC21	TFC22	CNV11	CNV21	CNV22	Mobilenet
Baseline	LUTs	8606	11374	16112	19437	25193	32281	368902
	FFs	13463	15367	20176	26676	32951	37221	265908
Fine-grained	LUTs	6413	10615	14081	20807	22817	29741	407114
	FFs	9673	13640	16034	30403	26159	29856	116476

Table 4.6 shows the effect of fine-grained method on the resource utilisation of some FINN networks. It was observed that FG improves the resourse utilisation in all cases except for Mobilenet, where there is a 10% increase in resources due to FG.

4.4. Case Study - Network Intrusion Detection System (NIDS)

NIDS is a Multi-Layer Perceptron used to detect malicious network packets. It operates on real-time network data and needs to be able to process millions of packets per second. Hence, this network is an ideal candidate for acceleration using FINN. This particular network is trained on the *UNSW-NB15* dataset [25]. It network has a similar structure to the *TFC* networks, four fully connected layers but different number of neurons. To obtain maximum throughput, all layers are fully unfolded here, i.e., there is no time multiplexing within a layer and each layer can produce all its outputs every cycle. Hence, a throughput equal to the clock rate can be obtained since the network has only fully connected layers.

Since all the layers have full unfolding, this network represents an edge case where the most improvement in FG can be obtained and was investigated more deeply.

Table 4.7: NIDS Build Times

Flow stage	**Time CG (minutes)**	**Time FG (minutes)**
HLS Synthesis	112	4
Vivado Synthesis	13	22
Implementation	15	26

Table 4.8: NIDS Final Utilisation Results

CG		FG		FG - improved	
LUTs	FFs	LUTs	FFs	LUTs	FFs
14252	12971	38752	32836	21452	20524

The timing improvement and utilisation results are shown in Table 4.7 and Table 4.8. A significant improvement (28×) in HLS Synthesis timing was observed as expected due to the large folding factors. However, it was also observed that the resource utilisation is more than twice the utilisation of CG method. Upon further investigation, it was found that the network has *sparse* weights. The CG method was implemented in 'const' weight mode, where the weight memory of the MVAU node is embedded into inside the the node itself. So, weight parameters are available to computational units during HLS Synthesis process itself. Since the network is fully unfolded, i.e., there is no time multiplexing, the computational unit is completely aware of all the coefficients to be used in the MAC operation during HLS Synthesis itself. Most parameter coefficients were observed to be 0, Vivado HLS is able to optimize away most of the computational logic by reducing it to just an addition of the inputs that have their respective weight coefficients set. On the other hand, FG can only be implemented in 'decoupled' mode, where the weight memory is implemented outside the MVAU, hence, HLS Synthesis is performed only for computational part of the MVAU. The RTL generated after HLS Synthesis is unaware of the weights and is designed to handle any generic computation, and subsequently, generic hardware is synthesised. The hardware becomes aware of the weights only during the *opt* design phase of the implementation process, since the weight memory is stitched with the computational node and the entire design is linked in the *link_design* phase. Now, Vivado is still able to sweep away some of the extra hardware during the constant propagation phase, however, it is not able to do it as efficiently as generating the optimised hardware itself.

Hence, to aid the Vivado tool to make the same optimisations in fully unfolded cases for decoupled mode, a specific architectural modification is proposed. This involves shuffling the weight memory to have all the sparsely set weights entirely in either the LSB or MSB. To obtain correct functionality, the inputs need to be reshuffled as well. Since this is a static shuffling for a given network, it can be done at no resource cost. Using this method, utilisation dropped by 45%, and FG was able to obtain 28× HLS time improvement and 2.7× overall build time improvement.

5

Increasing FINN Scalability

This chapter describes the throughput bottleneck in FINN that could prevent network topologies from reaching ultra-low latencies. It aims to develop MMV (Multiple Matrix Vector multiplications), a general solution to perform computations for multiple output pixels simultaneously, that can be utilised when existing methods for performance scaling are unable to meet the throughput demand. Section 5.2 describes an initial solution to implement MMV and its limitations. Section 5.3 provides the modification to the FINN dataflow that can remove this limitation and introduces additional modules that were developed to facilitate MMV.

5.1. Throughput bottleneck

The number of cycles (N) required by an MVAU in a convolutional layer to complete execution for a single image is given by

$$N = \text{OFMWidth} \times \text{OFMHeight} \times \text{k\_w} \times \text{k\_h} \times \frac{\text{IFMChannels}}{\text{PE}} \times \frac{\text{OFMChannels}}{\text{SIMD}} \tag{5.1}$$

Since there is no method of parallelising across the output image dimensions, it can present a bottleneck to the maximum throughput acheivable. Due to the presence of pooling layers, frontal size of feature maps progressively decrease along the path of a DNN. Initial DNN layers that operate on feature maps with larger frontal dimensions and shallow depths are sometimes not able to match throughput of subsequent layers with deeper channels and smaller image dimensions which can exploit the existing SIMD and PE parallelism extensively. With a provision to perform Multiple Matrix Vector (MMV) multiplications in a single cycle, it is possible to obtain corresponding results for multiple output pixels simultaneously. The following section explores the possibility of adding this third level of parallelism to FINN dataflow.

5.2. Single layer MMV

The existing PE computational units in an MVAU can be replicated *MMV* number of times to improve parallelism by a factor of MMV. This MMV solution is implemented on top of the fine-grain modification detailed in Chapter 3. This provides an advantage as only one processing element actually has to be synthesised per MVAU layer, irrespective of the parallelism needed. Correspondingly, the sliding window unit (SWU) must also be modified to produce information for MMV pixels simultaneously. The SWU is usually implemented using BRAM buffers inferred as Simple Dual Port RAMs. Since data can be read from only one address at a time, the SWU would need MMV repeated buffers. Then, outputs of each matrix vector operation can be switched in the required order to the next layer using the AXI Stream Switch. In case of FINN, the next layer accepts pixels only in raster order and all channels of a pixel have to arrive before any new information about the next pixel is obtained. Additionally, since inputs to the switch arrive at the same time, but have to be switched out sequentially, FIFOs are required to queue the pixel information data that have to wait for the previous pixel to be completely switched out. The structure of the proposed solution is shown in Figure 5.1.

For the structure in Figure 5.1, and an MVAU having parameters NF = 4, SF = 9 and MMV = 3, the output waveform is shown in Figure 5.2. To revisit, *SF* is the number of cycles needed to complete a single MAC operation. Output of a single MAC operation corresponds to pixel information for a single output channel.

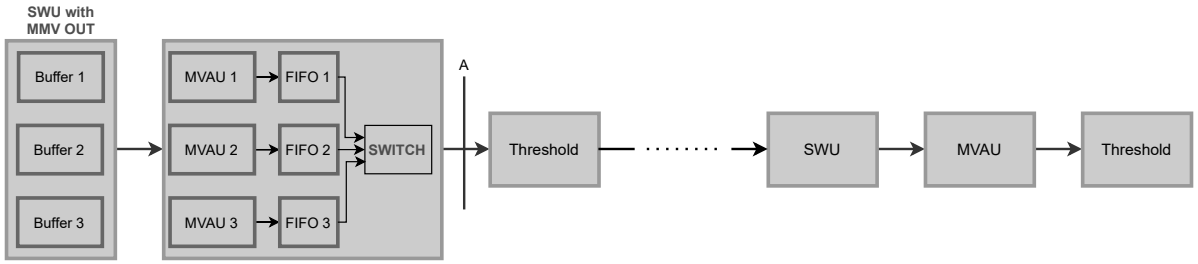


Figure 5.1: MMV Initial Solution

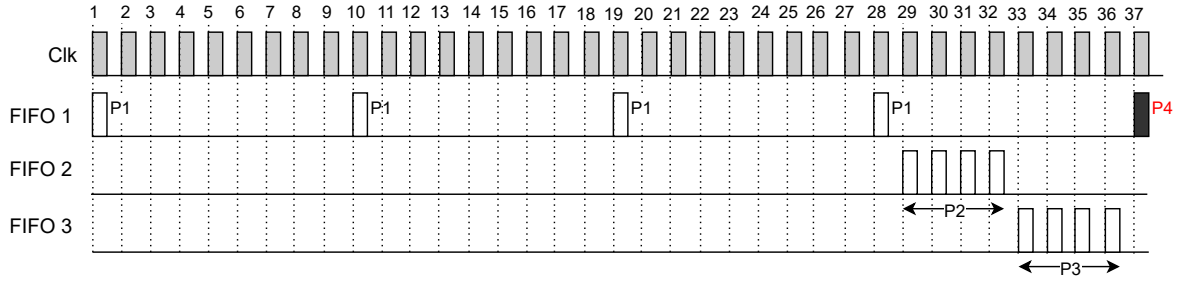


Figure 5.2: MMV Problem

NF is the number of MAC operations to obtain all output channel information for a single pixel. Hence, it takes a total of $NF \times SF$ cycles for all channel information about a single pixel to be calculated in the MVAU. From the waveform, it is apparent that the following constraint applies to obtain extra performance using MMV:

$$(MMV - 1) \times NF < SF \tag{5.2}$$

In fully unfolded networks, where all existing PE and SIMD parallelism has been utilised, SF = 1 and NF = 1. So, it is not possible to scale performance using this approach.

5.3. Multi-layer MMV

For MMV to be useful, subsequent layers must be modified to receive multiple pixels produced by the MVAU simultaneously. The parallely produced data needs to be sequentialised at the point of dataflow when MMV mode of parallelism is not required to meet throughput requirements anymore. Further, the process of sequentialisation must not slowdown the improvement obtained from MMV. The constraint in Equation (5.2) exists because subsequent layers require the input pixels to be of a certain order. If we remove the constraint of raster order of inputs from the subsequent layers, this limitation can be overcome. In this regard, the structure can modified as shown in Figure 5.3, where some layers like thresholding are easily amenable to a transformation to process multiple pixels simuultaneously.

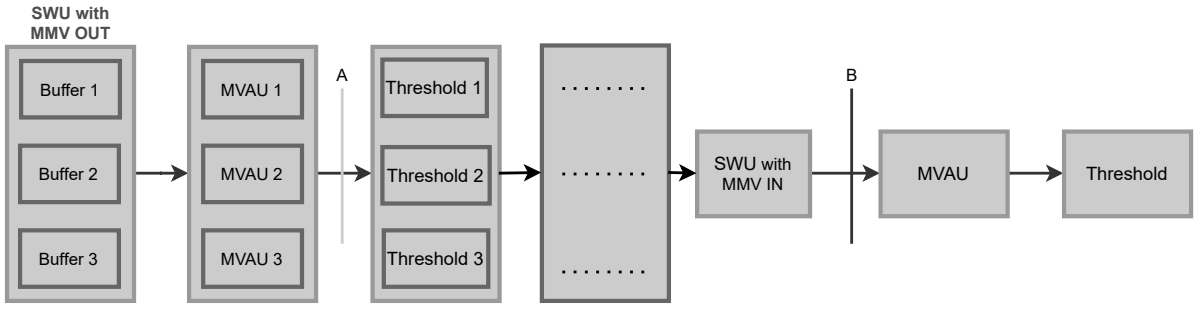


Figure 5.3: MMV Proposed Solution

The SWU of the subsequent convolutional layer is the first block that requires major architectural modi-

fications and has the capability to sequentialise its output after receiving MMV_IN inputs at the same time. For this purpose, it needs to be able to take in MMV_IN wide inputs and give out MMV_OUT wide outputs, or it could be the point of sequentialisation, in which case it outputs data required for just one pixel. Thus, the existing SWU would need to be modified to a) have input and output ports of different sizes b) have inputs arrive at a different order than previously and hence to read and write in different data patterns.

The existing implementation of the SWU is investigated more deeply. It operates in two modes (Figure 5.4a) :

- Phase 1 - Write only phase :
  - Write input images into buffer until first *k_h* input image rows are completely filled.
- Phase 2 - Read and write phase:
  - Write new row of input images into the extra row of the input buffer.
  - Simultaneously read from the first *k_h* rows already written into in the required order.
  - Wait until the new row is completely filled and until all reads from the first KH rows are complete.
  - Switch the write pointer to next wrapped around block (row) of buffer.
  - Continue phase 2 until all rows of output image are obtained.

The SWU is implemented using HLS and stores one more row of line buffer than the minimum required to get an output pixel for the sake of elasticity. This leads to more BRAM utilisation than required. In FINN, since all network parameters are stored on-chip, BRAMs are a critical resource.

So, the additional functionality of reading wider input words than output words was implemented in RTL, which can implement the same functionality with BRAMs that are only *k_h* rows long. This is possible since, in RTL, it is possible to model simultaneous read and write processes with more precise control than using HLS. The RTL implementation also unified all read and write variations that are already possible using the HLS implementation into a single Sliding Window Unit.

The functional and performance requirements of the required SWU are listed below:

- FUNCTIONAL REQUIREMENTS
  - Width, Height, Depth of input feature map, number of bits per input word (N), *k_h* and *k_w*, the kernel dimensions, SH and SW, strides in horizontal and vertical direction, MMV_IN, MMV of previous convolution, MMV_OUT, MMV of current convolution, location of padding (top/bottom, left/write) and type of padding (pad wth zero or pad with repeat) must be parametrizable
  - The implementation should use a Simple Dual Port BRAM at its core, of minimum size given the parameters.
  - AXI stream IN and OUT to interface with rest of FINN using the standard communication protocol.
  - Reads and writes are independent, except reads do not proceed beyond the previously written memory locations
  - Customizable read scheduling
- PERFORMANCE REQUIREMENTS
  - Reading from buffer must be possible as early as possible
  - The SWU must avoid stalling of incoming data wherever possible
  - One read per cycle must be possible when data is available.

(a) HLS SWU

(b) RTL SWU

Figure 5.4: Buffer size for RTL and HLS SWU

5.3.1. Implementation Details of RTL SWU

An RTL design that satisfies the above requirements was developed. The primary structural difference in this implementation for the default STRIDE = 1 case is illustrated in Figure 5.4. The highlighted additional row in Figure 5.4a is used in HLS SWU to store incoming data when the existing data is read from the first three rows.

Read and write control and scheduling in the RTL implementation are more complex than HLS since read and write pointers have to operate on a tighter area of memory. Also, adding the MMV_IN parameter leads to a much more irregular pattern of reads that is complex to implement.

Features of the RTL SWU architecture are summarized below:

- A write counter tracks the location of data written into the buffer. Once minimum writes needed for the first pixel read is complete, reads are already possible. This in contrast to HLS implementation, where the full new image row has to be written into before we can read from the first position.
- Another counter (*pending_read_counter*) tracks the positions in buffer that have been read as many times as required and can be replaced with new incoming data. In Figure 5.4b, position A can be overwritten when the highlighted reads are complete. This number works out to $write\_counter\%(KH \times KW)$. In the design, the modulo function is implemented with a combination of wraparound counters that identify when *pending_read_counter* can be incremented. The wraparound counters serve 2 functions
  - identify when to increment the *pending_read_counter*
  - to calculate the read address
- Wraparound counters are only allowed to increment when an output stream transaction has occured at the previous read address.
- The SWU is implemented as a 4 stage pipeline, with the sequence of operations described in Figure 5.5. The reason for a 4 cycle pipeline is explained below: There exists a minimum 3 cycle latency between a handshake at the output port for data 1, and receiving the next data at the output port: 1 cycle for the first level wraparound counters to increment after recognising a transaction at the output port, 1 cycle for read address to be calculated based on these pointers and the read address to be registered on RAM, 1 cycle for data in the read address to arrive at the output of the RAM. With just this design it is not possible to obtain 1 read per cycle under all conditions (Figure 5.6a). Since the consumer block is not always ready to accept new data, the two cycle latency to get data at the RAM prohibits one read per cycle under all conditions. Hence, an additional register is added at the RAM output. With this, the pipeline is kept completely full at every instant (Figure 5.6b) and 1 read per cycle is obtained irrespective of the assertion of output *READY* signal. The register is able to hide the two cycle latency of getting RAM data.

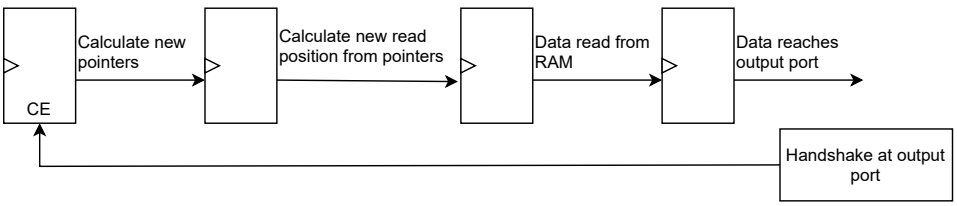
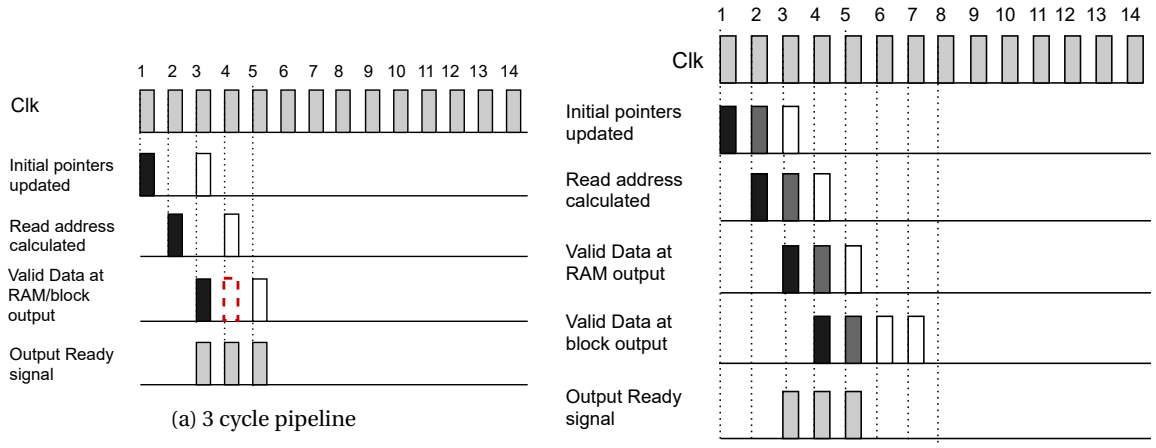


Figure 5.5: SWU Pipeline

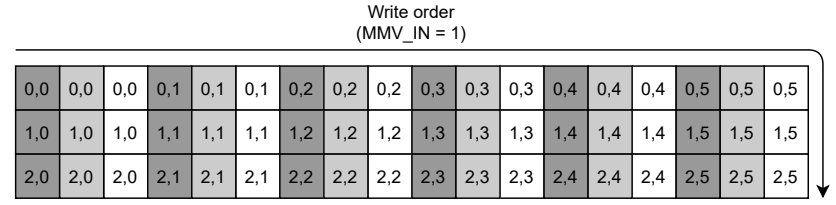


(a) 3 cycle pipeline

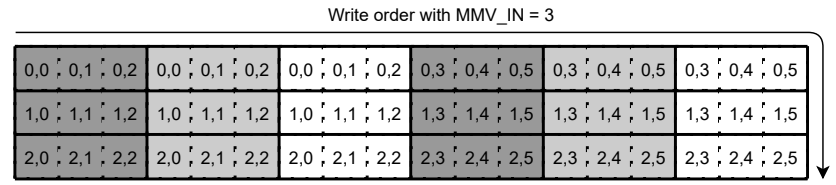
(b) 4 cycle pipeline

Figure 5.6: Pipelining in RTL SWU

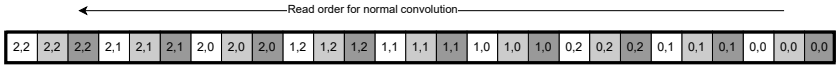
The read pattern from the SWU would vary depending on the padding, stride, SIMD, IFMChannels, MMV_IN (MMV of previous layer), MMV_OUT (MMV of current layer), type of convolution. Writing always happens in raster order, but the order of arrival also varies depending on the MMV_IN. Variations in read and write patterns are shown in Figure 5.7. Each shade represents a different channel of the pixel. Input and output port widths are the same for all configurations of the SWU, except for MMV_IN > 1. It has to take in MMV_IN wide inputs and give out a single pixel wide output. The core of such a SWU is instantiated as asymmetric RAM in hardware.



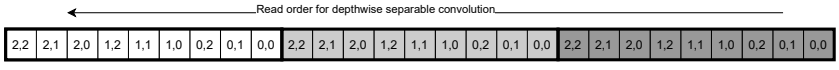
(a) Write Order MMV= 1



(b) Write Order MMV= 3



(c) Read Order for Normal Convolution



(d) Read Order for Depthwise Separable Convolution

Figure 5.7: Some read and write patterns in SWU

Asymmetric RAMs are blocks of memory that have different widths and depths in the input and output port. The advantage of using asymmetric RAMs in the MMV-closing SWU is that there is no point of explicit sequentialisation of the parallel inputs and thus, no added stalling of data. It enables receiving and storing $MMV\_IN \times InputPrecision$ wide inputs from the previous MMV enabled layer and gives out *InputPrecision*

wide data when MMV parallelism is not required from that layer. Even if `MMV = MMV_OUT` is required from this layer, data is stored in *MMV_OUT* asymmetric RAMs before reading *InputPrecision* wide data from each RAM.

For values of *STRIDE* > 1, it is not possible to meet the requirement of one output per cycle with the same buffer size. This can be explained using an example for STRIDE = 2. Generally, the number of reads from a block of data is around an order of magnitude greater than the number of writes. After the first output image row is calculated, the next output image row would require two new rows of the input image. However, data of only one row of the input image can be written into the oldest row of the buffer while we are reading from the same buffer at the same time. Hence, when all reads from the buffer are complete, the SWU would need to wait *IFMWidth* cycles before the next row of the output feature map can be output. To facilitate better scheduling, the buffer was modified to store an additional *STRIDE - 1* rows of input image. The existing HLS implementation also follows a similar approach.

6

Evaluation of proposed MMV modifications

The correct functionality of the RTL implementation of the SWU was tested and verified on the BNN CNV networks in simulation and in hardware (Pynq-Z2). The proposed MMV modification to the dataflow was also functionally verified on the CNV11 BNN. The BNN contains six SWUs that have different combinations of variable parameters. The SWU was also verified on the Mobilenet in simulation to account for some other configurations that were not present in the BNNs. This way mutiple combinations of defining parameters of the SWU were tested. First, this chapter compares the utilisation and performance of the RTL and HLS SWU. Then, it applies MMV as an additional parameter on a standalone MVAU node from a FINN BNN network and measures the throughput increase and the associated increase in resource utilisation.

6.1. RTL vs HLS implementations of SWU

6.1.1. Utilisation

An RTL description is expected to generate more compact hardware than HLS implementations for the same functions. In this section the two implementations of the sliding window unit are compared with different configurations. The HLS implementation refers to the existing group of sliding window unit modules from the FINN hlslib repository [1]. The RTL version is the SWU introduced in this thesis.

In RTL, intricate control over the design has to be exercised and the exact position of read and write counters needs to be tracked, to prevent stalling with only *kh* rows of data. Contrastingly, in HLS, the algorithm requires only the row number to be tracked. Also, to meet throughput requirements, the RTL SWU tracks the exact number of reads performed to calculate the number of new allowed writes. Hence, the control algorithm in RTL is more complex than the HLS SWU. Table 6.1 shows the hardware utilisation for some SWU configurations found in BNNs and Mobilenet. Despite more complex control, the RTL SWUs control requires lesser LUTs than the HLS SWUs.

6.1.2. Throughput

Throughput obtained from the SWUs were also compared since they follow different read and write scheduling. In the *CNV11* network, HLS SWUs were replaced by RTL SWUs of the same configurations and throughput of the entire network was compared. A 1.6× throughput improvement was observed with the RTL SWUs. This is because, the HLS implementation has to wait for an entire input image row to be written into the buffer before the corresponding output feature map row is read. On the other hand, the RTL implementation tracks the exact amount of new data obtained and an early start is obtained in the reading phase.

6.2. MMV Performance and Utilisation

To check the impact of the proposed modification to the FINN dataflow, especially the sequentialisation of data with the *MMV_IN* wide sliding window unit, an MVAU and its subsequent layers until the modified SWU

[1] https://github.com/Xilinx/finn-hlslib

Table 6.1: Utilisation Comparison : RTL vs HLS Sliding Window Unit

Module	Buffer Size	Implementation Style	Total LUTs	FFs	Memory LUTs	Control LUTs
SWU - 1	120 x 32	HLS	501	610	176	325
	90 x 32	RTL	440	244	132	308
SWU - 2	112 x 32	HLS	413	608	96	317
	84 x 32	RTL	350	202	88	262
SWU - 3	192 x 32	HLS	441	578	176	265
	144 x 32	RTL	418	241	132	286
SWU - 4	80 x 32	HLS	378	603	96	319
	60 x 32	RTL	259	158	44	215
SWU - 5	96 x 32	HLS	427	602	96	347
	72 x 32	RTL	202	165	88	114
SWU - 6	1120 x 24	HLS	1222	641	640	120
	896 x 24	RTL	1028	529	448	580
SWU - 7	444 x 128	HLS	2253	1374	1376	877
	333 x 128	RTL	1568	1034	1032	536

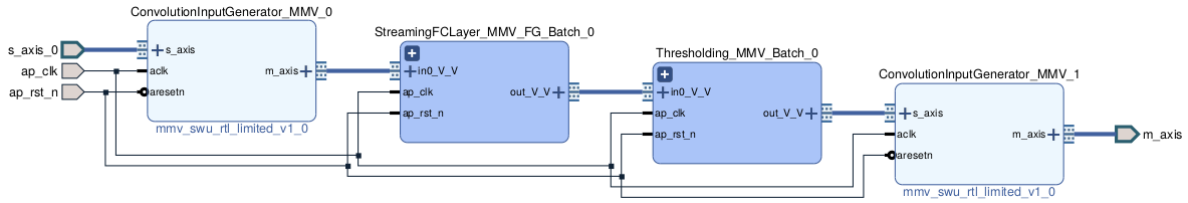


Figure 6.1: Block design to test MMV

were also simulated using Vivado. For a single MVAU node, *PE* was kept constant and *MMV* was increased. The simulated block diagram is shown in Figure 6.1. In this figure, *StreamingFCLayer_Batch_0* is the target of MMV parallelism.

Currently, in FINN, the throughput is limited by the slowest layer, that is, the layer that takes the most number of cycles to perform all computations for a single image. In this section, we verify if MMV can improve this factor for nodes that have already reached their peak performance using SIMD and PE parallelisms. Results of increasing MMV are tabulated in Table 6.2. The total number of cycles to completely process a single image were used as a measure of throughput. It was observed that the throughput obtained matches the theoretically expected performance increase. The associated utilisation increase is also reported. Note how the throughput improvement scales linearly with the set value of MMV with respect to the baseline design where MMV = 1. The resource utilisation also increases with increasing MMV.

Table 6.2: MMV Performance

MMV	LUTs	FFs	BRAMs	Total Cycles Improvement	LUT utilisation increase	FF Utilisation Increase
1	1801	1593	7.5	1 x	1 x	1 x
2	2976	2656	9	2 x	1.65 x	1.7 x
4	5287	4658	12	4 x	2.93 x	3.0 x

7

Conclusions and Future Work

In this thesis, the objectives of integrating and evaluating a fine-grained flow and utilising the fine-grain architecture to provide performance scaling were implemented and verified succesfully in simulation and on hardware. Chapter 2 introduced the problems associated with implementing NNs in hardware and how parallel architectures can be exploited, motivated the use of FPGAs for this purpose, discussed various hardware architectures currently present and drew focus to FINN, and identified two problems that will be the focus of this thesis. Chapter 3 motivated the focus of architectural improvement and derived the fine-grain architecture. Chapter 4 evaluated the fine-grained method on a standalone MVAU node, explored the effectiveness of the approach for different sizes of the node and discussed the advantages obtained. The chapter also explores the advantage of parallel processing on the fine-grained (FG) methodology. After evaluation on a standalone node, some FINN supported accelerator networks were also implemented with FG and the results were discussed and explained based on the results obtained from the standalone node. Chapter 5 explored and described methods to introduce a general MMV capability to FINN, and proposed a method to modify the dataflow of the MMV Modified portion of the structure to include this modification. It also highlighted the characteristics of the architecture of a modified Sliding Window Unit introduced for this purpose. Chapter 6 evaluated the proposed MMV solution on a sample BNN network. The new SWU was verified first, then the module was integrated with all layers that are modified for MMV and the design was functionally verified.

7.1. Conclusions

How can the construction methodology of FINN architectures be modified to improve their build times? How large is the resource overhead induced by this approach and how can it be minimised?

The construction methodology of MVAU, the primary computational structure in FINN implemented architectures, was modified to follow a fine-grained approach. The fine-grain (FG) approach is generic and can be directly applied to any network supported by FINN. To get the most advantage from FG, all processing element architectures are made as identical as possible, so that they can be reused. FG was evaluated first on a standalone MVAU node, and then on entire accelerator networks. From a standalone analysis of a single MVAU node, it was observed that this method can give an improvement for all values of the number of processing elements (PEs). Benefits offered by fine-grain on entire accelerator networks are dependent on the number of MVAU nodes in the network, their PE values, and the number of CPUs available to parallelise the synthesis process. FG always adds extra jobs to the synthesis process when the baseline design is implemented with embedded thresholds. The number of extra jobs scales with MVAU nodes and their effects can be amortized by running synthesis with more threads. Moreover, to obtain an overall advantage, the benefits of synthesising a smaller PE in FG must be able to compensate for the cost of synthesising the additional modules. For a single node, the benefit scales with the PE value of the MVAU and the cost also increases with PE value. Nevertheless, Vivado, the synthesis software, primarily restricts the benefits possible by FG. Firstly, it expends a significant amount of time to invoke a cache retrieval, almost matching the time that it takes for actual synthesis. Second, synthesis of simple data movement infrastructure consumes as much time as the synthesis of the actual computational unit. This indicates a significant fixed build time cost of a synthesis job that arises from housekeeping tasks such as loading the project part and launching child processes. By optimizing the software to remove these inefficiencies, full benefits of this method can be extracted. In effect, FG

removes the bottleneck in the size of synthesis jobs and encounters a bottleneck in the software that invokes these jobs. From analysis of actual synthesis times on some network accelerators, it was observed that FG can provide up to 12× speedup in HLS Synthesis times, up to 2× speedup in overall synthesis times with little to no negative impact on resource utilisation. FG is most beneficial in networks with a smaller number of MVAU nodes, each with large PE values and synthesis run on large number of threads.

How can FINN-generated dataflow architectures be modified to introduce an additional level of performance scaling?

A general method to perform Multiple Matrix Vector (MMV) multiplications and provide parallel computation of multiple output pixels in a convolutional layer was proposed and implemented. MMV cannot be implemented by modifying only the layer that requires throughput improvement. A general MMV method to increase performance capabilities requires a more extensive modification to downstream layers to seamlessly sequentialise the parallelly produced MMV pixel wide outputs. The part of the dataflow of some networks which cannot meet throughput requirements with the available modes of parallelism was modified with the proposed MMV implementations and tested. It was effectively able to complete its execution cycles *MMV* times faster at the cost of less than *MMV* times additional resources. A new SWU that can sequentialise the parallelly produced MMV-pixel wide outputs was developed using Verilog. This module was necessary to integrate the MMV-modified part of the dataflow with the downstream layers that do not need MMV parallelism. The RTL implementation of the SWU is able to efficiently unify multiple existing HLS implementations and introduce functionality needed for MMV. It is more efficient, both in terms of control and memory resources and scheduling reads and writes.

7.2. Future Work

The backend software, Vivado, needs to be optimised to extract complete benefits from the fine-grained approach. Further, resource costs as well as timing costs can benefit from using simpler interconnect and the resulting simpler data movement infrastructure, instead of the AXI Stream interface. Another improvement could be to reduce the number of additional jobs by keeping the threshold unit embedded for fine-grain approach and modifying the MVAU to take in additional threshold stream interface from an external threshold memory instead. This way, synthesis of a separate threshold unit can be avoided. An external threshold unit is disadvantageous since it leads to synthesis of wider output port for MVAU and input port for the threshold unit and the interface costs more resources. Thresholded values have much lower datawidth and this cost can be avoided by embedding the unit. Also, integrating the threshold unit will reduce the number of jobs that increase proportionally with the number of MVAU nodes and can remove the fixed synthesis time overhead of new jobs and of synthesising the additional wider ports.

For the MMV method, limitation to the current approach is that Vivado is able to infer asymmetric RAMs, which are needed for the closing sliding window unit, only for certain ratios of input-output port widths. To make the method more generic, a wrapper module needs to be created to force Vivado to infer an asymmetric RAM for all values of MMV. Moreover, read patterns from the SWU are complex to implement. Other transformations like the Winograd transform [20] can be used to simplify the pattern and reduce the number of computations in a convolution, allowing for efficient resource utilisation when implemented in hardware. In dataflow architectures like FINN, area is the limiting factor in maximising throughput, so this can also have a positive impact on overall performance of the network.

Bibliography

[1] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Sérot, and François Berry. Accelerating CNN inference on fpgas: A survey. *CoRR*, abs/1806.01683, 2018. URL http://arxiv.org/abs/1806.01683.

[2] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Sérot, and François Berry. Accelerating CNN inference on fpgas: A survey. *CoRR*, abs/1806.01683, 2018. URL http://arxiv.org/abs/1806.01683.

[3] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'Brien, and Yaman Umuroglu. FINN-R: an end-to-end deep-learning framework for fast exploration of quantized neural networks. *CoRR*, abs/1809.04570, 2018. URL http://arxiv.org/abs/1809.04570.

[4] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Muhammad Shafique, Guido Masera, and Maurizio Martina. An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks. *Future Internet*, 12(7), 2020. ISSN 1999-5903. URL https://www.mdpi.com/1999-5903/12/7/113.

[5] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. 10 2006.

[6] 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. ISBN 978-3-319-11179-7.

[7] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016. URL http://arxiv.org/abs/1602.02830.

[8] Steven Derrien and Sanjay Rajopadhye. Loop tiling for reconfigurable accelerators. pages 398–408, 08 2001. ISBN 978-3-540-42499-4. doi: 10.1007/3-540-44687-7_41.

[9] Quentin Ducasse, Pascal Cotret, Loïc Lagadec, and Robert Stewart. Benchmarking quantized neural networks on fpgas with FINN. *CoRR*, abs/2102.01341, 2021. URL https://arxiv.org/abs/2102.01341.

[10] Mohammad Ghasemzadeh, Mohammad Samragh, and Farinaz Koushanfar. Resbinnet: Residual binary neural network. *CoRR*, abs/1711.01243, 2017. URL http://arxiv.org/abs/1711.01243.

[11] Song Han, Huizi Mao, and William Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. 10 2016.

[12] 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.

[13] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR*, abs/1712.05877, 2017. URL http://arxiv.org/abs/1712.05877.

[14] Petar Jokic, Stephane Emery, and Luca Benini. Binaryeye: A 20 kfps streaming camera system on fpga with real-time on-device image recognition using binary neural networks. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pages 1–7, 2018. doi: 10.1109/SIES.2018.8442108.

[15] Xu Kang, Bin Song, and Fengyao Sun. A deep similarity metric method based on incomplete data for traffic anomaly detection in iot. *Applied Sciences*, 9(1), 2019. ISSN 2076-3417. doi: 10.3390/app9010135. URL https://www.mdpi.com/2076-3417/9/1/135.

[16] Alireza Khodamoradi, Kristof Denolf, and Ryan Kastner. S2n2: A fpga accelerator for streaming spiking neural networks. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 194–205, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382182. doi: 10.1145/3431920.3439283. URL https://doi.org/10.1145/3431920.3439283.

[17] Samuel Kriman, Stanislav Beliaev, Boris Ginsburg, Jocelyn Huang, Oleksii Kuchaiev, Vitaly Lavrukhin, Ryan Leary, Jason Li, and Yang Zhang. Quartznet: Deep automatic speech recognition with 1d time-channel separable convolutions, 2019.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017. ISSN 0001-0782. doi: 10.1145/3065386. URL https://doi.org/10.1145/3065386.

[19] Xilinx Research Labs. Finn¶, 2020. URL https://finn.readthedocs.io/en/latest/.

[20] Andrew Lavin. Fast algorithms for convolutional neural networks. *CoRR*, abs/1509.09308, 2015. URL http://arxiv.org/abs/1509.09308.

[21] Ji Li, Ao Ren, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, and Yanzhi Wang. Towards acceleration of deep convolutional neural networks using stochastic computing. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 115–120, 2017. doi: 10.1109/ASPDAC.2017.7858306.

[22] Xin Li, Yujie Cai, Jun Han, and Xiaoyang Zeng. A high utilization fpga-based accelerator for variable-scale convolutional neural network. In *2017 IEEE 12th International Conference on ASIC (ASICON)*, pages 944–947, 2017. doi: 10.1109/ASICON.2017.8252633.

[23] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. ICML'16. JMLR.org, 2016.

[24] Janardan Misra and Indranil Saha. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing*, 74(1):239–255, 2010. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2010.03.021. URL https://www.sciencedirect.com/science/article/pii/S092523121000216X. Artificial Brains.

[25] Nour Moustafa and Jill Slay. Unsw-nb15: a comprehensive data set for network intrusion detection systems (unsw-nb15 network data set). In *2015 Military Communications and Information Systems Conference (MilCIS)*, pages 1–6, 2015. doi: 10.1109/MilCIS.2015.7348942.

[26] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. Can fpgas beat gpus in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 5–14, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450343541. doi: 10.1145/3020078.3021740. URL https://doi.org/10.1145/3020078.3021740.

[27] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees Vissers, Joseph Zambreno, and Phillip Jones. Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels. 05 2019. doi: 10.1109/ICESS.2019.8782524.

[28] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. ISSN 0893-6080. doi: https://doi.org/10.1016/j.neunet.2014.09.003. URL https://www.sciencedirect.com/science/article/pii/S0893608014002135.

[29] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv 1409.1556*, 09 2014.

[30] Wonyong Sung, Sungho Shin, and Kyuyeon Hwang. Resiliency of deep neural networks under quantization. 11 2015.

[31] Yaman Umuroglu and Magnus Jahre. Streamlined deployment for quantized neural networks. *CoRR*, abs/1709.04060, 2017. URL http://arxiv.org/abs/1709.04060.

[32] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Heng Wai Leong, Magnus Jahre, and Kees A. Vissers. FINN: A framework for fast, scalable binarized neural network inference. *CoRR*, abs/1612.07119, 2016. URL http://arxiv.org/abs/1612.07119.

[33] Tobias Weyand, Ilya Kostrikov, and James Philbin. Planet - photo geolocation with convolutional neural networks. In *European Conference on Computer Vision (ECCV)*, 2016.

[34] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. 02 2009.

[35] Xilinx. Understanding fpga architecture, 2020. URL https://www.xilinx.com/html_docs/xilinx2017_2/sdaccel_doc/topics/devices/con-fpga-architecture.html.

[36] Rikiya Yamashita, Mizuho Nishio, Richard Do, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, 9, 06 2018. doi: 10.1007/s13244-018-0639-9.

[37] 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, page 161–170, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333153. doi: 10.1145/2684746.2689060. URL https://doi.org/10.1145/2684746.2689060.