Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science

Master's thesis

Integration of a convolutional neural network for speech-to-text recognition in an FPGA compiler flow

Author:
M. Mrahorović

Proposer / Supervisor:
dr. ir. Z. Al-Ars

Supervisor:
ir. J. Petri-König

Chair:
dr. ir. Z. Al-Ars

Jury:
dr. ir. S. Verwer
ir. J. Petri-König
dr. ir. Y. Umuroglu

Integration of a convolutional neural network for speech-to-text recognition in an FPGA compiler flow

by

Mirza Mrahorović

to obtain the degree of

Master of Science in Computer Engineering

at the Delft University of Technology,
to be defended publicly on Friday September 24, 2021 at 09:00 AM.

Student number:	4596536	
Submission date:	September 15, 2021	
Thesis committee:	dr. ir. Z. Al-Ars,	TU Delft, supervisor
	ir. J. Petri-König,	TU Delft, supervisor
	dr. ir. S. Verwer,	TU Delft
	dr. ir. Y. Umuroglu,	Xilinx Research Lab

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

Deep Neural Network (DNNs) have increased significantly in size over the past decade. Partly due to this, the accuracy of DNNs in image classification and speech recognition tasks has increased as well. This enables a great potential for such models to be applied in real-world applications. However, due to their size, the compute and power requirements are often too large to deploy these models on edge devices. This prohibits applying such models within a rich field of application demanding high-throughput and real-time execution.

Deploying quantized DNNs on Field Programmable Gate Arrays (FPGAs) overcomes this problem. FPGAs are well known for their low-latency, high-throughput, and low-energy capabilities. However, creating hand-tuned FPGA designs requires expert-level knowledge of the underlying hardware domain. Especially for mathematicians or software engineers that develop new quantized DNNs, but also for experienced hardware designers that want to implement a large DNN on an FPGA, the implementation burden is often too large to reap any practical benefits from accelerating the application on an FPGA.

The open-source FINN compiler, introduced by Xilinx Research Lab, provides an excellent bridge between the software and hardware domain by allowing quantized DNN inference FPGA accelerators to be generated from a high-level description of the quantized DNN in the widely adopted open-source ONNX format. Due to lower-level implementation details being abstracted away, the question is how this affects the performance of the generated accelerator.

This work examines whether FPGA implementations of CNN-based models for speech-to-text inference can be generated automatically by means of FINN. For this purpose, a sub state-of-the-art CNN for speech-to-text recognition, named QuartzNet, is targeted for FPGA acceleration. To achieve this, extensions to the FINN compiler are proposed to enable generating 1D CNN inference accelerators for FPGAs. Furthermore, a proof-of-concept FPGA accelerator of a quantized QuartzNet model is implemented by means of FINN. Compared to a high-end CPU device, the proposed FPGA accelerator achieves $7.7\times$ higher throughput and $8.2\times$ lower latency for a speech recognition inference task. Compared to a high-end GPU device, the proposed FPGA accelerator improves the energy efficiency by 6.8% at the expense of lower throughput and higher latency.

By generating an FPGA accelerator for a quantized version of the QuartzNet model, this work bridges the software and hardware domain by showcasing how a trained CNN in the software domain can be transformed to create a high-throughput, low-latency, and energy-efficient FPGA accelerator with a fraction of the design effort required compared to constructing a handwritten RTL implementation.

Preface

This work, even though a single author is formally mentioned, marks the culmination of nine months of effort which could not have been achieved without the support of others. In particular, my lovely parents, my brother, my grandmother, and my uncle, for being the drive behind my motivation during the last 5 years (and beyond), as well their immeasurable patience, love, and care. From the very first day as an undergraduate student, where I was overwhelmed by the workload, to the very last moment as a graduate student (and beyond). Another thanks goes out to my friends along the journey for the table football breaks and many other joyful moments.

Finally, a special thank you to express my gratitude to my supervisors who have supported me immensely, taught me valuable life lessons, and were always available to help and hear me out: Zaid Al-Ars, Jakoba Petri-König, and Yaman Umuroglu.

Mirza Mrahorović
~~Delft~~ Bedroom in Rotterdam, September 2021

Contents

List of Figures

List of Tables

List of Acronyms

1D	One-Dimensional
2D	Two-Dimensional
ASIC	Application-Specific Integrated Circuits
ASR	Automatic Speech Recognition
AXI	Advanced Extensible Interface
BN	Batch Normalisation
BRAM	Block Random-Access Memory
CNN	Convolutional Neural Networks
CPU	Central Processing Unit
CTC	Connectionist Temporal Classification
CU	Compute Unit
DF	Dataflow
DMA	Direct Memory Access
DNN	Deep Neural Network
DSP	Digital Signal Processing
FC	Fully Connected
FF	Flip-Flops
FIFO	First In, First Out
FP	Floating Point
FPGA	Field Programmable Gate Array
GMM	Gaussian Mixture Model
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
HDL	Hardware Description Language
HLL	High-Level Language
HLS	High-Level Synthesis
HMM	Hidden Markov Model
im2col	Image to Column (algorithm)
Im2Col	Image to Column (operator)
INT	Integer
IO	Input/Output
IP	Intellectual Property
LM	Language Model
LSTM	Long Short Term Memory

LUT Look-Up Table
LUTRAM LUT (Distributed) Random-Access Memory
LUTROM LUT Read-Only Memory
MAC Multiply-Accumulate
MLP MultiLayer Perceptron
MPE Matrix of Processing Elements
MVAU Matrix Vector Activate Unit, StreamingFCLayer_Batch
NN Neural Network
OffC Off-Chip
OnC On-Chip
PE Processing Engine
RAM Random-Access Memory
ReLU Rectified Linear Unit
RNN Recurrent Neural Network
ROM Read-Only Memory
RTL Register-Transfer Level
SDWC StreamingDataWidthConverter
SIMD Single Instruction, Multiple Data
SLL Super Long Line
SLR Super Logic Region
SWU Sliding Window Unit, ConvolutionInputGenerator
URAM Ultra Random-Access Memory
VVAU Vector Vector Activate Unit, Vector_Vector_Activate_Batch
WER Word-Error-Rate
XO Xilinx Object

1 Introduction

1.1. Context & problem statement

With the advent of faster compute infrastructure and more training data becoming available, Deep Neural Networks (DNNs) have started to become increasingly useful [28]. DNNs have shown to beat humans in their own designed games, such as chess and Go [59], and have for example shown to surpass human-level accuracy on image classification tasks [57, 60]. For speech recognition tasks, DNNs have also become increasingly more accurate.

These successes are partly attributed to models becoming larger and larger. For example, a state-of-the-art speech-to-text recognition network employs more than 1 billion floating point parameters [80]. As models are growing in size, their compute and storage requirements, as well as power consumption, are increasing likewise. This makes deploying such models on edge devices, that have limited compute and memory and demand low-power and low-latency model execution, infeasible. Offloading such operations to the cloud is also not an option if low-latency inference is required, which is for example the case for real-time applications. This severely limits the applicability of large DNNs.

Researchers have been addressing the issue to make DNNs more power-efficient [17]. One popular method to achieve this is called quantization; it has been shown that lowering the bit precision of weights and intermediate results in a DNN has a minor impact on the accuracy of the model [5, 81, 82]. This method reduces the memory footprint and allows for faster and more power-efficient low-precision arithmetic, potentially allowing the deployment of such models to a whole new field of devices and applications.

Clearly, this only has a theoretical advantage if it cannot be applied in practice, as conventional compute infrastructure, such as CPUs/GPUs, are not able to fully reap the benefits of lower precision arithmetic. Considering alternative hardware platforms, Field Programmable Gate Arrays (FPGAs) are highly suited for such tasks. As opposed to CPUs/GPUs, FPGAs allow users to implement custom bit-width arithmetic and memory hierarchy tailored to meet user-defined performance targets. Each layer in the DNN can be executed at the moment data is available, allowing for a fully streamed dataflow-style design achieving high-throughput, low-latency and a power-efficient accelerator for the DNN of interest. FPGAs offer a high level of flexibility and offer a cost affordable solution, as opposed to Application-Specific Integrated Circuits (ASICs).

Again, this only implies a theoretical advantage if the architecture of the DNN maps efficiently to the streaming dataflow paradigm and available resources on an FPGA. Convolutional Neural Networks (CNNs) have a favourable property that potentially allows the design and implementation of high-performance and energy-efficient FPGA accelerator. CNNs are highly parallelisable due to the inherent matrix multiplication that forms the main building block of such models, meaning that it matches well to the acceleration profile of FPGA devices. On top of that, quantized CNNs have a significantly reduced memory footprint, allowing to store all the weights onto on-chip memory, which implicitly reduces the latency and increases power efficiency. Hence, quantized CNNs on FPGAs could lead to low-latency and low-power inference accelerators, which allows the increasingly larger and more accurate CNNs to be deployed in practical use-cases on edge devices that require real-time and power-efficient inference execution.

However, the development efforts and design space exploration for an FPGA are considerably larger compared to creating a CPU or GPU accelerator. In fact, for very large models, it is arguably infeasible to implement such a model by hand in RTL on an FPGA. Academia and industry have been addressing this issue by introducing tools that abstract low-level hardware details away, which effectively lowers the design effort required for FPGA devices. The FINN compiler is an excellent example thereof [7]. The FINN compiler is an open-source framework for developing high-throughput and low-latency DNN inference accelerators on a wide variety of FPGA devices. Every single layer can be time-multiplexed to ensure that the design meets a user-defined latency or throughput target. This allows for large portability of the custom-designed DNN model; low-latency and high-throughput, e.g. for real-time inference applications, can be targeted at the cost of higher resource utilisation, or area-efficient designs can be created to target a smaller (cheaper) FPGA device. Compared to implementing a network in RTL, such higher-level tools generally result in less optimal solutions in terms of throughput, latency, and area and power efficiency. However, the design effort becomes significantly smaller.

In this work, the focus is on developing an FPGA accelerator for a speech-to-text inference model. The potential advantage of an FPGA accelerator over a CPU or GPU implementation is that the FPGA-based solution could be more energy-efficient and even result in higher throughput and lower latency inference execution. The network of interest is a quantized version of the QuartzNet model developed by the Xilinx Research Lab [16]. In order to lower the design effort to make the implementation feasible within the time span of this project, as well as aiding reproducibility, the open-source FINN framework is used and extended to design and implement the FPGA accelerator.

Research questions
The research questions for this work can be summarised as follows:

1. Can we automatically generate speech-to-text CNN inference FPGA accelerators using a data-flow compiler for DNN inference on FPGAs?
2. What are the throughput and latency characteristics of such a generated FPGA accelerator?
3. How does the generated FPGA accelerator compare to mainstream CPU and GPU-based implementations of the CNN inference model?

1.2. Challenges & contributions

In order to create an FPGA accelerator for a speech-to-text CNN, this project has been divided into two parts. The first part is focused on extending the FINN framework. The FINN framework currently only supports DNNs that operate on 2D square input data. For speech-to-text networks, the input is typically a 1D audio waveform. This means that the architecture of the DNN is tailored to operate on 1D tensors instead of 2D tensors. Hence, the FINN framework needs to be extended to enable creating efficient FPGA accelerators for 1D DNNs. In order to achieve this, the challenge is framed more broadly; the FINN framework needs to be extended to enable creating efficient FPGA accelerators for DNNs operating on non-square 2D input images. This would allow a wide variety of networks to be mapped onto an FPGA by means of the FINN framework. For my specific work, note that a model that operates on a 1D input can be considered as a model operating on a non-square 2D input with one dimension set to the extreme case of 1.

The second part is focused on using the FINN framework to create a high-performance and energy-efficient FPGA accelerator for a DNN performing a speech-to-text inference task. The specific network that is targeted is QuartzNet, which is a CNN. Due to the size of the network, the challenge lies in ensuring that the model can fit on an FPGA. With the FINN framework, an extensive design space exploration can be performed to assure that this condition is met. By doing so, this work bridges the software and hardware domain by showcasing how a trained CNN in the software domain can be transformed to create a high-throughput, low-latency, and energy-efficient FPGA accelerator with a fraction of the design effort required compared to constructing a handwritten RTL implementation. Most importantly, it showcases how this bridge is accessible to both hardware and software engineers due to a lowered design burden offered by the FINN framework.

Contributions
To summarise, the contributions of this work are:

1. Extensions to the FINN compiler to support non-square and 1D CNNs. This enlarges the set of networks and application domains that can potentially be accelerated with FINN.

2. Design, implementation, and evaluation of a proof-of-concept FPGA accelerator for a CNN for a speech-to-text inference application. This work showcases the largest topology for a speech-to-text CNN ever implemented on an FPGA and the largest topology ever mapped on an FPGA by means of FINN. Compared to a high-end CPU device, the proposed FPGA accelerator achieves $7.7\times$ higher throughput and $8.2\times$ lower latency. Furthermore, the proposed FPGA accelerator improves the energy efficiency by 6.8% compared to a GPU-based implementation at the expense of lower throughput and higher latency.

The majority of the extensions mentioned in 1) are available in the open-source GitHub repository of FINN [36, 37, 40].

1.3. Thesis outline

The rest of this thesis is organised in the following way. Chapter 2 discusses in more detail what speech-to-text neural networks are, why and how the quantization technique is useful in such networks, and how these two combined can be suited for FPGA implementation. A brief overview of FPGA accelerators for speech-to-text inference applications is also presented. Furthermore, a detailed discussion of the QuartzNet model proposed by NVIDIA and the quantized QuartzNet model developed by Xilinx Research Lab is presented, as well as an in-depth discussion of the FINN framework. Based on this, the challenges for the QuartzNet model in FINN are summarised. Chapter 3 presents in more detail how the FINN framework needs to be adjusted to support the QuartzNet model. Chapter 4 presents a design space exploration of various layers encountered in the QuartzNet model. More precisely, the effect of time-multiplexing specific layers from the QuartzNet model on compute and memory resource utilisation is studied. Chapter 5 profiles and compares the implementation of the quantized QuartzNet model on a CPU, GPU, and FPGA device. As last, Chapter 6 concludes the work and presents future research directions.

2

Background

This chapter will cover topics that are required to understand the proposed extensions to the FINN compiler as well as the design choices made for QuartzNet. First, an introduction to speech-to-text neural networks and convolutional neural networks is given in Section 2.1. After having identified the trend of models getting increasingly larger, a popular model compression technique named quantization is discussed in Section 2.2. After having established that Field Programmable Gate Arrays (FPGAs) provide excellent opportunities for implementing custom bit-precision logic, a brief introduction on FPGAs as well as one specific target board, the Alveo U250, is presented in Section 2.3. Subsequently, a brief introduction on related work targeting FPGA inference accelerators for speech-to-text recognition tasks is given in Section 2.4. Next, we will present more details of the architecture of the QuartzNet model in Section 2.5. As the model is too large and complex to design an FPGA accelerator by hand, we shall motivate why the FINN compiler is an excellent tool to create an FPGA accelerator and also discuss the internal details of the tool in Section 2.6. After that, Section 2.7 presents an overview of how the main building block of QuartzNet, a convolution, is mapped to FINN custom operators and a brief overview of the underlying hardware component is given. Finally, the design and implementation challenges for enabling the FINN compiler to create an FPGA accelerator for the QuartzNet model is discussed in Section 2.8.

2.1. Neural networks

A neural network (NN) architecture can be seen as a function family $f(x)$ that produces some output given an input x [6]. The architecture describes the sequence of operations used as well as the configuration of those operations. The basic computational structure of a neural network is the perceptron introduced in 1958 [56]. A schematic of the perceptron model is shown in Figure 2.1. The nodes/vertices are referred to as neurons and the edges are referred to as synapses. The synapses have a certain weight associated with them. The output of each neuron consists of a weighted sum of neurons plus an additional bias term followed by an activation function, as shown by the highlighted box. The weights are typically trained during a phase referred to as training. The activation function is typically a simple mathematical operation, such as a Rectified Linear Unit (ReLU) activation function. This operation will be introduced for the specific network of interest in Section 2.5.2. The output of a neuron is also referred to as the (output) activation.

Typically, neural network architectures consist of a sequence of layers composed of neurons as shown on the right side of Figure 2.1. The first and last layers are called the input and output layers, respectively. The layer in the middle is referred to as hidden layers. Essentially, each layer transforms the input, also called the input feature (map), into an intermediate representation, also referred to as the output feature (map). The so-called input image can be, for example, a 2D vector representing an image or a 1D vector representing an audio waveform, and is typically composed of several channels. These channels are also referred to as features of the image; such as the RGB channels in an image. The final layer is typically fully connected, meaning that all of the neurons in the previous hidden layer are connected to all of the neurons in the final output layer. In the end, an output is produced that matches with the scope of the task; in image recognition tasks, a label is produced that relates to the

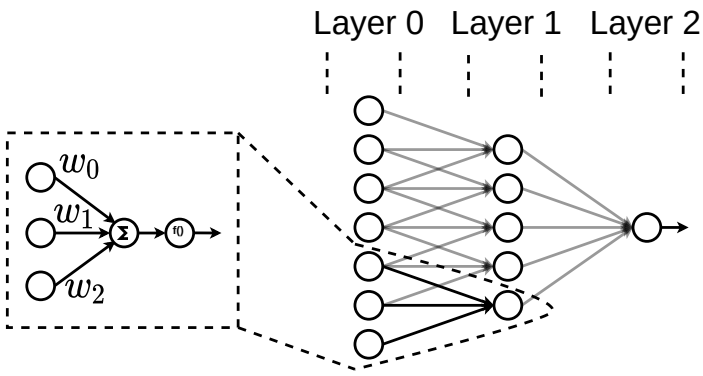


Figure 2.1: High level schematic of a perceptron model and simple neural network.

supplied input image.

The configuration of a layer relates to specific properties of the operations; e.g. convolutions can have different filter shapes. A neural network model describes also the parametrisation of the neural network architecture, i.e. $f(x;W)$ [6]. The inference phase refers to deploying a model with fixed parameters to perform a specific classification task. Note that the following discussions will be focused on the inference aspects of NNs.

2.1.1. Speech-to-text deep neural networks

Deep learning is a specific type of machine learning where the neural network is composed of many consecutive layers, hence the name 'deep', that are able to extract relevant features from raw inputs and subsequently combine those features to produce some output. Deep neural networks (DNNs) have become increasingly useful over time as the amount of training data has increased and computer infrastructure has improved [28]. This is partly attributed to the increasing size of deep neural network models and is in accordance with one of the pillars deep learning is based on; namely, the philosophy of connectionism [8, 28]. A single biological/artificial neuron might not express any kind of intelligence, but a large pool of connected neurons might create an intelligent system. Deep neural networks have proven to surpass human capabilities in various tasks. A DNN named AlphaZero is able to achieve superhuman performance in the games of chess, shogi and Go [59], and GoogLeNet is even able to surpass a human annotator, albeit a not thoroughly trained one, on a 1000-label image classification task [57, 60]. Within natural language processing, deep learning models are also achieving increasingly lower word-error-rates (WERs). In this work, our focus is on models that perform Automatic Speech Recognition (ASR), which is the task of recognising words from audio and converting them to text. More formally, given an acoustic sequence X and a linguistic sequence y, the goal of ASR is to create a function that maximises the conditional distribution (P) that relates the acoustic inputs X to the linguistic target y, as given in Equation (2.1) [28].

$$f_{ASR}(X) = \arg\max_{y} P(y|X = X) \tag{2.1}$$

In other words, the task is: given a sequence of words, what is the most likely string of characters. In this work, the focus is on creating an FPGA accelerator for a model performing inference for an ASR task.

Since 1980, systems combining Hidden Markov Models (HMMs) and Gaussian Mixture Models (GMMs) were used to solve this task [28]. However, with the advent of deeper and larger models, neural networks were able to achieve state-of-the-art accuracy results on ASR tasks. There are many variants to DNN models, such as Long Short Term Memory (LSTM) models [24], other types of Recurrent Neural Networks (RNNs) and variants thereof [79], Transformer models [64], Convolutional Neural Networks (CNNs) such as Jasper and QuartzNet [34, 43], or combinations of CNNs and Transformer models, so-called Conformer models [18]. Discussing the architectural details of each model goes beyond the scope of this work, but one particular type of DNN is of interest: CNNs [41]. These networks are

similar to MultiLayer Perceptrons (MLPs) meaning that the network consists of a group of neurons with learnable weights and biases. Each of these neurons receives an input image, perform a dot product with a weight value and apply an activation function to produce the output. The main difference for CNNs is that MLPs are fully connected, while CNNs are not. CNNs are characterised by the convolution operator.

Convolutional Neural Networks

A convolution operation in deep learning can be thought of as a filter that applies a linear transformation on the input pixels within its local field and as it slides over the input image, it creates a new representation of the image. A visual example is given in Figure 2.2; the input image (IFM) is of size $[IFM_H, IFM_W]$ with IFM_C channels and there are OFM_C kernels related to the number of output channels, where each kernel is of size $[K_H, K_W]$ with IFM_C channels. The output value, the red box in the right matrix, is obtained by a sum of the dot products of each convolutional operation. Note that a 1D convolution is obtained by setting IFM_H and K_H or IFM_W and K_W to 1.

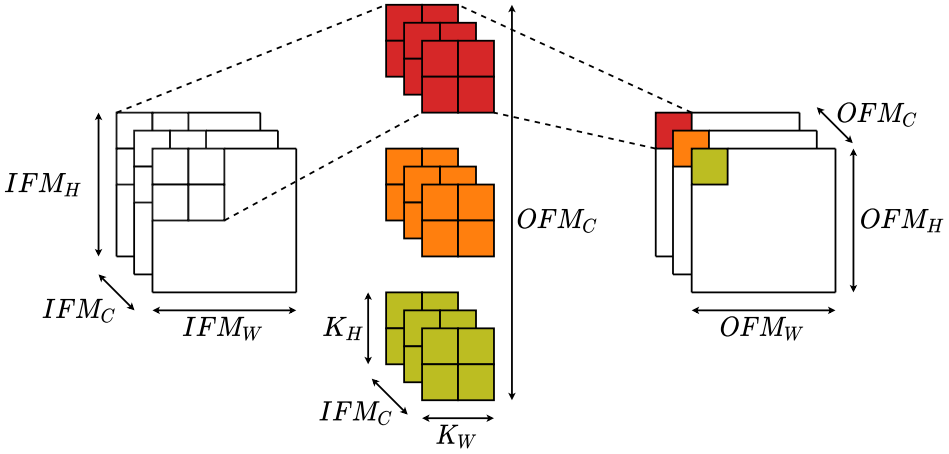


Figure 2.2: Convolution operation between an input image of 3 channels, kernel with 3 channels, producing a single output value.

For output location (i, j), the 2D convolution operation with a kernel of size K_h by K_w can be expressed as a discrete convolution operation as shown in Equation (2.2), where OFM, IFM and K represent the output image, input image, and kernel respectively.

$$OFM(i,j) = \sum_h \sum_w IFM\,(i-h, j-w)\,K(h,w) \tag{2.2}$$

The important point to notice is that a convolution can be expressed as a matrix multiplication if the weight matrix is restructured into a Toeplitz matrix [28]. However, as this matrix is very sparse, this particular computation cannot be done efficiently. In Section 2.6.2, a more efficient way of performing a convolution operation is explained.

This work is focused on QuartzNet, which is a CNN. The convolution operation has several favourable properties with respect to fully connected layers which are worth mentioning [28]:

1. Convolutions have sparse interactions and rely on parameter sharing. Each neuron in a hidden layer is connected to a limited set of neurons from the previous layer and the weights of these connections are all dictated by the kernel. The practical advantage of having fewer parameters to train is that the final model requires fewer parameters to be stored in hardware. Secondly, sparse interactions also imply fewer computational operations to produce the output of a neuron. A more theoretical advantage is that fewer parameters reduces the complexity of the model and could reduce the chance of overfitting.

2. Convolutions are equivariant to translation. In the case of images or time-series data as input, if a certain shape is moved in space or a certain sound event occurs earlier or later in time, the same representation will appear at the output of the convolution; i.e. certain corners or sounds can be detected at different points in space or time respectively. In the case of ASR, this property can provide robustness against speaker and environment variance, as shown in [3], albeit with a different type of weight sharing technique compared to those used for the convolutions in QuartzNet.

Besides the convolution operator, several other operators can be found which are used for the activation function and normalisation respectively. The activation function can be linear or non-linear, where a well-known example is the ReLU. Regarding normalisation, Batch Normalisation (BN) is also commonly found in DNNs [29]. These operators are explained later when the QuartzNet model is introduced.

Accuracy of ASR models

There are essentially two objectives when using a DNN; training and inference. Training relates to the process of estimating the model parameters to optimise a certain loss function. Inference relates to the actual deployment of the model, i.e. using the model with the learned parameters to perform a particular task. As mentioned before, in this work the focus is on inference. To evaluate and compare the accuracy of different models in ASR tasks, standardised benchmark datasets are used. There exist many different benchmark datasets to train and validate a model on, such as LibriSpeech [51], Wall Street Journal [53], Fisher [12], Mozilla's Common Voice [47], or CHiME-6 [68], each having differences either in the audio context or size and hence each provide a different challenge for the model. A common metric for the accuracy of the speech-to-text recognition model is the WER; this metric relates to how well the predicted string of characters matches the actual transcript. As the state-of-the-art WER has dropped significantly in recent years, researchers have identified and discussed shortcomings in the benchmark models that portray an unrealistic accuracy of ASR models [46, 61]. The benchmark datasets do not represent well a spontaneous human conversation and lack diversity in speakers. Such biases in datasets have for example shown to create a racial and gender discrepancy in ASR systems [33]. However, training a model falls out of the scope of this work and hence a thorough analysis of which dataset to train and/or evaluate on is not considered. For implementation purposes, a pre-trained model on a single or a combination of the standard benchmark datasets will be considered. The benchmark dataset of interest is LibriSpeech, as the targeted model introduced in Section 2.5.4 is trained on, amongst others, this dataset and benchmark results are reported for this dataset.

Regarding the LibriSpeech dataset, there are two versions; *clean*, which represents clean speech and controlled environments, and *others*, which represents more noisy and challenging speech.

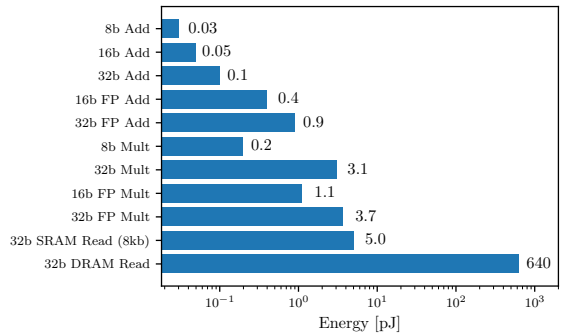
2.2. Quantization

State-of-the-art DNN models for ASR tasks utilise more than a billion floating point parameters [80]. A large number of learnable parameters implicitly lead to high compute and memory bandwidth challenges. This makes this kind of models impractical for devices with limited compute, memory, and power budgets, which limits their use cases.

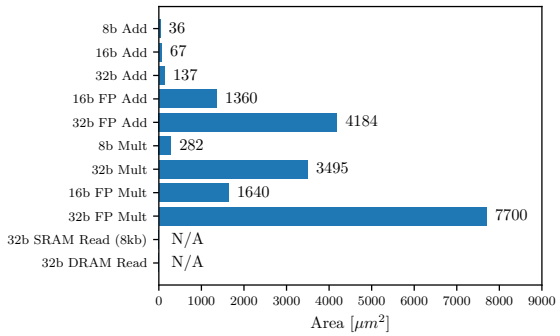
Researchers have been addressing the question on how to make DNNs more efficient in terms of power, latency, and memory footprint [17]. One category falls under designing efficient NN model architectures. One example of this, as will be introduced in Section 2.5.2, are depthwise separable convolutions. Other methods are pruning and knowledge distillation. The methods of interest for this work are quantization and implementing an NN architecture on an alternative hardware platform, namely an FPGA. Quantizing an NN model refers to mapping the set of continuous floating point parameters, such as the weights and activations, to a discrete set of integer values within a specific range dictated by the number of bits used to represent the quantized value. The newly obtained NN model can be fine-tuned by re-training the model, which is referred to as Quantization-Aware Training, or kept as-is, which is referred to as Post-Training Quantization.

The advantage of a quantized architecture is that the computations are expressed in lower precision arithmetic, which is more power and area-efficient and faster than floating point arithmetic. Figure 2.3a and Figure 2.3b visualise the energy and area costs of various operations in 45nm technology respectively. Note that an 8-bit integer addition is 30 times more energy-efficient and roughly 166 times smaller

than a 32-bit floating point addition. The same holds for an 8-bit integer multiplication, which is over 18 times more energy-efficient and over 27 times smaller than the 32-bit floating point counterpart. Secondly, quantized parameters imply a smaller memory footprint, meaning that fewer bits would be needed to be stored and transferred. Hence, a quantized network also lowers the power consumption of memory transactions and the weights can be stored in smaller, often faster, types of memory 'closer' to the compute logic. Clearly, this only has a theoretical advantage if it cannot be implemented in practice. This is where FPGAs can be used to realise a fast and energy-efficient DNN inference engine. Different to CPUs and GPUs, with FPGAs users can implement any type of bit-precision arithmetic and due to the small memory footprint, the weights can be stored on-chip instead of having power-hungry and slow off-chip memory transactions.



(a) Energy consumption in pJ for various operations in 45nm technology taken from [17, 26].



(b) Area utilisation for various operations in 45nm technology taken from [17, 26].

Figure 2.3: Energy and area comparison of different operations for different precision of operands in 45nm technology.

The question is, what happens to the accuracy of the quantized model? Interesting to note is that neural networks are often over-parametrised, i.e. containing millions of floating point parameters, and it has been shown that applying a quantization method often has a minor impact on the accuracy of the model [17, 45]. In fact, for particular models and quantization techniques, the quantized model could achieve higher accuracy compared to the baseline floating point model [45]. For a specific set of floating point CNNs and quantized CNNs, the trade-off in terms of accuracy and hardware cost is visualised in Figure 2.4 for an image classification task. The x-axis represents the hardware cost expressed in Look-up Tables (LUTs), which is one of the components in an FPGA that can implement arithmetic logic and will be explained in further detail in Section 2.3. As mentioned above, quantized networks can significantly lower the hardware cost at the expense of a slight accuracy drop.

Note that quantization is not limited to CNN models or image classification tasks; for example, a transformer-based model for speech recognition, named BERT, has been quantized from floating point to 8-bit format, compressing the model by $4\times$ with a minimal reduction in accuracy [77].

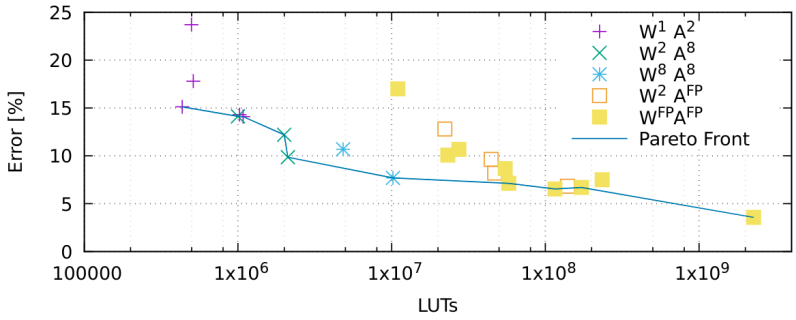


Figure 2.4: Accuracy versus hardware cost, expressed in look-up tables (LUTs), for floating point CNNs and several quantized CNNs, taken from [7]. $\mathrm{W}^{w}, \mathrm{A}^{a}$ refers to the precision in bits (w, a) of the weights (W) and activations (A). Note that FP refers to floating point.

2.3. Field Programmable Gate Arrays

As discussed in the previous section, quantized NNs can offer multiple advantages in terms of performance, area, and energy efficiency. FPGAs are an excellent device to implement such lower-bit precision arithmetic as a programmer has full control over the instantiated compute logic.

FPGAs are a specific type of integrated circuit that can be reconfigured after fabrication [73]. Figure 2.5 visualises in a broad sense the differences between CPUs, GPUs, FPGAs, and Application-Specific Integrated Circuits (ASICs) in terms of their flexibility and performance, area, and power efficiency. Compared to CPUs and GPUs, FPGAs can potentially provide improvements in the throughput and/or latency of an algorithm by exploiting its parallel dataflow-style architecture and it is typically more power and energy-efficient. Despite the fact that an ASIC seems superior in terms of performance and power efficiency, note that developing and producing an application on an ASIC is time-consuming and expensive. Besides, FPGAs are reconfigurable, which means that any type of application can be changed and optimised in the course of time. Especially in the fast-developing field of neural network inference, this flexibility is important. In this section, a brief and broad overview of FPGAs is given.

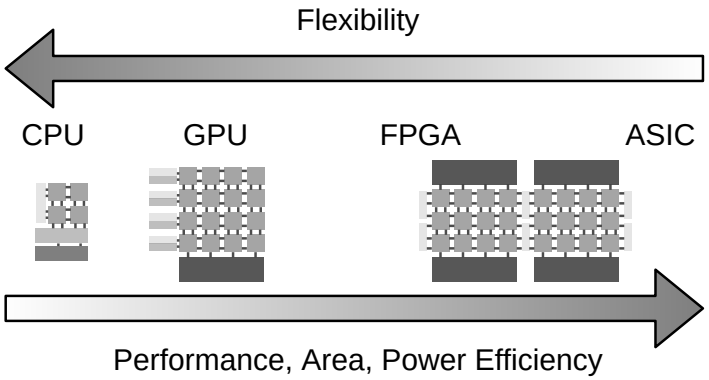


Figure 2.5: A spectrum showing the differences in terms of flexibility and performance, area, and power efficiency between CPUs, GPUs, FPGAs and ASICs.

FPGAs are composed of programmable logic blocks (slices) and memory elements that are connected to each other by means of a programmable interconnect, denoted by the routing channel and switch-box, as shown by the highlighted block in Figure 2.6. Modern FPGAs also contain on-chip memories, coarse grain hard-wired functions and are even integrated with an on-chip CPU. The typical resources found in an FPGA are look-up tables (LUTs), flip-flops (FFs), Digital Signal Processing (DSP) blocks, memory components, wires and input/output (IO) ports. A simple overview of a modern FPGA device is shown on the right side of Figure 2.6. Computations are expressed as truth tables, which are implemented using LUTs and FFs. An n-bit LUT consists of a 2^n:1 multiplexer and an 2^n-bit memory, where n is typically six. An FF can be seen as a storage component that can hold a certain output signal for a specific duration and is regulated by the clock. An FF is useful for example for temporary data storage and breaking up long signal paths or computations into multiple cycles. Note that the longest duration of a signal path between two FFs dictates the clock frequency.

Typically, FPGAs also contain other primitives, such as DSPs, which is a specialised hardware block that allows efficient computation of $a \cdot (b + d) + c$, where a, b, c, d are scalars. Besides DSPs, there are also other specialised computational components, such as an optimised adder. As stated before, all types of computations can also be implemented with LUTs. However, for specific computations, a DSP can be preferred over a LUT-based implementation due to lower power consumption and/or faster execution.

Within an FPGA, there are also embedded memory elements that can be instantiated as random-access memory (RAM), read-only memory (ROM), or shift registers. Depending on the device, the on-chip memory elements can be implemented in LUTs (LUTRAM), Block RAMs (BRAMs) and Ultra RAM blocks (URAMs).

Vivado High-Level Synthesis (HLS) compiler abstracts away many details from the target platform and enables a user to describe the intended algorithm in a High-Level Language (HLL) instead of Hardware Description Language (HDL). A programmer can still have control over lower-level details by

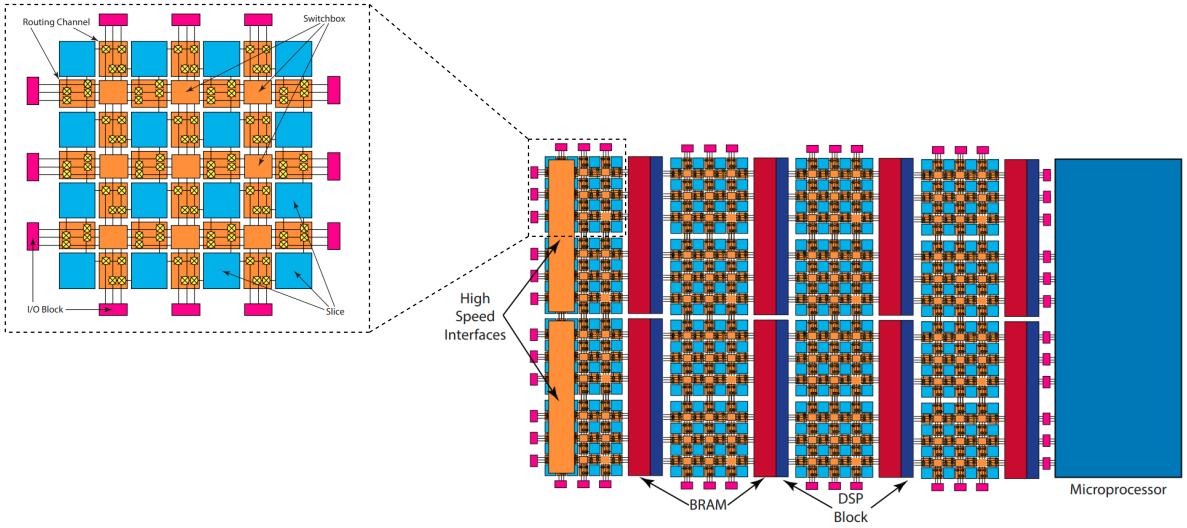


Figure 2.6: High level overview of the different components that can be typically found in a modern FPGA, taken and adapted from [31].

means of pragmas, such as which types of memory resources should be used to implement a specific array, how to partition a memory element, how much a specific loop should be unrolled in hardware, whether specific variables have a specific dependency. This lowers the design effort significantly, but often results in sub-optimal performance of the inferred hardware implementation. In Chapter 4, a more detailed analysis of the inferred hardware components is given.

In order to aid the discussions and interpretations in Chapter 4 and Chapter 5, the following sections will discuss the target device, the Alveo U250 card, in more detail.

2.3.1. Alveo U250

The Alveo U250 card is built on the Xilinx 16nm UltraScale architecture and uses the XCU250 FPGA [74]. The U250 card is one of the largest, in terms of compute capabilities, of its kind, and is designed to suit the needs of data centre tasks such as machine learning inference. Due to its large physical size, the U250 FPGA is split into multiple interconnected dies. Hardware components are divided among these so-called Super Logic Regions (SLRs) [71]. Hardware components that are implemented on different SLRs are connected to each other via Super Long Lines (SLLs). Note that these SLLs are slower than regular lines connecting the components within an SLR [66]. Secondly, these SLLs are scarce and they only connect adjacent SLRs. For large designs where resources must be allocated among multiple SLRs, SLR crossings are inevitable. Hence, in order to aid timing closure and obtain a design that runs potentially on a higher clock frequency, SLR crossings must be minimised. Another benefit of minimising SLR crossings is that it can reduce the power consumption [71]. The Xilinx tools offer algorithms that attempt to provide the fastest design by, among others, limiting the number of (critical) paths that cross SLRs and balancing the allocated resources among SLRs [71]. However, it has been shown that for large designs, the obtained solution attains a low performance due to the longest path traversing multiple SLRs [4]. The team from Xilinx Research Lab proposed a floorplanning algorithm that can provide designs with significantly higher performance; it has been shown that the clock frequency could in some cases be increased by 80 MHz.

2.3.2. Alveo U250 resources

To aid the understanding of the discussion presented in Chapter 4, the differences between the various hardware memory resources of the Xilinx Ultrascale FPGAs will be discussed. For those devices, there are three types of memory; LUTRAM, BRAM, and URAM.

LUTRAM
LUTRAMs are the smallest type of memory and are often preferred to implement smaller arrays in hardware. As mentioned before, LUTs can either be used to implement logic or to store data. More precisely, all LUTs can be initialised by the bitstream and used as read-only memory (ROM), while only a specific subset contains a data port and can therefore be used as both ROM and random-access memory (RAM) (i.e. written to during execution of the algorithm). The UltraScale FPGAs contain 6-input LUTs, meaning that they can be configured as 64-bit ROM and/or RAM [72]. A single 6-input LUT can either be initialised as a 64 addresses deep, 1-bit wide or 32 addresses deep, 2-bit wide single-port RAM. Within these slices, multiple LUTs can be combined to create a larger 512-bit RAM module. LUTRAMs are the most fine-grained type of memory; within a single slice, the RAM block can be inferred as, among others, single, dual, quad, or octal port memory with a range of address-width ratios such as $(32, 64, 128, 256, 512)$ addresses and $(1$ to $16)$-bits [72]. Note however that only a subset of combinations of port configuration and address-width ranges is possible, but the amount of flexibility for Xilinx tools to map an array to the most suited LUTRAM implementation is nevertheless large. Besides, a single LUT can also be used as a 32-bit shift register and combining the LUTs in a slice, up to a 256-bit shift register can be constructed.

BRAM
To implement larger memories, BRAMs or URAMs are more suited due to their more coarse-grained structure. BRAMs in the UltraScale architecture can store 36 kbits; either as a single 36 kbits RAM module or two independent 18 kbits RAM modules [75]. Table 2.1 visualises for each BRAM primitive the number of read and write ports and the address-width ratios that can be set for each port independently. Note that the limited amount of read ports influences how arrays are partitioned among BRAMs, which is further analysed in Chapter 4. Compared to LUTRAMs, BRAMs are more coarse-grained.

Primitive	N.o. read ports	N.o. write ports	Addresses	Width
RAMB18E2	2	2	16384	1
RAMB18E2	2	2	8192	2
RAMB18E2	2	2	4096	4
RAMB18E2	2	2	2048	9
RAMB18E2	2	2	1024	18
RAMB18E2	1	1	512	36
RAMB36E2	2	2	32768	1
RAMB36E2	2	2	16384	2
RAMB36E2	2	2	8192	4
RAMB36E2	2	2	4096	9
RAMB36E2	2	2	2048	18
RAMB36E2	1	1	1024	36
RAMB36E2	1	1	512	72

Table 2.1: Summary of various depth-width ratios for RAMB18 and RAMB36 primitives, as well as the number of read/write ports [75].

URAM
URAMs are the largest and least flexible type of memory resource available on the Alveo U250 device. A single URAM can store 288 kbits [75]. The URAM block has a fixed shape of 4096 addresses deep and 72-bits wide entries. Similar to BRAM, URAMs have two ports that can either perform a read or write operation per clock cycle. Furthermore, URAMs cannot be initialised by means of the bitstream; during power-up or device reset, URAMs are initialised to all 0's. This will pose a limitation for using URAMs due to the limited number of AXI-Lite interfaces that can be instantiated per Xilinx Object file and will be explained in more depth in Section 4.1.

2.4. Inference accelerators on FPGAs for ASR tasks

As CNNs become larger and larger, dedicated hardware has gained more attention for CNN acceleration due to the potential performance gain that can be attained. GPUs are a well known computational platform that is relatively easily accessible and usable to accelerate NN inference. However, FPGAs are more power-efficient. Secondly, for quantized neural networks with low precision integer weights and activations, FPGAs can potentially outperform GPUs in terms of performance. Lastly, the implementation barrier for FPGA-based CNN inference accelerators is becoming increasingly lower, where the FINN framework is an excellent example. Hence, CNN inference on FPGAs has become progressively popular [19].

The different speech-to-text implementations on FPGAs can be roughly divided into three groups: CNNs, RNNs, and HMMs. CNN-based networks are the best fit for an FPGA due to the fully parallelisable matrix multiplication. Due to the recurrent nature of RNN-based networks, hardware implementation is typically more complicated. On top of that, quantized NNs are also frequently considered. As FPGAs have limited on-chip memories, the memory footprint is typically lowered by compression or quantization techniques.

A brief summary of several FPGA accelerators for speech-to-text inference applications is shown in Table 2.2. Note that model size refers to the size of the learnable parameters and OnC and OffC refer to whether the weights are stored on-chip or off-chip respectively. CNN-based image classification networks have numerous FPGA accelerators in literature [19, 70]. However, interesting to note is that for ASR tasks, little proof of concepts have been demonstrated. Similar to this work, Wen et al. proposed an energy-efficient quantized CNN accelerator on an FPGA for speech recognition inference [69]. The main difference compared to this work is the complexity of the CNN, the ASR task, as well as the framework used to implement the accelerator. The CNN used in [69] consists of two convolutional layers and three fully connected layers and is trained to classify six words only. The size of the CNN in this work is much larger; it consists of 84 convolutional layers, as well as a large number of other types of layers such as activation functions and batch normalisation, trained to perform ASR tasks on 29 characters. Dinelli et al. also present a quantized CNN-based speech-to-text implementation on an FPGA with weights stored onto on-chip memory; however, this model is also small in terms of the number of layers and is able to recognise 10 different words only.

A lot of other research is focused on RNNs. Lee et al. presented an implementation of an LSTM based speech-to-text recognition network on an FPGA [42]. The weights of the LSTM are quantized to 6-bits and are stored onto on-chip memory. Besides, the authors have also implemented an RNN-based language model (LM), which improved the accuracy of the model. Han et al. proposed a framework named ESE to allow efficient implementation of a sparse LSTM model. To reduce the memory utilisation, they have applied pruning and quantization techniques, but the weights of the network still cannot fit fully onto on-chip memory. Another framework for implementing LSTMs is proposed by Wang et al., named C-LSTM. The difference to ESE is that the compression technique in C-LSTM allows to store all weights onto on-chip memory and allows for a more efficient hardware implementation. Another framework for implementing LSTM and Gated Recurrent Unit (GRU) RNNs on FPGAs is proposed by Li et al., named E-RNN [44]. Compared to the ESE and C-LSTM frameworks, the E-RNN models achieved higher performance and higher energy efficiency.

As RNNs are harder to implement in hardware due to their recurrent nature, the focus of this work is on CNNs. As the models shown in Table 2.2 are evaluated on different data and also vary in size, one-to-one comparison with the accelerator proposed in this work is unfortunately not trivial.

Model	**Model size** [MB]	**Frequency** [MHz]	**Power** [W]	**Latency** [µs]	**Performance** [GOps]	**Board**
CNN [69]	0.09 (OnC)	150	9.758	218.4	71.55	XCKU-115
CNN [13]	- (OnC)	116.2	2.259	390	-	Zynq-UltraScale+
LSTM [42]	1.1 (OnC)	100	9.2	-	-	XC7Z045
LSTM [20]	- (OffC)	200	41	82.7	0.0233	XCKU060
LSTM [67]	- (OnC)	200	22	16.7	-	XCKU060
LSTM [44]	- (OnC)	200	25	8.3	-	ADM-PCIE-7V3

Table 2.2: Comparison of several FPGA accelerators for speech-to-text inference.

2.5. QuartzNet

QuartzNet is an end-to-end CNN for ASR [34]. End-to-end models allow a single model to convert raw audio, or a pre-processed format, to a transcript [9]. The advantage of end-to-end models is that it simplifies training and inference. Compared to most other speech-to-text recognition models, the QuartzNet model has two fundamental differences. First, the QuartzNet model is based on a CNN. Compared to RNNs, CNN-based models are simpler to implement in hardware as it does not have a recurrent structure and consists of solely matrix multiplications. Secondly, QuartzNet uses a relatively small amount of parameters. Hence, fewer memory resources are required to store the parameters, making it suited to be implemented on edge devices. In the case of FPGA devices, the parameters can potentially be stored on-chip, which reduces memory access latency as well as power consumption. Besides, fewer parameters, in this particular case, translates to fewer compute operations and potentially a higher inference throughput.

The entire pipeline, starting from the raw audio signal and ending with a transcript, is shown in Figure 2.7. On the left, a high-level overview of the pipeline is given and moving to the right, the QuartzNet model is presented in more detail. The audio waveform gets converted into an intermediate format by means of the pre-processing stage. This intermediate format represents features of the audio waveform and is fed to the QuartzNet model. The QuartzNet model, in turn, predicts the most probable sequence of characters. Finally, the decoding stage converts the predictions of the model to a readable string of characters.

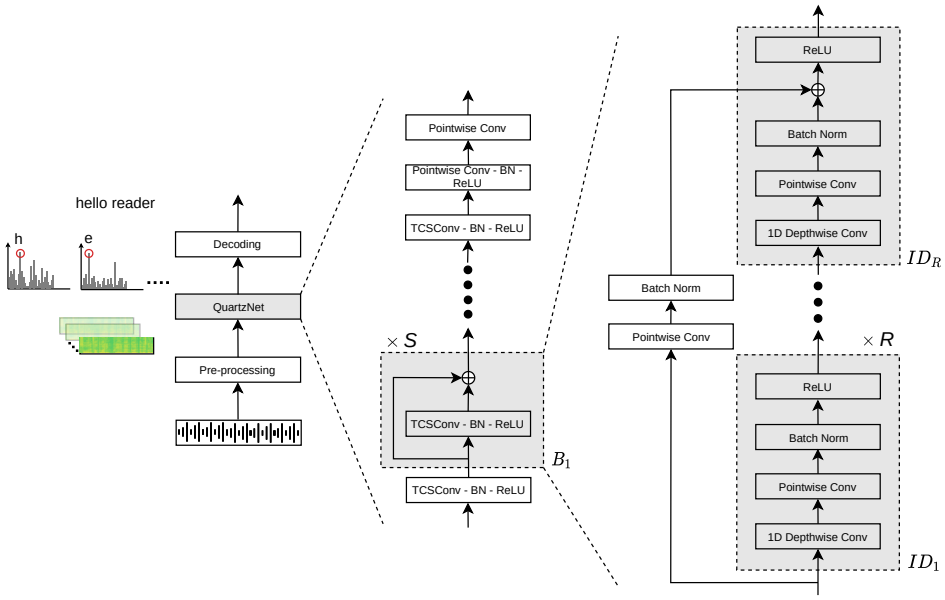


Figure 2.7: Schematic of the ASR pipeline in this work consisting of three stages: pre-processing or feature extraction, the acoustic model, which in this case is QuartzNet, and decoding.

The following subsections will discuss the various components of the QuartzNet model in more detail. Note that the focus of the discussion is on inference.

2.5.1. Pre-processing

The front-end of the pipeline is denoted as the pre-processing stage. Note that the QuartzNet model is not trained on raw audio data, but on an intermediate representation extracted from the raw audio waveform. The intermediate representation contains multiple features extracted from the audio data which are referred to as Mel-filterbanks. It is interesting to note that CNNs have shown to be able to learn directly from the raw waveform [50, 78], as CNNs can inherently learn and optimise/tailor the computation of Mel-filterbank features. For example, Zeghidour et al. propose a filter that can be treated as an additional layer and trained with backpropagation with the rest of the model, which allows an end-to-end model to learn from the raw waveform instead of pre-processed Mel-filterbanks [78]. However, re-training the QuartzNet model and examining the pros and cons of learning from raw waveforms

versus Mel-filterbanks falls out of the scope of this work. Hence, we will simply take Mel-filterbanks as input to the network.

The input audio is sampled at 16 kHz. Converting the raw audio to Mel-filterbanks consists of several steps [14]. The audio data is first dithered and subsequently, a pre-emphasis filter is applied to amplify high frequencies. This balances the frequency spectrum, avoids numerical problems for the Fourier transform that will follow, and could improve the signal-to-noise ratio. Next, the Hann window function and the Short-Time-Fourier-Transform are applied and the power spectrum of the signal is computed. After that, another filter function is applied to the power spectrum on a Mel-scale followed by taking the log of the obtained filterbank energies. The advantage of the Mel scale over the Hertz scale is that the Mel scale resembles the human perception of sound better; i.e. the Mel scale is more selective at lower frequencies compared to higher frequencies. Finally, the obtained spectrum is normalised by subtracting the mean of each filterbank and the output is padded to a convenient length. The resulting output consists of 1D tensors where each element represents the Mel-filterbanks. Each time-sample has 64 features and resembles 20 ms of raw-audio speech, where each sample has a 10 ms overlap with the previous sample. Due to the complexity of the steps involved to obtain the Mel-filterbanks, this step is not considered for hardware acceleration in this work. Besides, the execution of the QuartzNet model is expected to be the most computationally intensive part.

2.5.2. QuartzNet architecture

The entire architecture of QuartzNet consists of a sequence of 1D time-channel separable convolutions, batch normalisation, and ReLU layers as shown in the middle image in Figure 2.7. Besides, the QuartzNet architecture contains residual blocks. Note that the architecture of the QuartzNet model is based on the Jasper model [43]. The size of the QuartzNet model depends on the number of residual blocks B, the number of repetitions of each residual block S, and the repetitions of a particular sub-block within each block residual block R. Within each residual block, all 1D convolutions are identical in their shape (K) and the number of input/output channels C, with the exception of two convolutions in the third residual block (B_3) which operate on 256 input channels and produce 512 output channels.

More precisely, the first layer of the QuartzNet model consists of a time-channel separable convolution with a stride value of 2, followed by a batch normalisation and ReLU activation. Next, there are 5 residual blocks B_i, where the deeper ones have wider kernel widths for the convolution and an increased number of channels. The number of repetitions of each residual block B_i is either 1, 2, or 3. Each residual block B_i consists of a sub-block of time-channel separable convolutions, batch normalisation and ReLU activations. Each of these sub-blocks has the same general structure for the convolution, i.e. the same kernel width and the number of input/output channels, but with different weights. The sub-blocks are repeated R times. Note further that each residual block B_i contains a pointwise convolution and batch normalisation layer on the residual lane.

The authors have presented the following notation for the size of the architecture: $(5 \cdot S) \times R$. Thus, a 5x5 architecture refers to each residual block B_i being repeated once. By varying the number of repetitions of each sub-block, S, the number of parameters used for the model is altered. Note that this also has a consequence for the WER on the LibriSpeech dev-other dataset; the smaller models achieve a lower WER on the LibriSpeech dev-other and dev-clean dataset as reported in [34]. An overview is presented in Table 2.3.

Block	R	K	C	S 5x5	S 10x5	S 15x5	Others
C_1	1	33	256	1	1	1	Stride is 2
B_1	5	33	256	1	2	3	
B_2	5	39	256	1	2	3	
B_3	5	51	512	1	2	3	
B_4	5	63	512	1	2	3	
B_5	5	75	512	1	2	3	
C_2	1	87	512	1	1	1	Dilation is 2
C_3	1	1	1024	1	1	1	
C_4	1	1	29	1	1	1	
Params (M)				6.7	12.8	18.9	
WER dev-other (300 epochs)				15.69	12.33	11.58	
WER dev-other (1500 epochs)				-	-	10.78	

Table 2.3: Summary of the QuartzNet architecture, taken and adapted from [34].

Table 2.4 summarises the best-reported WER for the QuartzNet model from the original publication [34]. Note that an LM essentially adds additional information about whether a sequence of characters or words is a valid combination and as can be seen from Table 2.4, an LM can substantially improve the accuracy. The decoding and post-processing phase will be explained in further detail in Section 2.5.3. Note that the focus of this work is to create an FPGA accelerator for the QuartzNet model and hence, the focus is solely on the QuartzNet architecture. Creating a hardware accelerator for the LM or applying an LM to improve the WER is not considered.

Model	Language Model	Test clean	Test other
QuartzNet 15x5	None	3.90	11.28
QuartzNet 15x5	6-gram	2.96	8.07
QuartzNet 15x5	Transformer-XL	2.69	7.25

Table 2.4: WER accuracy on test clean and test other LibriSpeech dataset for the best proposed QuartzNet model with and without a language model. Data is taken from [34].

Depthwise separable convolutions

The main difference with respect to the Jasper model is that the QuartzNet model is based on 1D time-channel separable convolutions, also called depthwise separable convolutions [58]. Intuitively, a regular convolution can be thought of as performing two steps; each input channel is transformed/filtered and subsequently, all of the input channels are combined to produce a new representation. A 1D time-channel separable convolution splits the regular convolution operation, as shown in Figure 2.2, in two parts: a depthwise convolution and a pointwise convolution. The depthwise convolution applies a single 1D filter of size K to each of the input channels resulting in the same number of output channels. The pointwise convolution applies a single 1D filter of size 1 over all input channels, producing one or multiple output channels. Note that the depthwise convolution operates on each channel individually, but takes into account K time frames, while the pointwise convolution takes into account all channels, but operates on a single time frame. For this reason, these convolutions are also referred to as time-channel separable convolutions [34].

A schematic overview of a depthwise separable convolution is shown in Figure 2.8. Note that a regular convolution for the same kernel width and number of output channels, as shown in Figure 2.2, contains $K_H \cdot K_W \cdot IFM_C \cdot OFM_C$ weights. A depthwise separable convolution requires only $K_H \cdot K_W \cdot IFM_C + IFM_C \cdot OFM_C$ weights. Thus, the reduction in number of parameters and computations is given by Equation (2.3).

$$\frac{K_H \cdot K_W \cdot IFM_C + IFM_C \cdot OFM_C}{K_H \cdot K_W \cdot IFM_C \cdot OFM_C} = \frac{1}{OFM_C} + \frac{1}{K_H \cdot K_W} \tag{2.3}$$

The advantage of these convolutions is two-fold. Depthwise separable convolutions reduce the number of parameters in the model to be learnt, without having to sacrifice the number of convolutions applied or the width of the convolutional kernel. Fewer parameters to be learnt reduces the complexity of the model and implies less storage and computational requirements for edge devices. Note that this type of convolution has already been used before in image recognition networks, such as MobileNets [27], and speech recognition [22].

In practice, a batch normalisation and ReLU activation can be inserted after the depthwise convolution, instead of applying the pointwise convolution right after the depthwise convolution. Furthermore, note that QuartzNet contains 1D depthwise separable convolutions. The analysis and discussion above applies as well to 1D convolutions by setting IFM_H and K_H or IFM_W and K_W to 1.

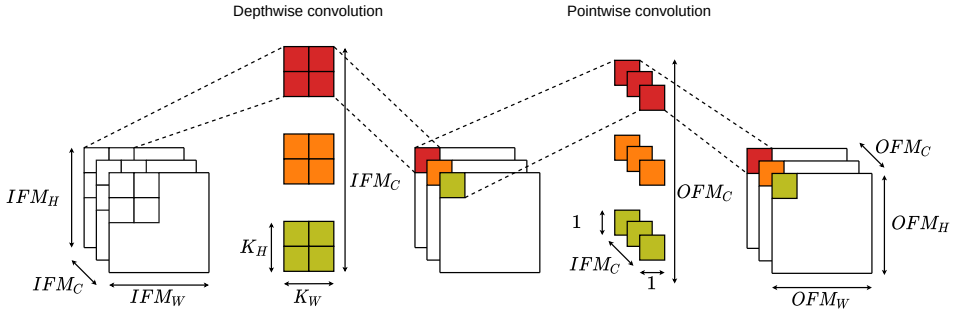


Figure 2.8: Overview of a depthwise separable convolution consisting of a depthwise convolution followed by pointwise convolution.

ReLU activation

The ReLU activation function is a simple mathematical function that, in the case of QuartzNet, adds non-linearity to the model. The function is applied on each element of the feature vector and is expressed in Equation (2.4). There are many variants to the ReLU activation function, as well as alternatives to the ReLU activation [49]. Discussing the pros and cons falls out of the scope of this work. However, it is interesting to note two favourable properties of the ReLU activation; it is computationally simple and secondly, a quantized ReLU is a uniform quantizer. The importance of a uniform quantizer will become clear when considering how such an activation is implemented in FINN, as explained in Section 3.2.

$$f(x) = \max(0, x) \tag{2.4}$$

Batch normalisation

Batch normalisation was introduced to accelerate the training of deep neural networks [30]. Given a feature map x, the output value y is obtained as given by Equation (2.5). Note that the parameters $\mu, \sigma, \delta, S, B$ are fixed during inference. The parameters μ, σ refer to the mean and variance of each channel of the feature map and are estimated during training. Note that δ ensures that a division by 0 never occurs. The batch normalisation layer restricts the output values to approximately have zero mean and unit variance. As this reduces the possible set of output representations that can be produced, the parameters S, B are added as additional scaling and bias terms that can potentially undo the batch normalisation. These parameters are learnt during training.

$$y = \frac{x - \mu}{\sqrt{\delta + \sigma}} \cdot S + B \tag{2.5}$$

Residual blocks
Lastly, the QuartzNet model also contains skip connections [23]. These skip connections form a so-called residual block as denoted by B_1 in Figure 2.7. In short, the intuitive idea is the following: a specific deep neural network should be able to have the same accuracy as its shallower counterpart because, in the worst-case, the deeper layers can simply be learnt to perform an identity mapping. However, in practice, this is not the case. By inserting a skip connection, deeper neural networks are easier to optimise and achieve higher accuracy compared to their counterparts that do not have these skip connections.

2.5.3. Decoding
The range of outputs of the QuartzNet model consists of 29 tokens; 26 tokens represent the characters of the English alphabet, one token represents an apostrophe, one token represents a space, and there is one so-called empty token. Each time-step in the output tensor contains a certain score for each of the 29 characters; by applying the softmax function, we can consider this output to be a probability distribution over the range of characters for each time-step. Note that the empty token is part of the output set as QuartzNet is trained with Connectionist Temporal Classification (CTC) loss [21].

More formally with regards to the CTC algorithm, the output of the QuartzNet model consists of a series of token predictions for T time-steps. At each time-step t, the output represents a probability distribution over the 29 possible tokens a given an input X, i.e. $p_t(a_t|X)$ [21]. We are interested in finding the most probable alignment of tokens a^* out of all possible alignments A. A simple algorithm to do this is to take the most probable token for each time step. In equations, the algorithm is expressed as Equation (2.6).

$$a^* = \arg\max_{A} \prod_{t=1}^{T} p_t(a_t|X) \tag{2.6}$$

There exist other more advanced heuristics or combinations with LMs for the decoding step, but employing and testing these are out of the scope of this work.

To obtain a readable sequence of characters, subsequent duplicate tokens are collapsed and the empty tokens are removed. In the example shown in Figure 2.7, assume that the output from the QuartzNet model is the sequence 'heellϵloo rreadderr', where ϵ refers to the empty token. This would in turn be collapsed to: 'hello reader'. Notice how the empty token is useful for distinguishing duplicate characters.

2.5.4. Quantized QuartzNet
The model used in this work is a pre-trained quantized 15x5 QuartzNet model developed by the Xilinx Research Lab [16]. The model is constructed and trained with Brevitas, which is a PyTorch library for Quantization-Aware Training that allows exporting the network to a FINN-supported ONNX graph [52]. Three different quantized models have been published. Two models have all weights and activations quantized to 8-bit values; one model is per-tensor scaled, the other per-channel scaled. The third model has the inner layers quantized to 4 bits, while the first and last layers are quantized to 8-bit values with a per-channel scaling technique. A summary of the published accuracy on the dev-other LibriSpeech dataset is provided in Table 2.5. Note that the reported WERs are obtained without an LM.

Kim et al. have also proposed several per-tensor quantized QuartzNet models with a particular technique named Q-ASR [32]. Their method does not require any training data as opposed to the quantized QuartzNet model from Xilinx Research Lab, which could be useful as gathering training data in a medical/healthcare context can be hard due to privacy concerns. Similar to the model published by Xilinx Research Labs, the quantization scheme reported is static; i.e. the algorithm does not require any statistics to be gathered or computed during inference. The authors have presented three different quantized models: 8-bit weights and activations, 6-bit weights and 8-bit activations, and 6-bit weights and activations.

As discussed previously, the accuracy drop for the quantized models is minor compared to the original floating point model. However, the memory footprint reduced by at least $4\times$.

Model	Scaling type	Bit width		WER	Memory footprint	MACs
		inner layers	outer layers			
NVIDIA [34]	-	32 FP	32 FP	10.78 %	75.38 MB	1659
Brevitas [16]	Per-tensor	8 INT	8 INT	11.03 %	18.58 MB	415
Brevitas [16]	Per-channel	8 INT	8 INT	10.98 %	18.58 MB	415
Brevitas [16]	Per-channel	4 INT	8 INT	12.00 %	9.44 MB	105
Q-ASR [32]	Per-tensor	8 INT	8 INT	10.28 %	18.45 MB	-
Q-ASR [32]	Per-tensor	6 INT	6 INT	12.31 %	13.84 MB	-
Q-ASR [32]	Per-tensor	W: 6 INT A: 8 INT	W: 6 INT A: 8 INT	10.83 %	13.84 MB	-

Table 2.5: Reported WER on LibriSpeech dev-other dataset for the QuartzNet model and various quantized QuartzNet models. Numbers are taken from [16, 32, 34, 52]. If not stated otherwise in the bit width columns, the bit width of weights and activations are equal to each other.

The final model used in this work is the QuartzNet model with per-channel scaling with 4-bit weights and activations for the inner layers and 8-bit weights and activations for the first and last layers. Note that the first and last layers have increased bit-width as these layers are more sensitive to quantization [7]. There are two specific reasons why this model was considered:

1. As the model is developed in Brevitas, the model can be exported in a specific open-source and widely used format (ONNX) that is suited to the FINN compiler. This work also serves to indicate how the software and hardware domains can be bridged by means of Brevitas and FINN.

2. The Q-ASR model was published on 31 March 2021, which is roughly at half of the timeline of this project. Thus, this model could not be taken into consideration.

2.6. FINN

There exists a wide variety of CNN to FPGA toolflows [65]. The generated accelerators can be divided into two categories: matrix of processing elements (MPE) and customised streaming dataflow (DF) architectures [4, 65]. The main difference is that the former executes the network layer-by-layer and contains a more generic hardware implementation that suits multiple layers, while the latter architecture executes each layer in parallel. DF-style accelerators contain optimised datapaths and compute units tailored to each individual layer and can theoretically lead to lower latency and more power-efficient designs compared to MPE-style accelerators, but have as disadvantage high resource and memory requirements for the FPGA device. FINN is a tool for enabling quantized DNN inference models to be mapped to a DF-style FPGA accelerator [7, 63]. There are four reasons that make FINN suited for creating an FPGA accelerator for the quantized QuartzNet model:

1. FINN is open-source; meaning that we have access to every part of the design flow to adjust it to support QuartzNet.

2. FINN is already used by a relatively large community and has already been used before to create a low-latency and high-throughput FPGA inference accelerator for other popular networks, such as MobileNet and ResNet-50 [38]. Note that similar to the QuartzNet model, the MobileNet model also contains depthwise-separable convolutions, albeit 2D instead of 1D.

3. FINN targets DF-style architectures and allows for a thorough design space exploration. As the QuartzNet model poses a challenge for the hardware implementation due to its very large size in terms of the number of layers and memory requirements, it is hard to fit this on an FPGA platform. Having the freedom to tailor each layer's compute and memory footprint allows for a trade-off between resource utilisation and throughput, which increases the likelihood of fitting the design on the target FPGA.

4. FINN supports arbitrary precision weights and activations and allows to target a wide variety of FPGA cards. This gives designers a lot of flexibility in quantizing the network; essentially allowing a more fine-grained trade-off between accuracy and compute required. Besides, the latter also offers a trade-off between target platform and target throughput.

A high-level overview of the various stages of the FINN compiler is shown in Figure 2.9. We typically start with a neural network description in PyTorch and trained with Brevitas. Brevitas is a PyTorch library for Quantization-Aware Training and allows for exporting models suited for the FINN compiler flow; models are exported in ONNX format with datatype annotations to weights to enable quantizing the weights to datatypes smaller than 8-bit integers [52]. ONNX is an open-source format to describe machine learning models [15]. The format defines what a neural network consists of in a protobuf description; e.g. a neural network model consists of a graph which in turn consists of nodes/operators. The ONNX format also describes a set of operators which are extended by FINN. An example will be given in Section 2.7.

A network of high-level ONNX layers is the starting point of the FINN compiler. The FINN compiler consists of a sequence of graph transformations that converts an ONNX graph with standard operators, found in [2], to a graph with FINN generated custom ONNX operators. Each of these custom operators has a corresponding C++ description in the FINN-hlslib library suited for the Xilinx Vivado HLS tool [40, 76]. The sequence of graph transformations can be divided into three main parts:

1. Streamlining floating point operations
2. Lowering convolutions
3. Conversion to HLS layers
4. Layer folding

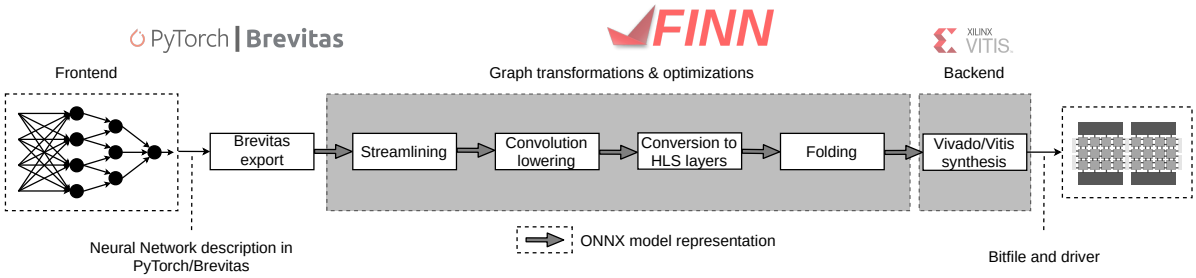


Figure 2.9: FINN compiler flow, inspired from [4].

2.6.1. Streamlining floating point operations

The first graph transformation is the streamlining transformation. Note that the FINN compiler currently only supports fully quantized NNs. However, in practice, quantized NNs can contain floating point computations in between quantized layers in order to improve the accuracy of the model [62]. For example, in the case of the quantized QuartzNet model, batch normalisation layers with floating point parameters for the mean and standard deviation can be found in front of several activation functions as well as channel-wise scaling operators after several convolutional layers. These floating point computations limit the ability to fully leverage the benefits of deploying such a model on an FPGA; floating point computations and floating point parameters are more expensive in terms of resources and power than their integer counterparts. The streamlining transformation eliminates all floating point operations from the model [62].

To understand how the floating point parameters get eliminated, or also referred to as absorbed, the underlying principle of the streamlining transformation must be explained. First, note that the streamlining algorithm only works for uniform quantizers. The algorithm contains three fundamental steps that enable the conversion to a network with only integer operations and parameters:

1. representing uniform quantizers as a thresholding function,
2. re-ordering and collapsing linear transformations, and
3. absorbing linear transformations into the weights of the thresholding function.

First, note that any uniform quantizer can be expressed as a thresholding operation followed by a linear transformation. A thresholding operation $f(x;T)$, also referred to as the (thresholding) activation function, maps real numbers x to an integer number $y \in [0, n]$, where y is the number of thresholds in T that x is greater than or equal to [62]. Note that T denotes the set of thresholds used for the activation function. Formally, this is expressed by Equation (2.7).

$$f(x;T) = \begin{cases} 0 & \text{for} \quad x \leq t_0 \\ 1 & \text{for} \quad t_0 < x \leq t_1 \\ \dots & \dots \\ n & \text{for} \quad t_{n-1} < x \end{cases} \tag{2.7}$$

Figure 2.10 visualises on the left a thresholding operation for three thresholds. Note that by applying scalar multiplication and addition, any other uniform quantizer $q(x)$ can be obtained. There is an important difference between the outputs of both quantizers; $f(x;T)$ has integer outputs, while $q(x)$ has floating point outputs.

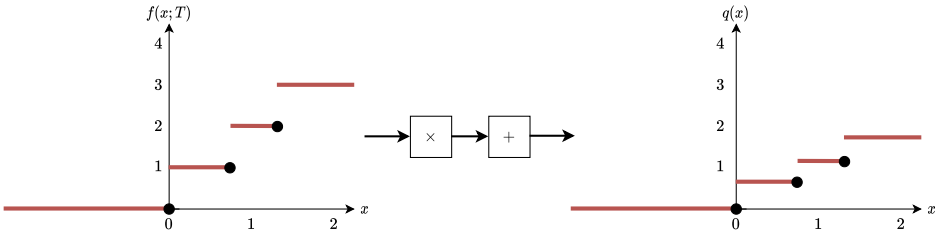


Figure 2.10: A 2-bit uniform HWGQ quantizer ($q(x)$) expressed as uniform quantizer ($f(x;T)$) followed by a multiplication by 0.538 and addition of 0 in this particular case. Example inspired from [62].

The question is, how can we get rid of the addition and multiplication in Figure 2.10, as we would then have our result after the thresholding function in integer format instead of floating point format. First, note that these linear transformations have favourable properties; linear transformations can be moved past subsequent convolutions or matrix multiplications without affecting the result and multiple consecutive linear transformations can be collapsed into a single linear transformation. By doing so, linear transformations at the output of a thresholding activation can be moved to the input of the subsequent activation. These linear operations can then be absorbed into the thresholds of the activation function. This is achieved by noting that for each input $x' = a \cdot x + b$, where a, b are scalars for simplicity, we can rewrite the thresholding function expressed by Equation (2.8) as Equation (2.9) by updating the thresholds.

$$t_{i-1} < a \cdot x + b \leq t_i \tag{2.8}$$

$$\frac{t_{i-1} - b}{a} < x \leq \frac{t_i - b}{a} \tag{2.9}$$

Note that this kind of absorption of floating point values into the thresholds can also be done for other floating point scalar operations in the network and, with some exceptions, also for channel-wise additions and multiplications. This will be explained further in Section 3.2. Finally, given that the input x to the quantization function is in integer format, all of the thresholds can be rounded up to the nearest integer without affecting the final result.

To summarise, the goal of the streamlining transform is to represent particular layers as linear transformations, re-order those linear transformations to appear in front of the thresholding activation, absorb those linear transformations into the thresholds of the thresholding activation function, and finally round the thresholds to the nearest integer. In that way, batch normalisation layers, which can be modelled by linear transformations (a multiplication and an addition), and other (channel-wise) scaling transformations can be absorbed into the activation function, which converts the graph into an integer only representation mathematically equivalent to the original graph.

2.6.2. Lowering convolutions

The second graph transformation is to *lower* convolutions to matrix multiplications [10]. The operation of a standard convolutional layer can be thought of as a filter sliding over the input data, as explained in Section 2.1.1. A naive convolution operation can be implemented in software as six nested loops where you first iterate over the batches, the 2D input image, the 2D kernel, and over the input channels. However, implementing this in hardware is not efficient, as it will impose large storage requirements to buffer the input images and the throughput will suffer due to the data access pattern [10].

For a convolution operation, note that each output pixel is obtained by a dot product between a vector of input pixels and a vector of kernel weights. The convolution lowering approach is taking advantage of this and replaces the aforementioned convolution operator with two steps: a sliding window unit (SWU), also referred to as an im2col operation, and a matrix multiplication. For the FINN compiler, there are two domains in which lowered convolutions can be executed; software domain (Python, ONNX-runtime) and hardware domain. In the software domain, an ONNX Conv operator is lowered to a custom FINN-ONNX operator named Im2Col followed by a MatMul ONNX operator. In the hardware domain, specific optimised kernels are written for the SWU and matrix multiplication. These will be explained further in Section 2.7.

An example of a regular convolution expressed as matrix multiplication is shown in Figure 2.11. This example shows a convolution between an input image with three input channels (IFM_C) of dimensions $[IFM_H, IFM_W] = [3, 3]$ and a kernel of dimension $[K_H, K_W] = [2, 2]$. For simplicity, the strides and dilation value is assumed to be 1 and the input image is not padded. The SWU reorganises the input image into rows that contain all the input pixels needed to compute a single output pixel. This is visualised by the lower leftmost matrix. Note that each row of the matrix contains, for each of the output pixels, all of the input pixels that are in the receptive field of a particular output pixel. As the weights are also reshaped in the appropriate format, each vector product results in a single output pixel.

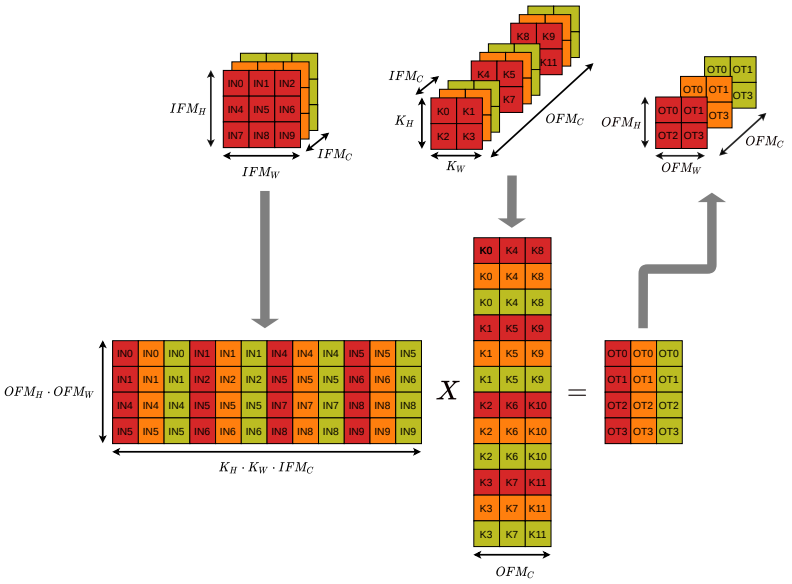


Figure 2.11: An example of a convolution lowering method for a regular convolution. Note that the image represents the SWU and matrix multiplication from the perspective of the HLS implementation of these kernels as well as how a lowered convolution is executed in the FINN ONNX domain.

An example of an equivalent depthwise convolution lowered is shown in Figure 2.12. Note that the depthwise convolution requires significantly fewer parameters and multiply-and-additions than the regular convolution.

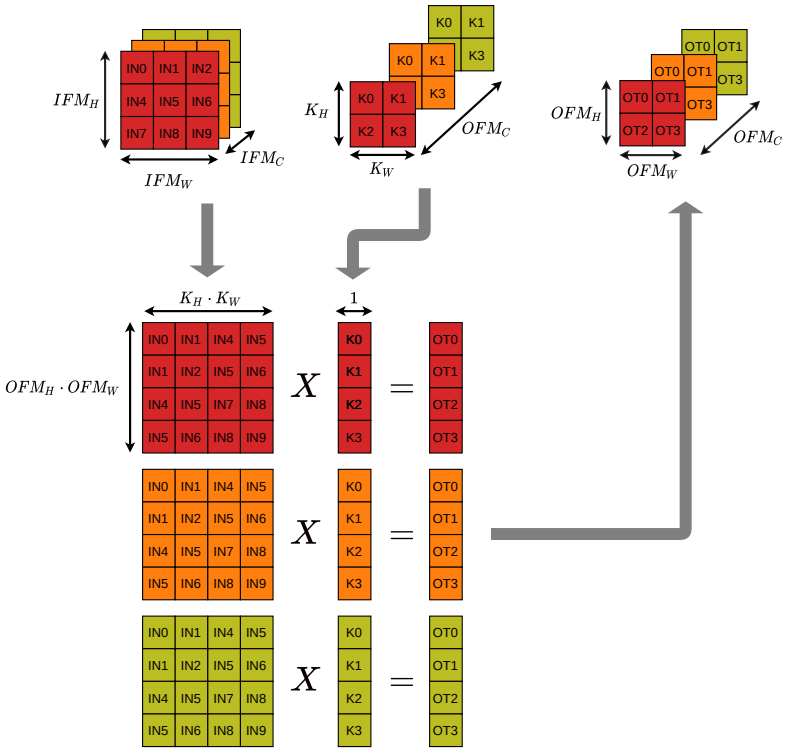


Figure 2.12: An example of a convolution lowering method for a depthwise convolution. Note that the image represents the SWU and matrix multiplication from the perspective of the HLS implementation of these kernels. In the ONNX domain of FINN, a depthwise convolution is executed in a similar way as shown in Figure 2.11, except that the weight matrix is sparse to ensure that a depthwise convolution is performed.

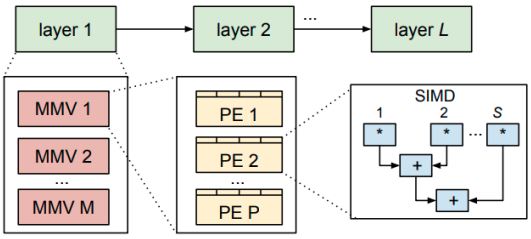
Lowering the convolutions as shown in Figure 2.11 and Figure 2.12 has the advantage of increasing the throughput as a matrix multiplication can be executed fast in parallel on GPUs. A similar approach to convolution lowering can for example also be found in the GPU acceleration library for deep neural networks [11]. In FINN, the matrix multiplication can be executed efficiently with a so-called Matrix Vector Activate Unit (MVAU) component. This will be discussed in more detail in Section 2.7.

2.6.3. Conversion to HLS layers

The next graph transformation bridges the gap to a hardware implementation. Note that after lowering, we have a graph consisting of a sequence of ONNX operators. In order to realise a hardware implementation, for each of the ONNX operators an equivalent C++ (HLS) implementation must be available that is suited for creating an HDL implementation with Vivado HLS. The FINN hls-library contains a large set of C++ kernels for several standard and custom FINN-ONNX operators, such as the Im2Col, MatMul, Add, and MultiThreshold operators [40]. Optimising the C++ implementation of the kernel is vital to achieving an efficient hardware implementation. The main difference between the operators from the QuartzNet model and the operators from other models previously used with FINN is that all operators from the QuartzNet model assume 1D input images instead of 2D. The implications and challenges of this will be discussed further in Section 2.8.

2.6.4. Layer folding

The parallelism factor of each layer, related to the amount of time multiplexing, can be adjusted to explore different configurations of the model. This parallelism factor, also referred to as the folding factor, specifies the amount of processing elements (PE) within each compute unit (layer) and the amount of Single Instruction, Multiple Data (SIMD) lanes per PE. Figure 2.13a and Figure 2.13b visualise the different levels of parallelism for the FINN layer that performs the matrix multiplication of a regular lowered convolution. At the highest level, note that each layer is executed in parallel as FINN generates a DF-style hardware architecture. Next, multiple output pixels can be generated in parallel. At the time of writing, this is still an experimental feature in FINN and hence, not considered in this work. Finally, the number of PEs and SIMD lanes in each PE can be scaled accordingly. Important to note is that this only holds for the so-called MVAU. Other types of layers have different levels of parallelism, which will be explored and discussed further in Chapter 4.



(a) Simple schematic that visualises the different levels of parallelism for the first layer. This specific example assumes that the first layer is a particular FINN node responsible for the matrix multiplication in regular (lowered) convolutions.

```
foreach layer: # L-parallel (all)
  foreach output_pixel: # M-parallel
    foreach output_channel: # P-parallel
      foreach input_channel: # S-parallel
        MAC(input, weights)
```

(b) Simple schematic in pseudo-code that visualises the different levels of parallelism in FINN.

Figure 2.13: Schematics that visualise the different levels of parallelism for a particular FINN layer. Images are taken from [4].

Setting the right parallelism for each layer is not a simple task. There are two parameters that are tuned by the parallelism factors; throughput and resource utilisation. The goal is often to maximise the throughput of the design while staying within the resource bounds of the target device.

In general, a designer would try to tune the parallelism in such a way to match the latency of each individual layer (compute unit); the goal is to create a latency balanced design. This is because the throughput of the network is essentially limited by the largest latency of a compute unit in the design. Decreasing the latency of other layers beyond the largest latency will not improve the throughput, but merely cost in terms of resource utilisation. However, in terms of end-to-end latency of the network, it is beneficial to have the latency of each layer as low as possible. However, as the QuartzNet model is large and resources are expected to become scarce, a latency balanced design is preferred.

Note that the folding factors of a layer have a linear relationship with the latency of that layer; increasing/decreasing the parallelism by a factor of 2 will decrease/increase the latency by a factor of 2 respectively. However, by tuning the parallelism of a layer, the logic and memory resource utilisation will be affected. As a simple and representative example, consider the node responsible for the matrix multiplication in a regular lowered convolution, referred to as the MVAU. Figure 2.14 visualises such a matrix multiplication, where the first operand represents the matrix with weights and the second operand represents the lowered image. Note that this figure is the same as Figure 2.11, but then with both matrices transposed and swapped in order. The blue colour indicates the dimension along which the $SIMD$ parallelism operates, the green colour indicates the PE parallelism, and the red colour indicates the MMV parallelism which is always 1 as explained before.

This layer performs a matrix multiplication between the input pixels and weight matrix of shape $[MH, MW]$, where MW and MH are related to the number of input and output channels respectively. For this layer, the parallelism factors PE and $SIMD$ are directly related to how many multipliers and adders are used to unroll the matrix multiplication. Hence, by increasing the parallelism, the expectation is that the logic resources, i.e. either DSPs or LUTs, will increase depending on which are used to implement the arithmetic logic. However, the parallelism factors PE and $SIMD$ will also dictate the shape for the array implemented in hardware that holds the weights. Note that the weight array contains a total of $MW \cdot MH \cdot W_B$-bits, where W_B stands for the number of bits used to represent a weight value. Furthermore, note that each PE will have $SIMD$ number of multiplication/additions in parallel between the input pixels and weights.

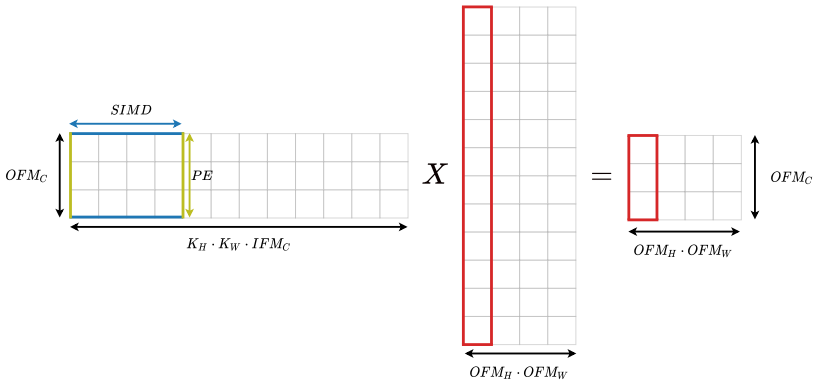


Figure 2.14: Convolution operation expressed as matrix multiplication between the weights (first operand) and input image (second operand).

Therefore, when the MVAU reads in a new data packet for processing, it should also read $PE \cdot SIMD \cdot W_B$-bits from the weight array. Experiments have shown that this results in a weight array of width and depth as expressed by Equation (2.10) and Equation (2.11) respectively.

$$\text{Width} = PE \cdot SIMD \cdot W_B \tag{2.10}$$

$$\text{Depth} = \frac{MW \cdot MH}{PE \cdot SIMD} \tag{2.11}$$

As explained in Section 2.3.2, the memory resources in hardware have a fixed set of shapes. Therefore, if for example the width of the weight array is not utilising the full width of the hardware memory component, the memory component will not be fully occupied and the remaining bits along the width are wasted. Especially for memory components that are less flexible, such as URAMs, this might lead to severe under-utilisation of the memory component. Thus, by changing the PE and $SIMD$ parallelism factors, the memory utilisation efficiency is affected. For large models such as QuartzNet, it is vital to ensure that the memory resources are utilised efficiently in order to fit the design on the board. Therefore, a thorough design space analysis regarding the effect of the parallelism factors on the logic and memory utilisation is presented Chapter 4.

2.7. FINN custom operators

FINN contains a set of custom ONNX operators that are used to represent the model in intermediate steps during the transformation flow. In general, there are three types of operators used:

1. ONNX operators: such as a Conv and MatMul.
2. Custom FINN-ONNX operators: such as an Im2Col and MultiThreshold.
3. FPGA dataflow nodes: such as a ConvolutionInputGenerator and StreamingFCLayer_Batch, also referred to as SWU and MVAU respectively.

The first two nodes are used to represent the graph until the layers are converted to HLS layers, as shown in Figure 2.9, and can be used used to execute the graph by means of ONNX-runtime. After the conversion to HLS layers, FPGA dataflow nodes are inserted and C++ and RTL simulation can be used to functionally verify the model.

Depthwise separable convolutions are the main type of layer encountered in QuartzNet and hence will be the focus point of this discussion. A depthwise separable convolution after streamlining has a similar structure as shown as the leftmost graph in Figure 2.15. After convolution lowering, notice that a depthwise convolution is replaced by an Im2Col operator followed by a MatMul operator, while a pointwise convolution is simply replaced by a MatMul operator. After conversion to HLS layers, the Im2Col operator is replaced by the SWU operator. The MatMul is either replaced by a Vector Vector Activate Unit (VVAU) or Matrix Vector Activate Unit (MVAU) depending on whether a depthwise or regular/pointwise convolution is applied respectively.

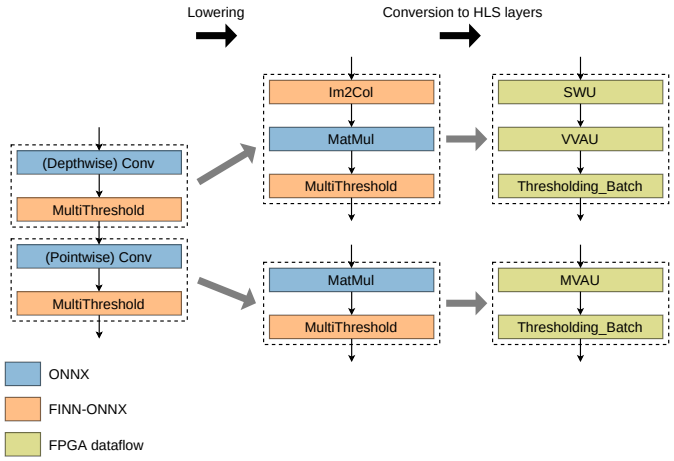
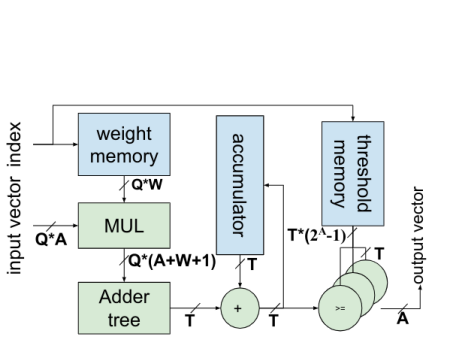


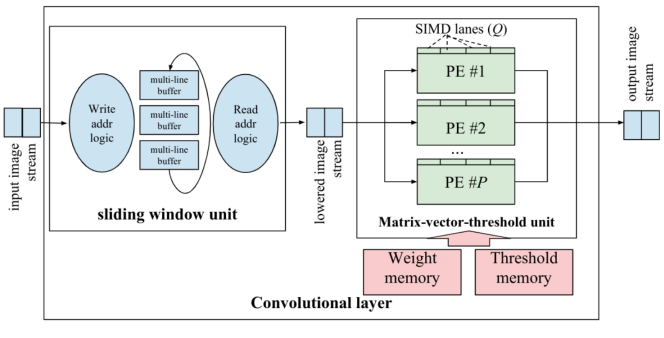
Figure 2.15: Schematic of how the ONNX operators are transformed to FPGA dataflow nodes, starting from a regular convolution and applying the lowering transformation and subsequently converting the layers to HLS layers. For simplicity, additional nodes such as ONNX Transpose operators are left out, but a more detailed example will be studied in Section 3.2. Blue nodes are standard ONNX operators, orange nodes are FINN-ONNX operators, and green nodes represent FPGA dataflow nodes.

The core of the MVAU and VVAU consists of a processing element as shown in Figure 2.16a. The variable Q refer to the number of input channels computed in parallel ($SIMD$). A,W,T refer to the bit width of the input/output activations, weights, and thresholds respectively. Each PE performs $SIMD$ multiplications in parallel, reduces the results in an adder tree and accumulates the result with the previous intermediate output. After the entire row of kernel weights and column of image values is multiplied and accumulated, referring to the row and column in Figure 2.14, the output value is compared against a set of threshold values to obtain the final output activation. Note that the weight memory can either be embedded within the component, or synthesised separately and streamed into the MVAU block. The former is referred to as *const* memory mode, while the latter is referred to as *decoupled* memory mode. Note further that the thresholding operation is embedded within the PE in Figure 2.16a. This thresholding operation can also be synthesised as a separate block.

Figure 2.16b shows a hardware level overview of a regular convolution. As explained before, a regular convolution is implemented by means of an SWU and MVAU. Internally, the SWU buffers the incoming stream of pixels and by means of a specific indexing logic, the buffer entries are written to the output stream to obtain a lowered image as shown in Figure 2.11. The MVAU executes the matrix multiplication, similar to what is shown in Figure 2.14, by broadcasting the weights and thresholds among multiple SIMD lanes and PEs and fetching $SIMD$ input pixels from the input stream in parallel. In Section 3.4.2, a more detailed analysis of the SWU is given. More precisely, the SWU implementation for 1D images is discussed and another SWU implementation for 1D images is presented that minimises the size of the intermediate buffer.



(a) A hardware level overview of a PE.



(b) A hardware level overview of a (regular) convolution which consists of a SWU followed by an MVAU.

Figure 2.16: Hardware level overview of how a convolution is represented in HLS layers and a schematic of a PE. Images are taken from [7].

2.8. Challenges for QuartzNet in FINN

Due to the size of the QuartzNet model, creating an FPGA implementation by hand is not feasible to do. Furthermore, exploring different architectural choices within a design space will be vital to obtain a successful implementation on an FPGA. The FINN compiler provides an excellent way to overcome these challenges, but the current version of the tool requires some extensions to support the layers encountered in the QuartzNet model. These are summarised below:

- The transformations and custom operations in FINN assume that the neural network description is 2D. However, the QuartzNet model is based on 1D convolutional layers. Thus, the FINN compiler needs to be extended to support 1D neural networks. This will be explained further in Section 3.1.
- The sequence of transformations to streamline the model needs to be tailored to successfully streamline the QuartzNet model. This will be explained further in Section 3.2.
- Lowering 1D convolutions requires extensions to the FINN-ONNX Im2Col operator, as well as a custom 1D C++ (HLS) implementation of the SWU. This will be explained further in Section 3.3 and Section 3.4 respectively.
- The design space must be explored carefully to ensure that the QuartzNet model can fit on a target device. Besides, balancing the resource utilisation and staying within recommended limits imposed by the synthesis tool is expected to ease timing closure and potentially achieve a higher frequency design. This will be explained further in Chapter 4.

3

Design challenges & implementation

As explained in Section 2.6, the entire end-to-end flow can be subdivided into the following five stages:

1. Exporting the ONNX model
2. Streamlining floating point parameters
3. Lowering convolutions
4. Converting (ONNX layers) to HLS layers
5. Layer folding

This chapter will discuss in more detail which modifications were required for enabling 1D convolutions to be mapped on an FPGA by means of the FINN compiler. First, design considerations for exporting the ONNX model are motivated in Section 3.1. Next, the sequence of streamlining transformations is elaborated on and the motivation behind new graph transformations is explained in Section 3.2. After that, the design challenges for lowering 1D convolutions is explained. Lastly, the required modifications for enabling 1D ONNX layers to be mapped to equivalent 1D HLS kernels is explained and the HLS implementation of a custom 1D SWU is discussed in Section 3.3. As the layer folding stage is critical to ensure that the model can fit on an FPGA, an extensive design space exploration is presented in Chapter 4.

Most of the extensions proposed in this chapter can also be found merged into the open-source GitHub repository of either FINN [36], FINN-base [37] or FINN hls-library [40].

3.1. Exporting ONNX model

Brevitas supports exporting the trained model into a FINN supported ONNX format with datatype annotations to weights to enable quantizing the weights to datatypes smaller than 8-bit integers. However, note that the input to the quantized QuartzNet model in Brevitas is still in floating point format. In order to create an FPGA implementation, note that all of the supported HLS kernels in the back-end of FINN assume integer format inputs. Secondly, transferring integer quantized input data instead of floating point input data is faster and more energy-efficient due to the compression in size. Thus, the first question when exporting the model into the ONNX format is: how to quantize the input data to the FPGA-based QuartzNet implementation to integer format? This will be discussed in Section 3.1.1.

Furthermore, note that the graph transformations in FINN assume to operate on 4D tensors, while the quantized QuartzNet model from Brevitas is exported with 3D tensors. Therefore, the second question when exporting the model is: how to enable support for 3D tensors in FINN? This will be discussed in Section 3.1.2.

3.1.1. Quantization

The first consideration is whether to quantize the input images to integer format. The task of quantizing the input to integer format is to convert a continuous number in the range $[\alpha, \mathrm{B}]$ to a discrete set of values in the range $[\alpha_q, \mathrm{B}_q]$. The disadvantage is that information is lost as floating point values are clamped within a much smaller discrete set of values. There are a few design choices to be made here; e.g. whether to implement a symmetric or asymmetric quantization scheme, what the quantization granularity is, whether to implement a uniform or non-uniform quantization scheme, and what the range (α, B) of the floating point values is [17, 48]. To simplify the implementation, we will consider a uniform quantization scheme. Given a floating point number $x \in [\alpha, \mathrm{B}]$ and a quantized number $x_q \in [\alpha_q, \mathrm{B}_q]$, converting the number between the two formats can be accomplished as shown in Equation (3.1) and Equation (3.2). The constants s, z, b refer to the scale, zero-point and bit-width of the targeted quantized format respectively. Note that $\lfloor . \rfloor$ refers to the rounding operator. The minimum and maximum number in the continuous domain can be derived from the training set. If we assume that the minimum and maximum number in the continuous domain map to the minimum and maximum number in the integer domain respectively, i.e. α maps to α_q and B maps to B_q, then the expression for s, z can be found by substituting $\alpha \rightarrow \alpha_q$ and $\mathrm{B} \rightarrow \mathrm{B}_q$ in Equation (3.1) and solving for s and z. Note that during inference, x can lie out of the range $[\alpha, \mathrm{B}]$ that was derived from the test set. In that case, x_q can exceed the predefined range $[\alpha_q, \mathrm{B}_q]$ and the input sample x_q will not represent the original floating point value x. To overcome this, x_q should be clamped within the range $[\alpha_q, \mathrm{B}_q]$. However, for simplicity, it is assumed that the domain derived from the training set is representative for the test set.

$$x = s \cdot (x_q - z) \tag{3.1}$$

$$x_q = \lfloor \frac{x}{s} + z \rceil \tag{3.2}$$

$$s = \frac{\mathrm{B} - \alpha}{\mathrm{B}_q - \alpha_q} = \frac{\mathrm{B} - \alpha}{2^b - 1} \tag{3.3}$$

$$z = \mathrm{B}_q - \frac{\mathrm{B}}{s} = \frac{\mathrm{B}\alpha_q - \mathrm{B}_q\alpha}{\mathrm{B} - \alpha} \tag{3.4}$$

A symmetric quantization scheme has the zero-point in Equation (3.1) set to 0. We will consider asymmetric quantization as the distribution of floating point input values to the quantized QuartzNet model is likely to be asymmetric as well. Regarding the granularity, we will for simplicity consider two types; per-tensor and per-channel quantization. Per-tensor quantization refers to quantizing the entire input image with the same scale and zero-point constants. These constants can be found for example by taking the min and max values of the tensors in the training dataset. Per-channel quantization refers to quantizing each channel individually by calculating for each channel the scale and zero-point constants. These constants can be found by taking the min and max values of each channel for all of the tensors in the training dataset. Per-channel quantization can achieve a higher model accuracy compared to per-tensor quantization, but it would require channel-wise divisions/multiplications and additions/subtractions which is slightly more complicated and expensive in terms of hardware. As will be shown later, both scalar and channel-wise operations can be streamlined away, incurring no costs in hardware resources on an FPGA. Lastly, note that finding the scale and zero-point requires us to know what the range of input values (α, B) is. As mentioned before, the range can either be determined statically; i.e. by taking the minimum and maximum values of the tensors in the training dataset and assume that these values are representative for the input images encountered during inference. This might cause a higher rounding error and thus more noise on the input image, possibly degrading the accuracy of the model. Another method would be to dynamically calculate the range during inference, but this requires floating point operations which are costly in terms of area and energy. Hence, static quantization is preferred.

There are three different designs to consider in order to quantize the input data to the model to integer format to obtain a fully quantized model. Note that the first activation layer effectively quantizes the output to a signed 8-bit integer number, as this is how the thresholding function operates as explained in Section 2.6.1. The three methods are:

1. Execute the first convolution and activation encountered in the QuartzNet model on a CPU as shown on the left in Figure 3.1.

2. Create a custom HLS kernel for the first convolution and activation to support a floating point input stream.

3. Apply a per-tensor or per-channel quantization of the input data and execute the first convolution and activation on an FPGA as shown on the right Figure 3.1.

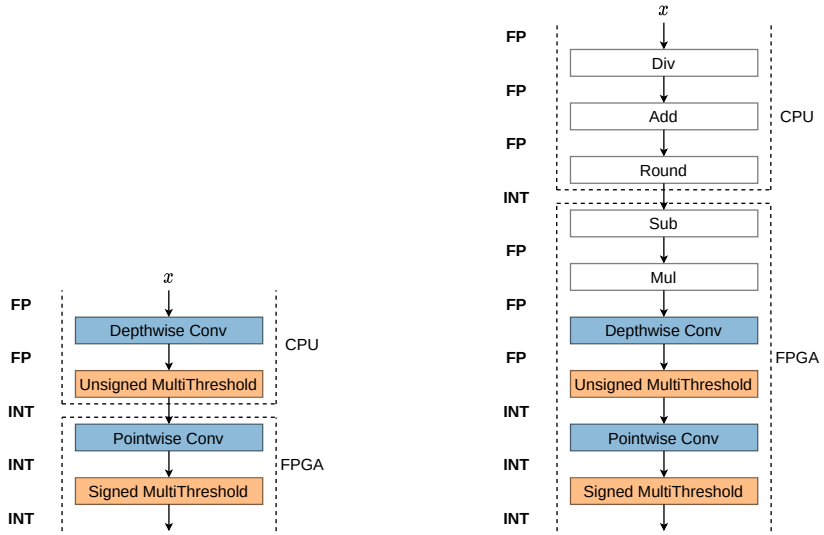


Figure 3.1: Schematic overview of two ways of ensuring that the input data to the FPGA is in integer format.

Regarding the first and second proposed methods, note that if the inputs to the QuartzNet model are in floating point format, the first depthwise convolution and MultiThreshold operator would have to be executed on a CPU. The reason for this is because the current HLS kernels for these operators in the FINN hls-library are written for integer format inputs. Note that the format of the intermediate tensors is also visualised next to each layer output in Figure 3.1. A custom HLS kernel could be written to overcome this problem, but for simplicity, exploring the existing infrastructure/components is prioritised. Besides, despite being a single convolution and activation function, implementing a floating point equivalent hardware implementation for these kernels will incur large resource overhead and power consumption compared to the integer counterparts as explained in Section 2.2.

Regarding the third proposed method, note that quantizing the input image to a reduced bit-precision and feeding it directly to the QuartzNet model, without any form of retraining, will most likely incur a relatively high penalty on the accuracy of the model. To minimise the accuracy loss, the trick shown in the right graph of Figure 3.1 can be applied. Note that the floating point tensor x is quantized to integer format and subsequently, converted back to floating point format. This introduces some additional computational overhead as the input data needs to be quantized on a CPU. Secondly, due to the non-linear rounding operation, the original floating point value is not reconstructed exactly; some noise is incurred on the input image. Note further that the input to the first depthwise convolution is still floating point format at this point. However, the point is that the floating point operations that convert the quantized tensor back to the original floating point format can be streamlined away. This would result in a graph where the input to the first depthwise convolution is in integer format. As will be explained in Section 3.2, both a scalar and channel-wise addition/multiplication can be moved past the depthwise convolution. Therefore, both a per-channel and per-tensor quantization technique can be considered to be applied.

To summarise, the first method requires the first two layers to run on the CPU, which will incur a (minor) penalty on the end-to-end execution time compared to executing those layers on an FPGA. The second method does not incur a penalty on the end-to-end execution time, but will incur additional hardware resources as it requires a floating point depthwise convolution and MultiThreshold hardware implementation. The third method requires only a scalar or channel-wise division and addition of the input tensor to be performed on the CPU, which is assumed to be negligible compared to a convolution operation as in the first method. Secondly, it does not incur additional hardware costs as the floating

point subtraction and multiplication at the input of the quantized QuartzNet model can be streamlined away. However, the downside is that the accuracy of the model will be lower as effectively noise is added to the input image due to the rounding operation.

To conclude, the first method is preferred as it does not incur an accuracy loss. For future work, it is worth investigating the third method to ensure that all layers of the QuartzNet model are executed on an FPGA to achieve a lower latency and higher throughput on an end-to-end inference task with an FPGA accelerated quantized QuartzNet model.

3.1.2. Handling 1D models

The second consideration to take into account is specific to the QuartzNet model. Note that the model operates on 1D images, while almost all of the core infrastructure in FINN, such as graph transformations for streamlining and graph transformations that infer HLS kernel implementations for the nodes in the graph, rely on the assumption of 4D input images. The simplest method to overcome this issue is to export the 3D tensors as 4D tensors by inserting a dummy dimension. In other words, a tensor of the format $[N, C, W]$ becomes $[N, C, H = 1, W]$, where N, C, H, W represent the batch size, number of channels, and height and width of the input image respectively. Secondly, we must ensure that the attributes of the nodes are compatible with 2D tensors. For example, a Conv ONNX operator should have the kernel dimension attribute extended from $[K_W]$ to $[1, K_W]$. The implementation of this transformation can be found on GitHub in the FINN-base repository [37]. The specific details of implementation are not deemed relevant for discussion.

3.2. Streamlining floating point parameters

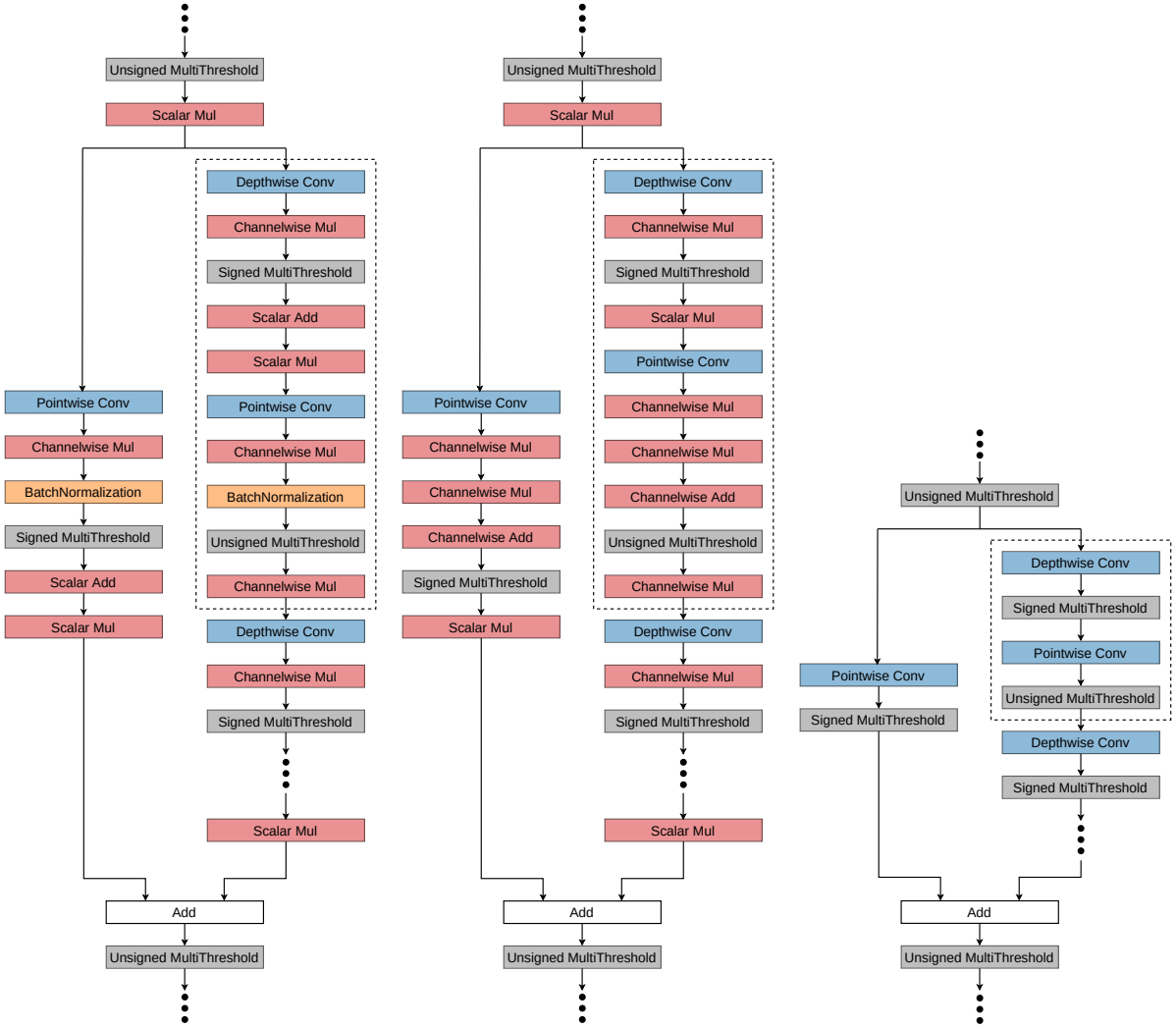
As explained in Section 2.6.1, quantized NNs can still contain floating point operations despite having quantized weights and activations. The goal of streamlining is to absorb floating point multiplications and additions into the activation function, which is called the MultiThreshold operator. This is achieved by applying graph transformations to rearrange the floating point multiplication and addition nodes in such a way that they are placed in front of a MultiThreshold node. A scalar or channel-wise floating point addition or multiplication can then be absorbed within the thresholds of the MultiThreshold operator; i.e. the same linear transformation can be applied effectively on the tensor by updating the threshold values of the activation function, as explained in Section 2.6.1. Note that this trick only works for scalar and channel-wise floating point multiplications and additions. Lastly, assuming that the input values of the MultiThreshold node are integers, the thresholds can be rounded up to the nearest integer while preserving the same outcome.

Regarding streamlining, the main challenge for the QuartzNet model is:

- As there is currently no support for running the streamlining transformation out-of-the-box, the right sequence of graph transformations must be found and/or new transformations must be written.

Figure 3.2a visualises a residual block of the ONNX graph for the QuartzNet model after exporting it from Brevitas. On the left, the residual lane is shown, and on the right, the first out of five of the repetitive sub-blocks is highlighted. As each of the repetitive sub-blocks and residual blocks is exactly the same, the analysis shown below extends to the rest of the residual blocks. However, as explained in Section 2.6, there are also a few layers before and after the residual blocks. These layers have minor differences and the analysis presented below is sufficient to apply to those layers as well. The grey boxes indicate FINN-ONNX operators, which is the node performing the activation named MultiThreshold. The red boxes indicate floating point additions and multiplications, also referenced as Add and Mul nodes. The orange boxes indicate batch normalisation layers, also written as BatchNormalization, and the blue boxes indicate quantized convolutions. The convolutions, also denoted as Conv, are either depthwise or pointwise. Note that in Figure 3.2a, the graph consists of a large number of floating point operators.

In order to get rid of the floating point operations, a sequence of transformations, that either expand, re-order or combine the nodes in the graph, must be applied. In general, the sequence of transformations should not matter as each of them preserve the functionality of the graph, but there is one particular graph transformation that must be executed first. Note that in Figure 3.2a certain activation functions are signed. Each of these MultiThreshold nodes is succeeded by a scalar addition which



(a) Visualisation of a residual block of the QuartzNet model with floating point operations, batch normalisation, convolutions and MultiThreshold operators.

(b) Visualisation of a residual block of the QuartzNet model where the batch normalisation operation is converted to a channel-wise multiplication and addition.

(c) Visualisation of a residual block of the QuartzNet model after streamlining, consisting of only convolutions and Multithreshold operators.

Figure 3.2: Overview of the nodes in QuartzNet before and after streamlining floating point operations.

follows from the way how Brevitas exports such an activation function. In order to have the signed ONNX MultiThreshold node compatible with the equivalent HLS layer named Thresholding_Batch, the scalar addition must be absorbed into the output bias term of the preceding (signed) MultiThreshold operator. Next, as explained in Section 2.5, the BatchNormalization operator can be expressed as a channel-wise multiplication followed by a channel-wise addition.

By absorbing the scalar additions into the preceding MultiThreshold and converting the ONNX BatchNormalization operator to a channel-wise multiplication and channel-wise addition, the graph will end up as shown in Figure 3.2b. Note that the residual block now solely consists of the MultiThreshold, Convolution, Add and Mul operators. Now, the question is: how to rearrange the multiplications and additions such that they can be absorbed into the thresholds of the MultiThreshold operator?

First, note that the scalar multiplication after the first unsigned MultiThreshold operator cannot be absorbed into the thresholds of the preceding MultiThreshold; the MultiThreshold operator is a non-linear operation. Hence, instead of applying a scalar multiplication on a tensor and broadcasting it to two other nodes, we can simply move the scalar multiplication to each lane and broadcast the output of the MultiThreshold operation to both of the lanes. The same applies for the two final scalar multiplication on both lanes; multiplying two addends by the same constant or vector and then adding the two outputs is the same as first accumulating the addends and then multiplying it by the constant or

vector (distributive property). Next, note that applying a scalar multiplication on a tensor followed by a depthwise or pointwise convolution is mathematically equivalent to applying first the convolution and then the scalar multiplication; these operations are commutative. The same holds for a channel-wise multiplication and depthwise convolution. For a scalar and channel-wise addition followed by a convolution, this property does not hold. However, note that applying a convolution operation on an input tensor $x + c$, where c is either a scalar or 1D vector, is the same as applying a convolution on both x and c separately and then accumulating the results. To summarise, scalar Mul nodes can be moved past depthwise and pointwise convolutions, channel-wise Mul nodes can be moved past depthwise convolutions, and scalar and channel-wise additions can be moved past depthwise and pointwise convolutions. Finally, note that multiple consecutive Add/Mul nodes can be absorbed into a single Add/Mul node by updating the additive/multiplicative constants respectively.

These graph transformations already exist in FINN. By applying them in the right sequence and absorbing the constants of the Add/Mul nodes into the MultiThreshold operator, we obtain the streamlined graph as visualised in Figure 3.2c. There is one important exception to note when absorbing the Add/Mul constants into the thresholds of the MultiThreshold operator. Negative multiplicands need to be split into a bipolar vector of signs and a vector of magnitude, where the vector of signs gets absorbed into the preceding Conv operator, and the vector of (positive) magnitudes into the thresholds of the succeeding MultiThreshold operator. The reason for this is that if a negative multiplicand would get absorbed in the thresholds, both the threshold and comparator must get updated to preserve the same mathematical expression, as shown in Equation (3.5). This flexibility would make the design of the MultiThreshold operator more complex. A simpler solution is to absorb the 1-bit bipolar vector of signs into the weights/multiplicands of the preceding convolution or matrix multiplication operation.

$$t_{i-1} < -x \leq t_i == -t_i \leq x < -t_{i-1} \tag{3.5}$$

3.3. Lowering 1D convolutions

Regarding lowering 1D convolutions, there are mainly two extensions to the FINN compiler that are made. The first one is related to the im2col algorithm, while the second one arose due to a practical limitation that occurs when the ONNX model becomes too large in size in terms of bytes.

3.3.1. Extensions im2col algorithm

Lowering convolution operators to an Im2Col ONNX operator followed by a MatMul operator allows for efficient hardware implementation as explained in Section 2.6.2. The previous models that have been successfully implemented on an FPGA by means of FINN were networks with 2D inputs. Thus, the FINN-ONNX Im2Col operator that implements the im2col algorithm only supported convolutions on 2D square images with equal padding along both dimensions and a 2D square convolutional kernel. Besides, convolutions with different stride values along the width and height dimension, as well as convolutions with a dilation value of greater than 1, were also not supported. Thus, to enable support for a 1D convolution, the Im2Col FINN-ONNX operator needs to be extended. For QuartzNet, the following design choices and challenges arose:

- The im2col algorithm needs to be extended to support 1D convolutions. To do so, a generic extension is made such that the im2col algorithm supports non-square convolutions. Note that a 1D convolution can be seen as an edge case of a non-square convolution. Besides, as 1D convolutions only apply padding along one dimension, support for non-equal padding must be enabled as well.
- The first 1D convolution in QuartzNet has a stride value of 2, as shown in Table 2.3 in Section 2.5. As a 1D convolution is treated as a non-square 2D convolution with one of the dimensions set to 1, the Im2Col operator should be able to support different stride values along both the width and height dimension of the image. Previously, the Im2Col operator only supported equal stride values along both of the image dimensions.
- The third-to-last 1D convolution in QuartzNet has a dilation value of 2, as shown in Table 2.3 in Section 2.5. Hence, the im2col algorithm needs to be extended in a generic way to support convolutions with a dilation value of greater than 1.

Besides adjusting the core of the Im2Col operator, adjustments to the attributes of the operator as well as to the transformation that lowers convolutions to a MatMul node, possibly with the Im2Col node in case of depthwise or regular convolutions, were made. These adjustments were mainly to break the assumption of operating on a square kernel; all of the changes can be found in the open-source GitHub repository of FINN-base [37]. The main contribution of the extensions is that the FINN-ONNX Im2Col node supports both square and non-square convolutions, with any kind of sensible combination between stride value and dilation value, with either equal or non-equal padding along the dimensions. Thus, within the ONNX domain of FINN, every type of convolution is supported as far as the tests performed can guarantee that. For future work, the only consideration is to extend the FINN hls-library with optimised HLS implementations of specific edge-cases in order for the FINN compiler to support any type of convolution. As explained before, a 1D convolution used in QuartzNet is an edge case of non-square convolutions; i.e. one of the dimensions of the image is equal to 1. The proposed extensions allow all of the convolutions encountered in QuartzNet to be lowered to a FINN-ONNX Im2Col node followed by an ONNX MatMul node.

3.3.2. Partitioning the model into sub-models

There is one important extension that arose from a practical limitation. Note that, as explained in Section 2.6.2, the transformation that lowers depthwise convolutions to a MatMul preceded by an Im2Col operator infers a sparse matrix multiplication. The weight matrix essentially contains a lot of zeros and this simplifies the depthwise convolution to an Im2Col and MatMul operation. However, as the QuartzNet model contains a lot of depthwise separable convolutions, the size of the ONNX model in bytes (to store) exceeds the maximum capacity due to the large memory footprint of the sparse weights associated to each MatMul. These large and sparse matrix multiplications also lead to a large run-time when executing the ONNX model with ONNX-runtime. There are two ways to solve this:

1. Implement a custom MatMul ONNX operator for depthwise separable convolutions that performs a dense matrix multiplication.
2. Implement a graph transformation that allows an ONNX model to be partitioned into multiple smaller ONNX sub-models.

The second method is preferred due to its simplicity. Note that this does not resolve the large run-time problem when executing the lowered model with ONNX-runtime. However, this is assumed to be acceptable as ONNX-runtime is not used for inference purposes, but merely for functional verification of the model during graph transformations. On top of that, the preferred method of verification is simulating the C++ kernels and performing RTL simulation as that matches closer to the functionality that will be implemented on the FPGA. Note that the HLS kernels for the equivalent Im2Col and MatMul node perform a dense multiplication. Converting the model into multiple sub-models was already added as transformation in FINN. However, going the other way around, i.e. expanding the sub-models and creating a single model out of it, had to be added. Discussing the details of the implementation is not deemed relevant; the changes can be found in the open-source GitHub repository of FINN-base [37]. Note further that the partitioning of the ONNX model into multiple connected sub-models also requires careful handling when a transformation is applied; the model needs to be partitioned into multiple sub-models, each sub-model needs to be expanded individually, the specific transformations need to be applied, and in the end, the sub-model needs to be packed again.

After partitioning the model into multiple sub-models and lowering the convolutions accordingly, there remains a final set of optimisations that have to be performed. Lowering a convolution introduces several transpose nodes as well. This is because the ONNX operators assume to operate on a tensor of $[N, C, H, W]$ format, where N, C, H, W represents the batch size, the number of channels, and the height and width of the image. The HLS kernels are written for tensors in $[N, H, W, C]$ format. Thus, after streamlining and lowering, the obtained model typically looks like Figure 3.3a. Note that only a sub-part of the residual block is shown, similar to what has been shown in Figure 3.2. Notice the difference with respect to the streamlined graph in Figure 3.2c; the Conv operator got replaced by a sequence of Im2Col, MatMul, and Transposes. The grey boxes indicate the activation layer, the red boxes indicate the Transpose ONNX operators, and the blue boxes indicate the layers that are responsible for performing a convolutional operation.

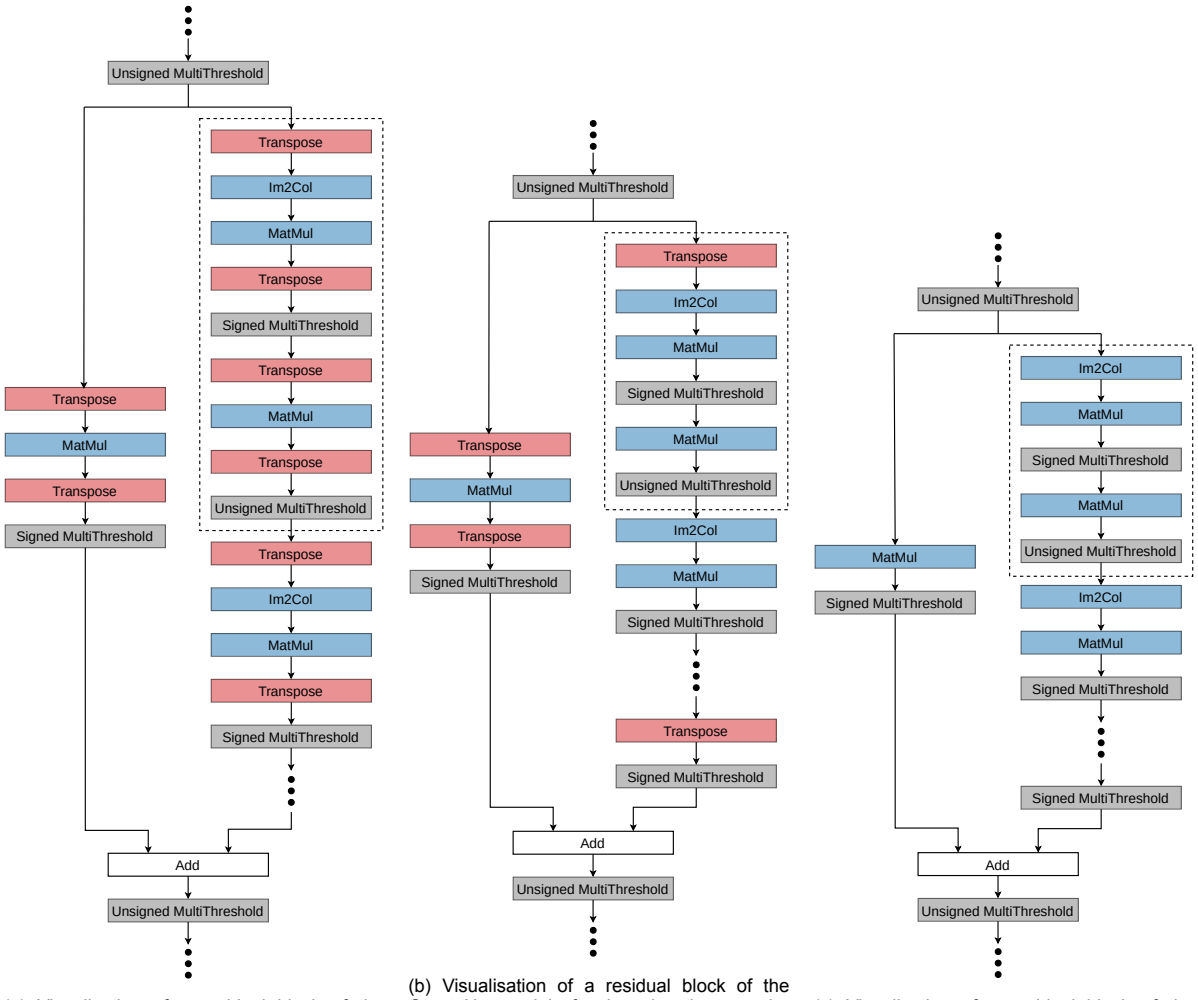
Note that the custom ONNX MultiThreshold operator can operate on a tensor of both layouts, but it is preferred to have this node to output a tensor with $[N, H, W, C]$ format as that matches with the

actual HLS kernel for that node. To simplify the design, the Transpose operators can be optimised away; a Transpose operator converting the tensor layout from $[N, C, H, W]$ to $[N, H, W, C]$ followed by another Transpose operator restoring the original tensor layout $[N, C, H, W]$ can be removed from the graph. Furthermore, a Transpose operator converting the tensor layout from $[N, H, W, C]$ to $[N, C, H, W]$ followed by a MultiThreshold operator and another Transpose converting the layout from $[N, C, H, W]$ to $[N, H, W, C]$ can be optimised away by ensuring that the MultiThreshold node internally operates on the right format. Note that these graph transformations already exist and by applying them, we end up with Figure 3.3b. Due to the residual architecture of QuartzNet, several Transpose operators are still present in the graph. This required three additional transformations to be added:

1. A graph transformation to move Transpose nodes past MultiThreshold nodes.
2. A graph transformation to move a Transpose node past a node that 'joins' two lanes; which in this case is an Add node as shown in Figure 3.3b.
3. A graph transformation to move a Transpose node in front of a node that 'forks' two lanes; which in this case is the (first) MultiThreshold operator as shown in Figure 3.3b.

The idea of all of the proposed graph transformations is to move the Transpose nodes to the MultiThreshold layers in front and after the residual block as shown in Figure 3.3b. As QuartzNet consists of a sequence of such residual blocks, what will effectively happen is that the MultiThreshold operators will be surrounded by two opposing Transpose operators. As explained above, these Transpose operators can then be optimised away.

By partitioning the graph into multiple sub-models, applying the convolution lowering transform, moving the Transpose nodes to appropriate places in the graph and finally, optimising the Transpose nodes away, we end up with a graph where the bulk consists of solely convolutional and MultiThreshold operators as shown in Figure 3.3c.



(a) Visualisation of a residual block of the QuartzNet model after lowering the convolutions.

(b) Visualisation of a residual block of the QuartzNet model after lowering the convolutions and optimising several Transpose nodes away.

(c) Visualisation of a residual block of the QuartzNet model after lowering the convolutions and optimising all Transpose nodes away.

Figure 3.3: Overview of the nodes in QuartzNet before and after optimising Transpose nodes away.

3.4. Converting HLS layers

Converting the ONNX layers to HLS layers is done before the layers are folded. An extensive design space exploration regarding the folding for the ConvolutionInputGenerator (also referenced as Sliding Window Unit, SWU), StreamingFCLayer_Batch (also references as Matrix Vector Activate Unit, MVAU), and VectorVectorActivate_Batch (also referenced as Vector Vector Activate Unit, VVAU) is presented in Chapter 4. In this section, the focus is on discussing the design challenges in order to enable converting the ONNX layers to another set of ONNX layers with appropriate methods for generating the HLS kernels. To be more precise, after streamlining the floating point operations, lowering the convolutions and optimising the Transpose operators away, the network should consist of a series of nodes that have an equivalent C++ kernel description for HLS in the FINN hls-library. In this stage of the FINN flow, the resulting nodes, or ONNX operators, are used to infer a C++ description of the corresponding operator. This is achieved by converting each node to another so-called FPGA dataflow node, which is essentially a wrapper around the FINN custom operator class for ONNX nodes with several additional attributes and methods that are used to infer the right HLS description for that kernel. When converting these ONNX layers to HLS layers, there are two things that should be taken care of. First, the right graph transformation should be present to ensure that the HLS layers are inferred correctly, as shown for the Conv operator in Section 2.7; i.e. the node must be inserted in the right place in the graph and the correct attributes must be inferred. Secondly, the FINN hls-library must contain an HLS description that matches with the ONNX operator. The former aspect will be discussed in Section 3.4.1. Regarding the

latter aspect, the motivation, design challenges and implementation of a custom SWU will be explained in more detail in Section 3.4.2.

3.4.1. Connecting the ONNX layers to HLS layers

As mentioned before, QuartzNet consists of 1D convolutions, while the FINN compiler only supports 2D square convolutions. Section 3.1 described how 1D ONNX nodes are converted to 2D ONNX nodes by extending the attributes and inferring a dummy dimension for 1D tensors to ensure that the graph transformations in FINN are compatible with the layers in QuartzNet. The C++ (HLS) implementations for a non-square SWU already exist in FINN. However, the transformations that infer these C++ kernel templates assume that the layers are 2D. Thus, the graph transformations for inferring a SWU, VVAU and Thresholding_Batch are extended to account for non-square images and attributes. These changes can be found merged in the open-source GitHub FINN repository [36].

3.4.2. 1D Sliding Window Unit HLS implementation

The FINN hls-library contains an SWU implementation for a non-square im2col algorithm. As a 1D convolution can be considered as an edge case of a non-square convolution, the first attempt to realising an SWU for a 1D convolution was to infer the non-square SWU with one of the dimensions of the image set to 1. However, this resulted in an overhead in memory utilisation. To understand why, we have to take a closer look at the internals of the SWU implementation. From a functional perspective, the SWU is responsible for re-arranging the stream of input samples into a particular sequence that corresponds to the so-called lowered image as explained in Section 2.6.2. From an implementation perspective, this is achieved by buffering the input image in an intermediate buffer and subsequently indexing elements from the buffer to write it to an output stream. The memory overhead is caused by how this intermediate buffer is allocated. Assume that on each cycle, a single value corresponding to the input image is read. The input image has a size of $[IFM_H, IFM_W, IFM_C]$ and values are read starting from the innermost (IFM_C) dimension. A naive implementation is to buffer the entire input image and then implement pointer logic to select the indices to produce the expected stream of output values. The disadvantage is that this approach requires the intermediate buffer to store the entire input image, which will incur a large cost in terms of memory utilisation. As hardware memory resources are scarce, the intermediate buffer should be made as small as possible. For a kernel size of $[K_H, K_W]$, the non-square SWU stores $K_H + 1$ rows of the input image along all channel dimensions in the intermediate buffer. After K_H rows of the input image are buffered, subsequent cycles can buffer the incoming data samples as well as produce output values by indexing which buffer entries should be written to the output stream. However, note that for a 1D convolution, the image and kernel are assumed to be flat along the height dimension (H). Thus, the intermediate buffer in the non-square SWU implementation is sized in such a way to account for 2 rows of the input image along all channel dimensions; i.e. effectively, the intermediate buffer is sized in such a way that it can store the entire input image twice. To overcome this overhead, a custom SWU for a 1D convolution needs to be constructed.

The author of FINN hls-library has provided an implementation of an SWU for a regular 1D convolution. The intermediate buffer in this implementation is sized in such a way to store the entire input image of size $[1, IFM_W, IFM_C]$. As QuartzNet contains 1D depthwise-separable convolutions, the proposed SWU is extended to support depthwise convolutions and dilated depthwise convolutions. These changes only involved how the pointer logic indexes the buffer. As an example, consider an input image of dimensions $[1, IFM_W, IFM_C]$ and a kernel of dimension $[1, K_W]$. $SIMD$ dictates the width of the input stream; i.e. the number of input/output channels read/produced in parallel. For simplicity, assume that $SIMD$ is equal to 1. Figure 3.4 shows a simplified overview of how the SWU operates on this particular example for a regular convolution as well as a depthwise convolution. The input image is streamed to the SWU and the SWU stores the entire image in the intermediate buffer. The arrows indicate which output values from the intermediate buffer are written to the output stream. Note that the leftmost value in the output stream indicates the most recent value produced. The highlighted box in red indicates the input pixels that are used to produce an output pixel in the convolutional operation. In this particular example, $IFM_W = 5$, $K_W = 3$, $OFM_W = 3$ and $IFM_C = 3$.

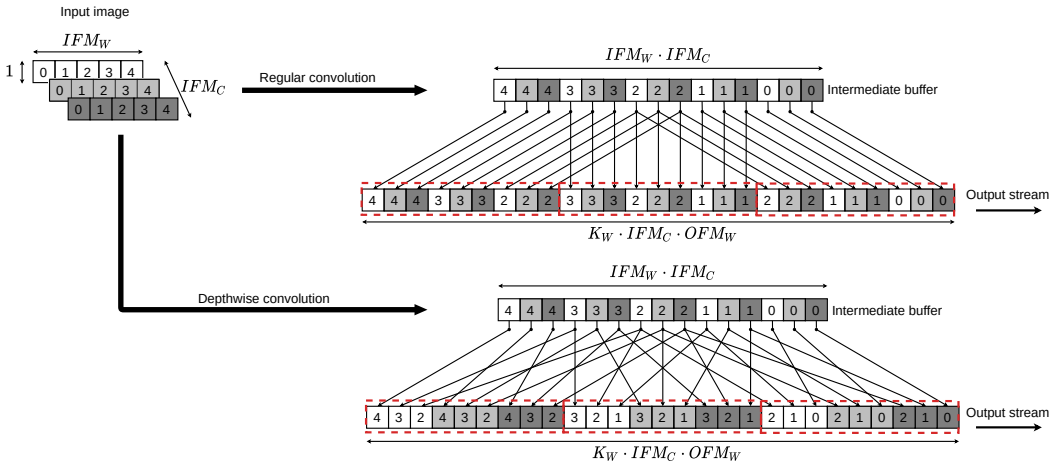


Figure 3.4: Simplified overview of a 1D SWU for regular and depthwise convolutions.

The main take-away points from this implementation are:

1. First, $K_W \cdot IFM_C$ cycles are required to fill a portion of the intermediate buffer with input values. Secondly, $OFM_W \cdot K_W \cdot IFM_C$ cycles are needed to generate the output stream. Note that while producing the output stream, subsequent input samples could be stored in the buffer, which hides the latency. The upper bound on the latency in cycles is shown in Equation (3.6).

$$\text{Latency} = K_W \cdot IFM_C + OFM_W \cdot K_W \cdot IFM_C \tag{3.6}$$

2. The intermediate buffer has a size in bits as shown in Equation (3.7). IFM_W, IFM_C, and IFM_B refer to the input image width, number of input channels, and the number of bits used to represent the input values respectively.

$$\text{Intermediate buffer size} = IFM_W \cdot IFM_C \cdot IFM_B \tag{3.7}$$

Note that buffering the entire input image allows for relatively simple pointer logic to read values from the intermediate buffer and write them to the output stream. However, as the kernel is sliding over the input image, certain values in the intermediate buffer are redundant. For example, for both the regular and depthwise convolution, note that the input values 0 along all three channels are only referenced once for the first output pixel, as shown in Figure 3.4. Similarly, the input values 1 along all three channels are referenced only for the first and second output pixel. This pattern can be exploited to reduce the size of the intermediate buffer significantly by overwriting input values in the intermediate buffer that are redundant after the relevant output pixels are produced. As memory resources are scarce on an FPGA, it is worth examining the pros and cons of implementing an SWU with a smaller intermediate buffer. More specifically, the trade-off between reducing the size of the intermediate buffer and the overhead in resources, caused by for example more complicated pointer logic, to efficiently store and read indices from the intermediate buffer will be analysed in depth in Chapter 4. The rest of this section will discuss the internal details of a so-called optimised buffer SWU implementation.

Optimised buffer SWU implementation

As the QuartzNet model contains only depthwise and pointwise convolutions, this discussion will only focus on the implementation of an optimised buffer implementation for a 1D SWU for depthwise convolution. The analysis can be easily extended to regular convolutions as well. Furthermore, for simplicity $SIMD$ is assumed to be 1, but the proposed algorithm extends for $SIMD > 1$ as well. The proposed implementation can also be found in the open-source GitHub repository of FINN hls-library [40].

For the 1D SWU described above, the intermediate buffer can store the entire input image, which imposes a large memory resource overhead as explained above. The question is, what is the smallest possible size for the intermediate buffer to still enable an efficient implementation for the SWU. From analysing how the intermediate buffer gets indexed, it was found that the buffer can be made as small

as $K_H \cdot IFM_C \cdot IFM_B$-bits to still allow for a relatively efficient implementation of the algorithm. To see why, note that each output pixel is composed of $K_H \cdot IFM_C$ input samples of IFM_B-bits. For an input image of $IFM_W = 3$, $IFM_C = 3$, and a kernel of $K_W = 3$, a simplified cycle-by-cycle analysis of the proposed SWU is presented in Figure 3.5. Each row on the right side of Figure 3.5 indicates a single cycle. The input stream column indicates which value is being read from the input image; following the convention where the innermost dimension of the input image of size $[1, IFM_W, IFM_C]$ is being read first. The intermediate buffer column visualises the content of the intermediate buffer at each cycle. The output stream indicates which output values are written to the output stream. A pseudo-code variant is shown in Algorithm 1.

The proposed algorithm works as follows. For the first $K_H \cdot IFM_C$ cycles, the intermediate buffer is reading from the input stream and buffering the input value. Note that after $(IFM_C - 1) \cdot (K_W - 1)$ cycles, the first output value can be written to the output stream with the idea to produce a continuous stream of output pixels. After the buffer is partly filled, output values are already being generated, with the idea of explicitly hiding the latency between filling the intermediate buffer and producing output values. The red and green highlighted boxes in Figure 3.5 indicate which entry of the buffer is written or read respectively. The highlighted boxes indicate how the write and read pointer logic traverses over the intermediate buffer. From Figure 3.5, note that there is regular pattern between reading/writing data from/to the intermediate buffer. Furthermore, note that the input stream needs to be halted for a certain number of cycles such that entries are freed up in the intermediate buffer. This will likely cause an overhead in resources as these input values need to be stored somewhere; in Chapter 4, it will be clear that FFs seem to be used for this purpose. The latency and buffer size of the proposed algorithm are:

$$\text{Latency} = IFM_C \cdot (K_W - 1) - (K_W - 1) + OFM_W \cdot K_W \cdot IFM_C \tag{3.8}$$

$$\text{Intermediate buffer size} = K_H \cdot IFM_C \cdot IFM_B \tag{3.9}$$

Algorithm 1 Proposed SWU algorithm in pseudo-code. HLS implementation can be found in Appendix B.

```
for Every cycle do
    if Intermediate buffer not partly filled then
        Read from stream and write to intermediate buffer
    else
        if Input image is not entirely read OR redundant entries from buffer can be overwritten then
            Read from input stream
            Update write index pointer
            Write to intermediate buffer
        end if
        Read from intermediate buffer
        Write to output stream
        Update read index pointer
    end if
end for
```

To summarise, the proposed SWU can theoretically lower the memory footprint of the intermediate buffer by a factor shown in Equation (3.10), obtained by dividing the size of the intermediate buffer of the original 1D SWU and the buffer optimised implementation described above. Note that generally, IFM_W is much larger than K_H. The penalty paid is that additional FFs and logic resources need to be used to ensure that the correct entries from the intermediate buffer are read and written to the output stream. Furthermore, note that the analysis presented above extends for $SIMD > 1$ by simply replacing IFM_C by $\frac{IFM_C}{SIMD}$ and reading/writing $SIMD$ input values in parallel to the input/output stream. The HLS implementation of the custom SWU for depthwise convolutions with details on how the pointer logic is implemented to exploit the regular read/write accesses can be found in Appendix B.

$$\frac{IFM_W \cdot IFM_C \cdot IFM_B}{K_H \cdot IFM_C \cdot IFM_B} = \frac{IFM_W}{K_H} \tag{3.10}$$

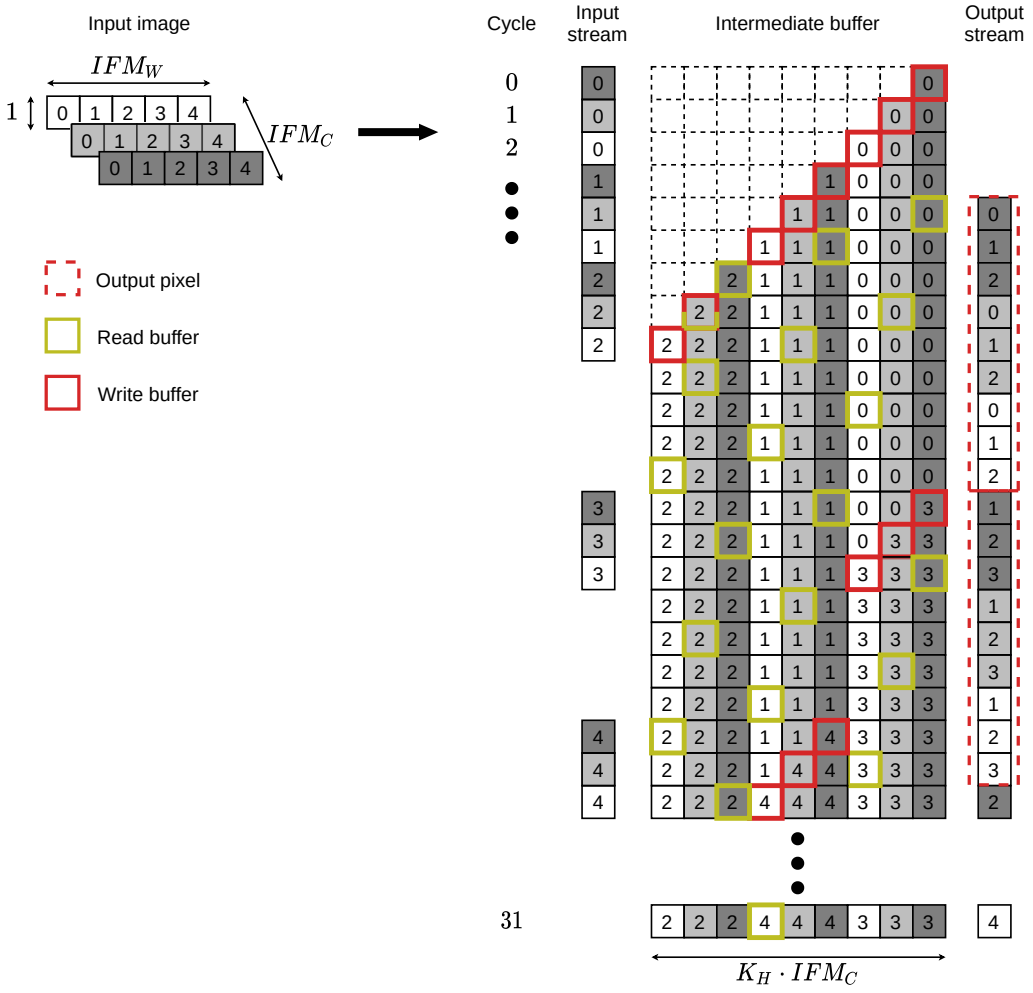


Figure 3.5: Simplified cycle-by-cycle analysis of the proposed SWU algorithm.

4

Design space exploration & analysis

In order to solve the design challenges presented in Section 2.8 and find a configuration for QuartzNet that could fit on an FPGA, this chapter will discuss in more detail the effect on the resource utilisation caused by (un)folding a layer. Figure 4.1 summarises the expected trade-off between the resources and throughput by varying the parallelism of the layers. As explained in Section 2.6.4, the compute resources are expected to scale up when increasing the parallelism of a layer as shown in Figure 4.1. However, despite the fact that the number of weight/threshold values remains constant, how these arrays are realised in hardware by Vivado is not always trivial to predict. Several factors influence the efficiency of the utilisation of hardware memory resources, such as how the weight/threshold array maps to the available hardware resource shape or how the memory hierarchy needs to be constructed to ensure low-latency read/write accesses.

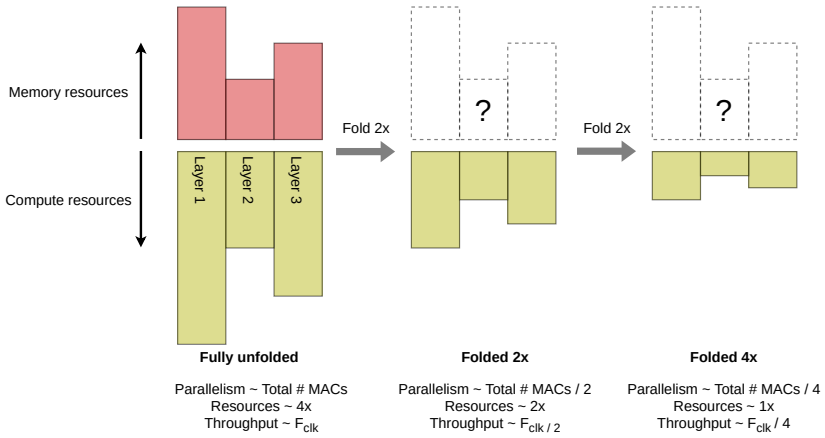


Figure 4.1: Example of trade-off between resources and throughput by varying the parallelism of the layers, inspired and adapted from [4].

The three main building blocks of QuartzNet are:

1. ConvolutionInputGenerator (or SWU): this layer is responsible for lowering the input image.
2. Vector_Vector_Activate_Batch (or VVAU): this layer is responsible for multiplying the weight matrix with the lowered image. Together with the SWU, these two layers form a depthwise convolution.
3. StreamingFCLayer_Batch (or MVAU): this layer is responsible for multiplying the weight matrix with the lowered image. Together with the SWU, these two layers form a regular convolution. Furthermore, this layer is also used to represent a pointwise convolution (without an SWU layer).

From early synthesis experiments, it was also observed that these components are the main contributors to the resource utilisation. Performing an extensive analysis for each individual layer is not feasible

to do. For that purpose, for each of the main building blocks of QuartzNet, a single layer from the QuartzNet model that resides roughly in the middle of the network has been selected for analysis; hence it is neither the largest nor the smallest in terms of size and complexity. It is assumed that the discussions for each of these selected layers can be generalised to hold also for the other layers of the same type, as the layer configuration differences within the QuartzNet model are minor.

In the remainder of this chapter, the folding strategy, as well as other practical limitations to the design space, are elaborated on in Section 4.1. After that, an analysis of the resource utilisation for each of the three aforementioned layers is presented in more detail in Section 4.2, Section 4.3, and Section 4.4. After that, a brief analysis on why FIFOs need to be inserted in the graph will be given in Section 4.5. Based on the analysis presented, the final configuration for the QuartzNet model is summarised in Section 4.6.

4.1. Design space

Before addressing the folding strategy, a brief overview of the entire design space and other practical limitations that limit this design space is presented. For the rest of this chapter, note that certain terms are used interchangeably; a highly folded layer refers to a layer with low parallelism/folding factors.

FINN estimations

For the purpose of exploring the design space, the resource utilisation of FINN HLS layers can be estimated by means of an approximation model constructed from extensive HLS experiments for different configurations of those layers [7]. However, for highly folded layers, these estimations can be inaccurate. Secondly, such a model has not been constructed for the custom 1D SWU yet. Therefore, to obtain accurate utilisation reports, Vivado RTL synthesis results are gathered.

Design space

Note that apart from the parallelism, a user also has the ability to change which hardware resources are used for implementing the weight and threshold memory and arithmetic logic for specific layers. For both the MVAU and VVAU, the multiplications can either be implemented in LUTs or DSPs. Additionally, for an MVAU, the desired memory resource for the weights can be selected as well as whether the weight array will be embedded within the component or whether the memory subsystem is decoupled from the component, as explained in Section 2.7. Decoupling the memory subsystem is achieved by means of an additional weight streamer which supplies the weights to the MVAU component. The weight streamer could be useful for designs where the BRAM resources are utilised poorly, as it allows the memory subsystem to run at a higher clock frequency relative to the rest of the system. This allows multiple addresses of the BRAM module to be supplied to the MVAU within the same clock cycle, meaning that multiple weight buffers can be packed within a single BRAM module and thereby improve the utilisation efficiency of the resource components [35]. It has been shown that the decoupled weight system can improve the BRAM utilisation at the cost of (relatively small) resource overhead. However, running the memory subsystem at a higher clock frequency than the compute is currently not supported in FINN. Thus, the main advantage of the decoupled memory subsystem is mainly that it allows for faster HLS synthesis time and that the memory primitive can be utilised more efficiently as the entire width (including the additional parity bits) can be used to store data. Note that both BRAM and LUTRAM can be embedded within or decoupled from the MVAU component. URAM is only supported when implemented with the memory streamer component, as this type of memory cannot be initialised during bitfile generation and will require an AXI-lite interface to initialise it.

Finally, as the SWU layer buffers the stream of incoming data packets and reorganises the packets accordingly, it is not considered a computational intensive layer. Therefore, the only attribute that is sensible to change is the memory resource type for the intermediate buffer within the layer.

Practical limitations

Aside from the design challenges, there are also a few practical limitations, in most cases specifically related to the QuartzNet model, that make the design space exploration more complex. Before going into detail, the limitations are first briefly highlighted:

1. The implementation of the weights of the VVAU cannot be set to a user-defined hardware resource type.

2. For depthwise convolutions, both the SWU and VVAU must have the same parallelism factor.
3. HLS synthesis of the VVAU consumes a significant amount of system memory. The same holds for an MVAU with weights embedded within the component (*const* memory mode).
4. The Vivado floorplanning algorithm results in a low-performance implementation.
5. Each Xilinx Object file can only have a single outgoing AXI-Lite interface.
6. Large weights of the MVAU cannot be stored in URAM on an Alveo board due to an address range mismatch.

The first limitation is that for the VVAU, the weights cannot be stored in a user-defined hardware resource type. Furthermore, for both the VVAU and MVAU, if the thresholds are embedded within the computational kernel, the hardware resource type cannot be specified. In these cases, Vivado HLS will infer the best possible hardware primitive to store the weights/thresholds. In some cases, this might be useful as the designer does not require to know lower-level details of the hardware resources. However, as will be shown later, the resulting hardware implementation is often not efficient and more control over the inferred hardware memory resource is preferred.

The second limitation is that for depthwise convolutions, both the SWU and VVAU must have the same parallelism factor. Note that by increasing the parallelism of a layer, the width of the output stream ($SIMD \cdot W_B$) is increased, where W_B denotes the number of bits used to represent the input values. For other subsequent layers with different parallelism factors, a mismatch between the width of the streams between two layers can be solved by inserting a StreamingDataWidthConverter (SDWC) node, which essentially converts a sequence of I_{in}-bit input samples to O_{out}-bit output samples by either chopping or concatenating the input packets. However, due to the underlying algorithm of the SWU, the ordering of output pixels from the SWU changes significantly with different levels of parallelism. Therefore, inserting an SDWC node will not solve the problem and hence, the successive VVAU node must process these data packets in the right order. This can only be achieved if the VVAU operates at the same level of parallelism as the SWU.

The other limitations are imposed by the tools that are used. From experiments, it was found that converting the VVAU to an RTL implementation with Vivado HLS took an extreme amount of time (in the order of tens of hours) and memory (in the order of more than 100 GB) for very low parallelism settings. In early experiments, HLS synthesis for several VVAUs could not be done due to out-of-memory errors. The reason for the large HLS synthesis time was due to how the weights array was instantiated. Each weight value would be cast explicitly to a predefined number of bits. However, this did not have any practical advantage and the problem was resolved by disabling the casting. The root cause for the high memory usage was not found and assumed to be due to something internally to the Vivado HLS compiler. A potential solution to the problem would be to decouple the weight array from the VVAU block such that it is not part of HLS synthesis anymore. However, this is considered out of the scope of this work and hence, a workaround had to be found. From experiments, it was found that there is a causation between the shape of the weight array and the memory usage. For very low parallelism factors, i.e. implicitly instantiating very narrow and deep memory arrays, HLS synthesis consumes an excessive amount of system memory. By increasing the parallelism by a factor of 2, which implicitly makes the weight array more wide and shallow, the memory usage would reduce by a factor of 2. Therefore, this limitation implies a lower bound on the parallelism of VVAUs. The same problem was also observed during HLS synthesis of an MVAU with weights embedded inside of the component. To overcome the memory bottleneck during HLS synthesis, the weights of the MVAU will be decoupled from the computational block.

Another limitation is imposed by the Vitis platform and Vivado. In the current flow, subsequent layers that should be placed on the FPGA are stitched together into a single IP block. The IP block is then compiled into a Xilinx Object (XO) file resulting in a total of three XO files: two XO files for the input and output DMA and one XO file for the model. All of the XO files are then linked together (with Xilinx Runtime) to produce the FPGA binary container file. Note that each IP block corresponds to a Compute Unit (CU). During the linking process, a specified amount of CUs are set to be instantiated, the kernel ports are mapped to memory, the CUs are connected to each other, and, the layers within a CU are assigned to SLRs. Determining which layer gets assigned to which SLR in an efficient manner is a non-trivial task and requires attention to several factors; e.g. spreading the resources across the SLRs

evenly, minimising the SLR crossings which minimises the SLLs required, and ensuring that the first and last layer reside on the same SLR as this requires a single DMA engine. There exists a heuristic in the FINN-experimental library where this optimisation problem is framed as cutting a graph into disjoint graphs according to a specific objective function to minimise SLR crossings and constraints such as not exceeding compute and connection resources for each SLR [39]. It is solved by means of Integer Linear Programming. However, this solution relies on the FINN resource estimates for each layer. Those resource estimates are obtained through experiments and regression analysis for the already existent layers, but are missing for the 1D SWU layer. Thus, in its current form, this method cannot be applied. Instead, Vivado tries to schedule the layers within the CU efficiently among the SLRs. However, due to the size of the QuartzNet model, this results in a detrimental impact on the attainable frequency [4]. This was also observed from the generated design; the critical path crosses all four of the SLRs of an Alveo U250 board for a particular reset signal, which severely limits the achievable frequency.

Regarding another limitation imposed by the Vitis platform, note that each XO file can only have a single outgoing AXI-lite interface. Hence, each IP block can contain at most a single MVAU with weights in URAM. For the QuartzNet model, this means that the model needs to be partitioned into multiple sub-models where each sub-model contains at most a single MVAU with weights in URAM. This required adjustments to the particular FINN-ONNX custom operator that encapsulates the FPGA dataflow nodes, named StreamingDataflowPartition, as well as the transformation that infers the StreamingDataflowPartition. In particular, this custom operator was built on the assumption to have a single input and output tensor for the sub-model. As QuartzNet contains residual blocks, this assumption could be broken as particular nodes have multiple inputs and outputs. Despite the implemented adjustment to the StreamingDataflowPartition node and transformation that infers the StreamingDataflowPartition node, other limitations arose that obstructed storing the weights of the MVAU in URAM. The maximum number of XO files that can be linked together is by default set to 60. A user could increase this limit, but pushing the tool beyond its suggested limit could lead to unforeseen problems. Early experiments, where each MVAU would have its weights stored in URAM, lead to a design with 99 XO files. By implementing the AXI SmartConnect IP, a larger number of MVAUs with weights in URAM can be instantiated while still staying within the recommended limits. This IP block allows multiple AXI memory-mapped master devices to connect to multiple memory-mapped slave devices. By grouping multiple AXI-Lite interfaces and presenting a single output AXI-Lite interface, multiple MVAUs with weights in URAM could be encapsulated within the same XO file. Another solution is to implement the weights of the MVAU in either LUTRAM or BRAM as these memories can be initialised during bitfile generation. This reduces the design space, but due to its simplicity, the latter solution is preferred, while the former is kept for future work.

On top of the aforementioned limitation, it was found during experiments with a generated accelerator for QuartzNet that the weights cannot be stored in URAM due to an address space misalignment which caused that the weights cannot be initialised properly. Resolving this problem will require extensive testing and debugging which falls out of the time span of this work. Thus, it is left for future work. The workaround to this problem is to not use URAM for storing the weights of an MVAU component. Despite the limitation, this chapter will also analyse the effect of varying the folding of an MVAU with weights in URAM to provide more insights.

Now that the design space and limitations are introduced, Section 4.2, Section 4.3, and Section 4.4 will discuss the resource utilisation for the SWU, VVAU, and MVAU respectively. It is assumed that the core of this analysis extends to other layers as well.

4.2. Sliding Window Unit

To determine the optimal folding and implementation of the SWU, the memory utilisation is analysed at different levels of parallelism. As explained in Section 3.4.2, a variant of the SWU has been implemented that minimises the size of the intermediate buffer and thereby also the size of the hardware memory utilisation. However, it is expected that this method requires more complicated indexing logic to correctly read and write from the intermediate buffer, as well as additional FFs to temporarily store incoming data packets. In order to find which variant is more suited for QuartzNet, both implementations of the SWU will be analysed. The examined layer has the following attributes:

- Kernel width: 51
- Input channels: 512
- Input image width: 178
- Output image width: 128
- Input precision: 4-bits

4.2.1. Memory utilisation

Figure 4.2 visualises the LUTRAM, BRAM tile, and URAM utilisation for both variants of the algorithm. A detailed table can be found in Appendix A.1. Note that in all three cases, the optimised algorithm reduced the memory footprint by roughly a factor of 3. This is conform the expected reduction in memory footprint as discussed in Section 3.4.2; the ratio input image width to kernel width is 3.49.

Interesting to note from Figure 4.2 is that for the LUTRAM implementation of weights, the fully folded layer (i.e. when SIMD is equal to 1) has a significantly larger memory footprint compared to unfolded layers for both the base and optimised buffer kernel. For a fully folded layer, the memory shape of the buffer is expected to be narrow (4-bits wide) and deep. The root cause for the inefficient memory utilisation is likely due to how Vivado HLS internally is unable to efficiently implement such a narrow and deep memory array.

Note that the BRAM modules support configurable data width, as explained in Section 2.3. This allows the Vivado tool to tailor the hardware memory shape to match the (synthesised) shape of the intermediate buffer. When the intermediate buffer is implemented in BRAM, the memory utilisation remains constant for all 3 of the algorithms for different levels of parallelism, except for the case where SIMD is 64 for the base kernel implementation. Note that for cases where the layers are unfolded by a relatively large amount, the memories become wider and shallower. This is again assumed to be due to how Vivado HLS internally implements such a wide and shallow memory array. One possible explanation for the base kernel could be due to the fact that a BRAM module has a limited amount of read/write ports per cycle. When $SIMD$ is relatively large, the number of reads/writes to the memory buffer in a single cycle increases. Thus, to still enable low-latency access to the weights, the Vivado tool could potentially decide to implement the weight array spread across multiple BRAM modules, leading to inefficient utilisation of the BRAM module.

When the intermediate buffer is implemented in URAM, there is a clear pattern deducible. This can be explained by looking into the properties of the hardware resource and the underlying algorithm of the SWU. Note that the URAM module has a fixed shape of 72-bits wide and 4096 addresses deep and contains two ports [75]. Each port can perform either one read or write operation per clock cycle. As explained before, the baseline SWU instantiates a relatively large intermediate buffer to store the input image. The entire input image contains $IFM_W \cdot IFM_C \cdot IFM_B$-bits, where IFM_W, IFM_C, IFM_B denote the input image width, number of input channels, and bit-precision respectively. The intermediate buffer contains $\frac{IFM_W \cdot IFM_C}{SIMD}$ entries of $SIMD \cdot IFM_B$-bits. Note that after the buffer has been filled, in each cycle one specific entry of $SIMD \cdot IFM_B$-bits is read out and written to the output stream. This can also be seen from the results, where the number of URAM modules is given by Equation (4.1) for cases where $SIMD \cdot IFM_B < 72$.

$$\frac{IFM_W \cdot IFM_C \cdot IFM_B}{SIMD \cdot IFM_B \cdot 4096} = \frac{IFM_W \cdot IFM_C}{SIMD \cdot 4096} \tag{4.1}$$

Note that if $SIMD \cdot IFM_B > 72$, the entry of the intermediate buffer exceeds the width of the URAM module. Depending on whether the desired access latency can be reached, Vivado potentially partitions this long word into multiple URAM modules.

The most efficient URAM utilisation is achieved by ensuring that $SIMD \cdot IFM_B$ equals the full width of the URAM module or an integer multiple thereof. With the current form of the HLS algorithm, the user must ensure that this condition is met.

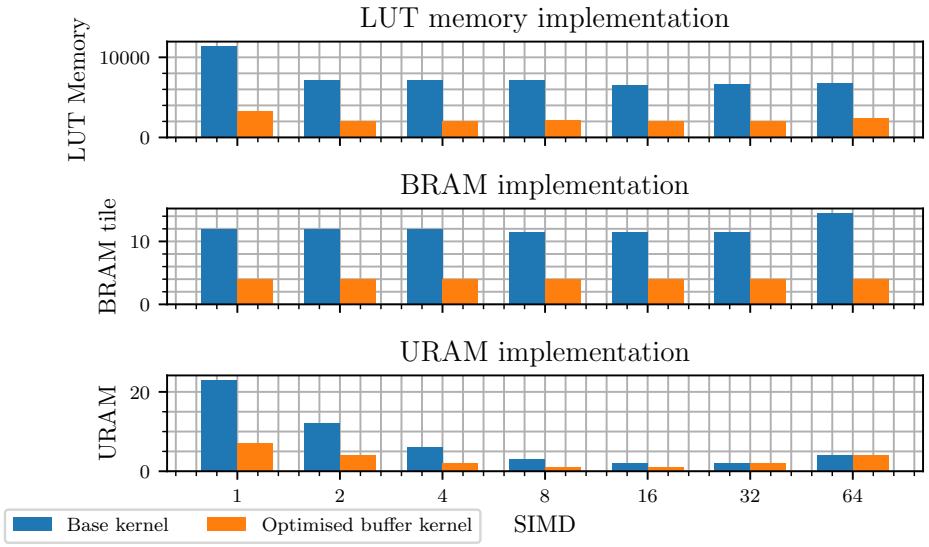
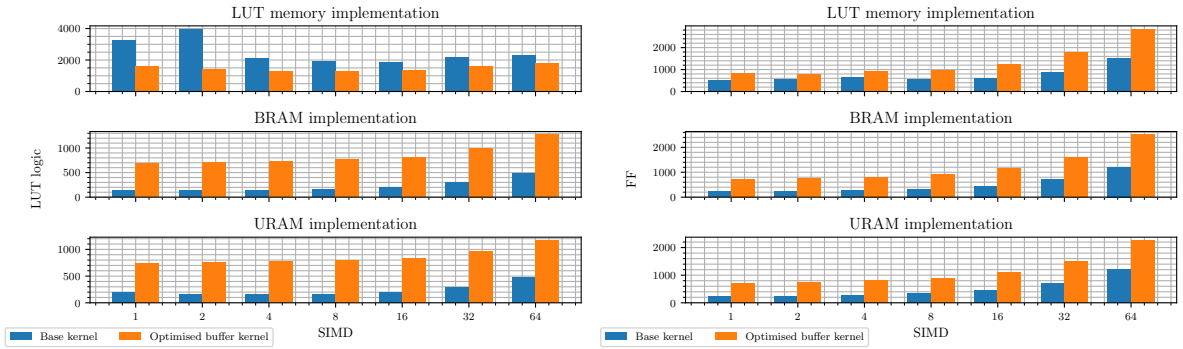


Figure 4.2: SWU memory utilisation reported from RTL synthesis post-implementation and route reports for various hardware primitives and SIMD factors.

4.2.2. LUT logic & FF utilisation

From Figure 4.2, we can conclude that the memory utilisation is indeed lowered with the optimised SWU kernel. However, the implementation requires more complicated accesses to the intermediate buffer and additional storage elements (FFs), which translates to additional logic used. Figure 4.3a and Figure 4.3b show the LUT logic and FF utilisation for both kernels for various SIMD factors. Note that in all 3 of the cases, the optimised kernel requires more FFs. Note that for the cases where the intermediate buffer is implemented in BRAM and URAM, the LUT logic utilisation is higher for the optimised buffer kernel. However, when the buffer is implemented in LUTRAM, the LUT logic utilisation is lower for the optimised buffer kernel. The root cause for this observation is assumed to be due to how Vivado HLS internally implements the weight array in LUTRAM.

To summarise, the optimised buffer kernel is preferred if the memory utilisation is the main bottleneck in realising an FPGA design. However, it will incur an overhead in FFs and, in case the buffer is implemented in BRAM or URAM, also in LUT logic. Furthermore, in case the buffer is implemented in URAM, the designer must ensure that the width of the URAM module gets utilised efficiently by tuning the parallelism.



(a) LUT logic utilisation.

(b) FF utilisation.

Figure 4.3: SWU LUT logic and FF utilisation obtained from RTL synthesis post-implementation and route reports.

4.3. Vector Vector Activate Unit

As discussed earlier, the VVAU and SWU must operate on the same level of parallelism by having the same PE and SIMD factors respectively. The VVAU utilises two resources mainly: memory for storing the weights and thresholds, and LUT logic or DSPs for performing the vector-vector product. As noted in Section 4.1, the current HLS kernel allows us to only modify which resource type is used to implement the vector-vector product. The examined layer has the following attributes:

- Kernel width (InnerProdDim): 51
- Input channels: 512
- Output image pixels (numReps): 128
- Input precision: 4-bits
- Weight precision: 4-bits

4.3.1. Memory utilisation

Figure 4.4 visualises the BRAM tile utilisation for various parallelism settings. From Figure 4.4, the BRAM utilisation seems to decrease until PE is equal to 8 and for PE greater than or equal to 8, the BRAM utilisation increases by a factor of two for every doubled parallelism factor. By inspecting the synthesis log files, we can find an explanation for this remarkable utilisation. For PE=2 and PE=4, the threshold values are implemented in BRAM, while for PE=8, the thresholds are implemented by means of LUTs and FFs. Thus, for different folding factors, the Vivado HLS tool infers a different hardware resource to implement the thresholds. To understand why this happens, we have to look at how the Vivado tool synthesises the thresholds in hardware memory. Note that each PE applies a vector-vector product and the activation function. As explained in Section 3.2, the activation function is expressed as a threshold operation; for each output channel, the output of the vector-vector product is compared against a set of thresholds and the final output represents the number of thresholds that the output product is greater than or equal to. In the specific context of this experiment, there are 512 output channels (IFM_C) and 15 thresholds (TH) for each output channel. When applying the activation function, each PE will therefore require 15 threshold values. To achieve the lowest latency, all of these threshold values must be fetched within a single clock cycle. Thus, the tool will partition the threshold array into $PE \cdot TH$ modules of IFM_C/PE words of a specific size in bits (depending on the size of the accumulated product) for a total of $IFM_C \cdot TH$ threshold values. Thus, by increasing the parallelism PE, the number of modules are increased and the size of each module (in words) is decreased. For PE=2 and PE=4, the number of bits for each module is relatively large and the Vivado tool determines that the most efficient implementation for the thresholds is BRAM. Note that the BRAMs are highly underutilised (as the product $IFM_C/PE \cdot T_B$ is much smaller than the size of RAMB18 unit, where T_B is the number of bits used to represent the threshold value); $\approx 1200/18432$-bits are utilised for a RAMB18. For PE=8 and larger, the number of bits for each module is relatively small and the Vivado tool determines that the most efficient implementation for the thresholds is by means of FFs and LUTs. This can be confirmed from Figure 4.5 as well. Note that the reported LUT logic utilisation increases for both the LUT and DSP logic implementation of the VVAU from PE=8 onward. This is due to the fact that the thresholds can be implemented as read-only and hence, are not implemented as LUTRAM but in 'regular' LUTs that could be used for logic as well. For that reason, the LUTRAM remained unused, while the reported LUT logic number encapsulated both the LUTs used for logic as well as LUTs used as ROM for implementing the thresholds.

Perhaps it could be possible to make a trade-off between latency and how the tool infers the hardware resource. If the initiation interval in the function is not set to 1, the tool could possibly infer something more efficient in terms of memory resources. However, for low-latency applications that are targeted, this is not prioritised. Another possible method to overcome this problem is to implement the threshold comparison in a separate layer following the VVAU. This layer, also called Thresholding_Batch, effectively decouples the threshold array from the VVAU and allows a user to select which hardware primitive should be used to implement the threshold array.

From the synthesis reports, it was observed that the weights are implemented in BRAM. Note that each output channel has a separate filter. In this specific context, there are 512 (IFM_C) input/output

channels, the 1D kernel has a length of 51 (K_W) and each kernel value is expressed with 4 bits (W_B). To achieve the lowest latency, each PE should be able to fetch the weights independently from other PEs (i.e. there are no stalls due to the fact that the BRAM module has a limited number of read/write ports). Thus, the memory array is implemented as PE chunks/modules of $IFM_C \cdot K_W / PE$ words of size W_B-bits. As the size of each module is relatively large, each module is implemented in BRAM. Thus, the number of utilised BRAMs increases linearly with the number of instantiated modules. Note that this only holds when the number of bits within each module does not fully occupy a single RAMB18 unit. For example, for PE=4, there are 4 modules of each 26112 bits and hence 2 RAMB18 ($2 \cdot 18432 = 36864$-bits) units are used for each module to store all of the bits. For PE=8, there are 8 modules of each 13056 bits and hence 1 RAMB18 (18432-bits) is used for each module. Thus, the optimal BRAM utilisation is achieved by ensuring that $IFM_C \cdot K_W \cdot W_B / PE$ is as close as possible to the size of a RAMB18 unit (18432 bits) or an integer multiple thereof. In this specific experiment, PE=4 and PE=8 result in the most efficient BRAM utilisation.

Improving the BRAM utilisation efficiency could be achieved by decoupling the weight memory from the compute kernel and running the weight subsystem at a higher clock than the compute, similar to what is proposed for the MVAU in [35]. This would allow the tool to implement different modules within the same BRAM instance without sacrificing the throughput of the system. However, as this is not yet supported in FINN, this is considered out of the scope of this work.

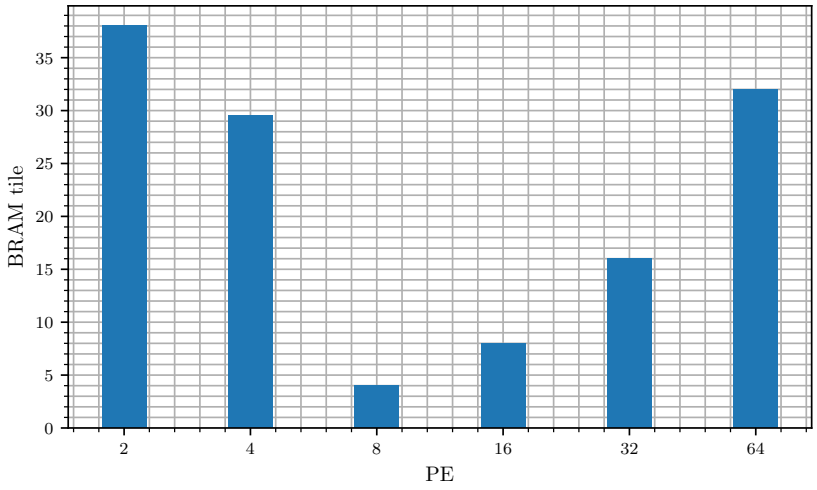


Figure 4.4: VVAU BRAM tile utilisation obtained from RTL synthesis post-implementation and route reports.

4.3.2. LUT logic & DSP utilisation

Figure 4.5 visualises the LUT logic and DSP utilisation for various parallelism steps for both resource type implementations of the kernel. A detailed table of the resource utilisation can be found in Appendix A.2. As the number of PEs doubles, the resource utilisation increases by roughly a factor of 2. Note that we would expect the number of DSPs to be equal to the number of Processing Engines; i.e. the number of multiplications performed in parallel. By inspecting the log files, it was noted that one of the MAC operations could not be implemented in DSPs. The exact reason for this error is not found, but expected to be something internal to the Vivado HLS compiler. For that reason, the number of DSPs is thus equal to the number of PEs minus 1.

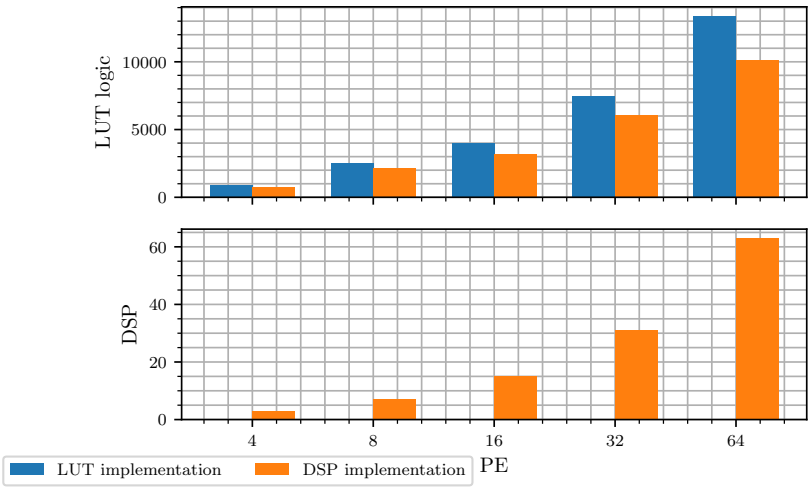


Figure 4.5: VVAU LUT logic and DSP utilisation obtained from RTL synthesis post-implementation and route reports.

4.4. Matrix Vector Activate Unit

For the MVAU, there are two parameters that govern the parallelism: the number of PEs and the number of SIMD lanes within each PE. As explained in Section 2.6.2, an SWU and MVAU essentially perform a convolution operation by lowering the image and then performing a matrix multiplication respectively. Note that the result of a matrix multiplication is composed of multiple vector-vector multiplications. Referring to this analogy, the SIMD parameter determines the parallelism of the vector-vector multiplication; i.e. the number of multiply-and-accumulate operations performed in parallel. The PE parameter determines how many of such vector-vector multiplications are performed. The latency of the matrix multiplication, as shown in Figure 2.14 in Section 2.6.4, is given by Equation (4.4), where N_F and S_F indicate the so-called neuron and synapse fold respectively. OFM_C, IFM_C, OFM_H, OFM_W, K_H, K_W refer to the number of output and input channels, output image height and width, and kernel height and width respectively. Note that both the SIMD and PE factors contribute equally to a reduction in the latency, which raises the question: what is the difference in terms of resource utilisation between the two parameters? This will be examined in further detail in Section 4.4.3.

$$N_F = \frac{OFM_C}{PE} \tag{4.2}$$

$$S_F = \frac{K_H \cdot K_W \cdot IFM_C}{SIMD} \tag{4.3}$$

$$\text{Latency} = N_F \cdot S_F \cdot OFM_H \cdot OFM_W \tag{4.4}$$

The examined layer in this experiment has the following attributes:

- Height of weight matrix (MH): 512
- Width of weight matrix (MW): 512
- Output image pixels (numReps): 128
- Input precision: 4-bits
- Weight precision: 4-bits

The SIMD and PE parallelism influences the logic and memory resources in different ways. Compared to the VVAU, the MVAU has a larger design space to explore. A designer can choose to implement the MAC operations in either LUTs or DSPs. Furthermore, a designer can choose which hardware resource type to use for the weight memory and whether or not the weight subsystem is decoupled from the compute kernel. As the memory resources are the main bottleneck in fitting the design on an Alveo

board, the main focus of the following analysis will be on the relationship between the scaling factors and the hardware memory utilisation. From early experiments, it was also found that a shortage of DSPs is less likely to become a bottleneck compared to a shortage of LUT logic. For the final design, the MAC operations will be implemented with DSPs in an attempt to reduce the LUT logic utilisation. Thus, for the first part of this analysis, it is assumed that the MACs are implemented in DSP units. A complete table, from which all of the graphs below are obtained, can be found in Appendix A.3; it contains for both the LUT and DSP implementation the estimated resources.

4.4.1. LUT logic utilisation

Figure 4.6, Figure 4.7, and Figure 4.8 visualise the logic utilisation for various folding factors for a configuration with weights implemented LUTRAM, BRAM, and URAM respectively. For each figure, the SIMD and PE scaling is examined in isolation; i.e. SIMD/PE is increased while PE/SIMD is kept equal to 1. In terms of logic utilisation, note that increasing SIMD and PE is directly related to the number of DSP blocks as can be seen from Figure 4.6, Figure 4.7 and Figure 4.8. Note that a similar error appeared for the VVAU where one particular MAC could not be implemented with a DSP block, which is also assumed to be caused by how Vivado HLS internally infers and implements a DSP. Furthermore, note that the LUT logic utilisation remains roughly constant while scaling SIMD. On the contrary, scaling PE increases the LUT logic utilisation significantly. This holds for all of the three different implementations of the weight array in hardware. This observation is examined in further depth in Section 4.4.3.

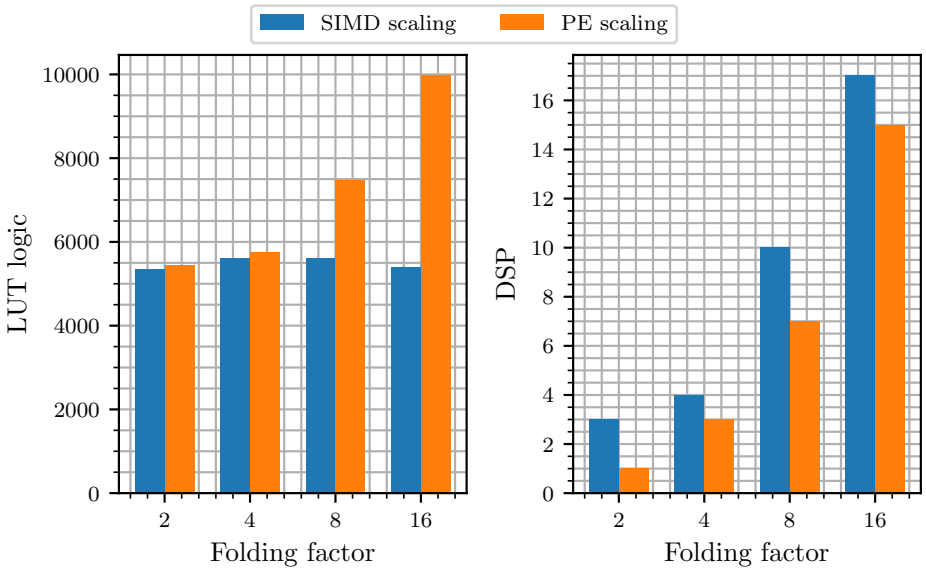


Figure 4.6: LUT logic utilisation for MVAU with decoupled weights in LUTRAM obtained from RTL synthesis post-implementation and route reports.

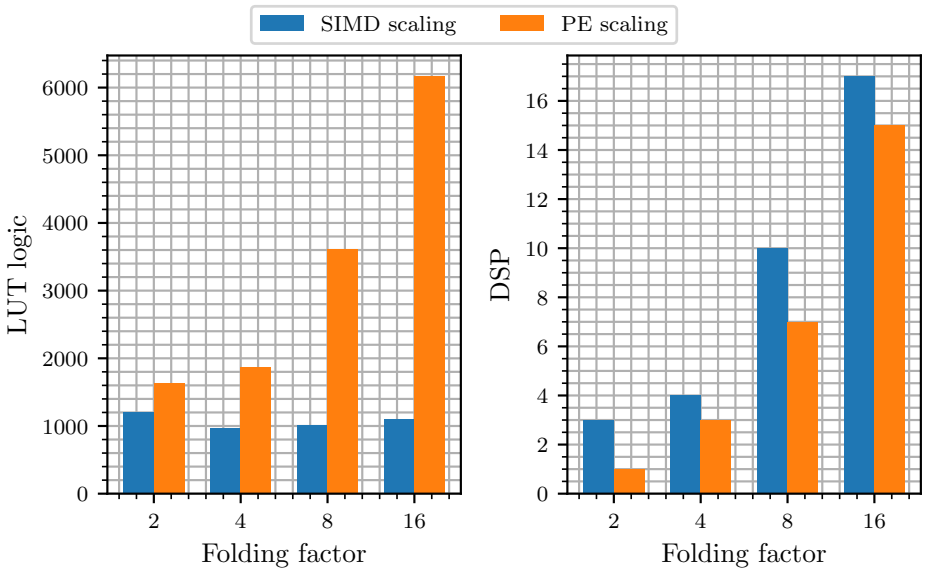


Figure 4.7: LUT logic utilisation for MVAU with decoupled weights in BRAM obtained from RTL synthesis post-implementation and route reports.

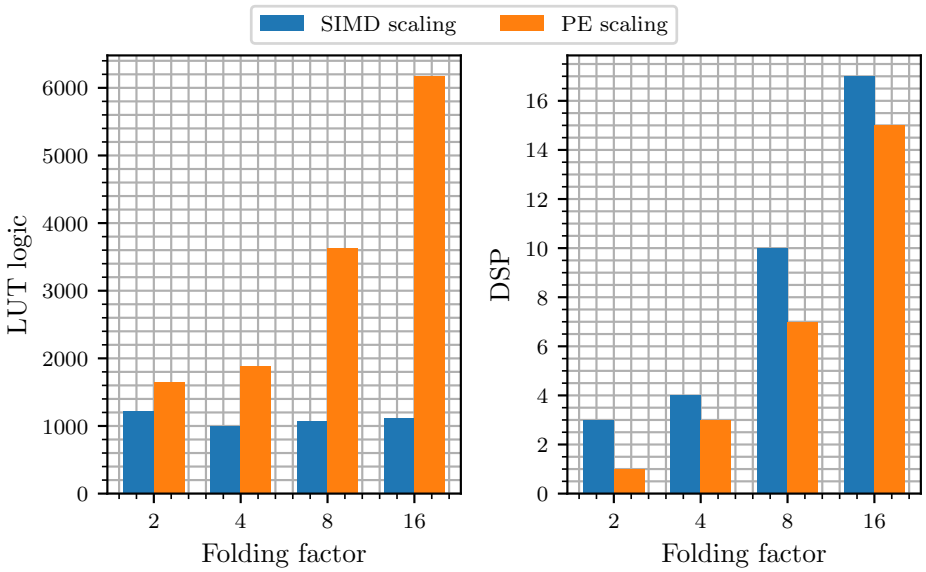


Figure 4.8: LUT logic utilisation for MVAU with decoupled weights in URAM obtained from RTL synthesis post-implementation and route reports.

4.4.2. Memory utilisation

Another interesting observation can be made when analysing the memory utilisation. Figure 4.9, Figure 4.10, and Figure 4.11 visualise the relevant memory utilisation for various folding factors for a configuration with weights LUTRAM, BRAM, and URAM respectively. For each plot, the SIMD and PE scaling is visualised; i.e. SIMD/PE is increased while PE/SIMD is kept equal to 1.

Note that the BRAM utilisation for all three of the implementations seems to depend highly on the number of PEs, while it remains constant when scaling SIMD. A similar observation was made for the VVAU and the same explanation holds. Note that, by keeping PE constant, increasing the number of parallel MACs does not influence the partitioning of the threshold array. However, as each PE requires a set of thresholds to be fetched within a single clock cycle, the Vivado HLS tool will partition the threshold array in a specific shape to assure that this condition is met. For a small amount of PEs, the threshold

array is partitioned in relatively large modules that are mapped to BRAMs. For a larger amount of PEs, in this case when PE>=8, the threshold array is partitioned into relatively small modules for which the Vivado tool infers LUTs and FFs. This can also be seen from Figure 4.12a, Figure 4.12b, and the right plot in Figure 4.10; the FF utilisation increases by a significant factor for PE>=8 for the case where PE scaling is applied. Additionally, from Figure 4.6, Figure 4.7, and Figure 4.8 it can be observed that the LUT logic utilisation also increases by a significant factor for PE>=8. Furthermore, from the synthesis report generated by out-of-context synthesis with Vivado, the additional LUTs used to implement the thresholds does not seem to be reported under the LUTRAM utilisation. Again, the same explanation holds as for the VVAU; the thresholds are implemented as ROM and hence, 'regular' LUTs instead of LUTRAM can be used. For this reason, the LUTs are captured under the 'LUT as logic' section in the utilisation report.

For the implementation with weights in LUTRAM, the parallelism factors do not alter the utilisation significantly. Note that for PE scaling, the LUTRAM utilisation is higher compared to SIMD scaling. This is assumed to be caused by how the weight array is partitioned and implemented in hardware. The observation that the number of LUTRAMs seems relatively equal among the folding factors aligns with what is expected from LUTRAMs; due to the fine-granularity of LUTRAMs as explained in Section 2.3.2, the hardware memory resource maps well to the shape of the weight array.

For the implementation with weights in URAM, note that higher folding factors result in more efficient memory utilisation. The same analysis as for the SWU holds. By increasing the parallelism factor, either the number of PEs or SIMD lanes, the width of the entries in the URAM are increased as well. As long as the width of the weight memory entry is smaller than the width of a URAM module, which is indeed the case for relatively small folding factors, the width of the URAM module gets utilised more efficiently.

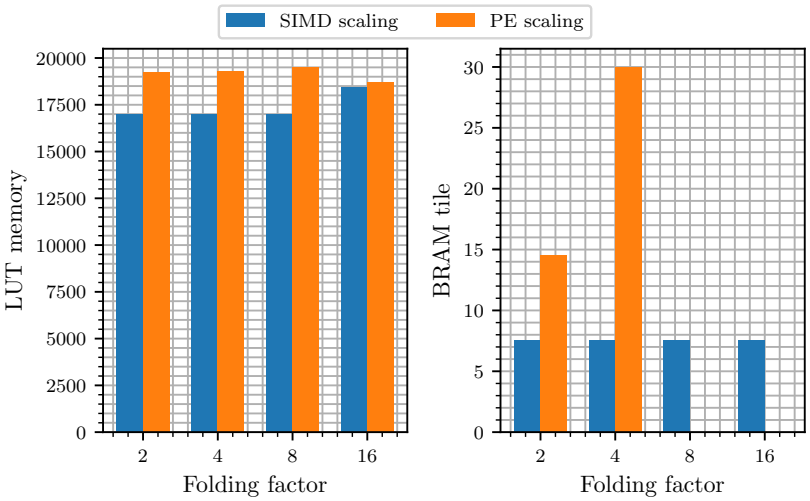


Figure 4.9: Memory utilisation for MVAU with decoupled weights in LUTRAM obtained from RTL synthesis post-implementation and route reports.

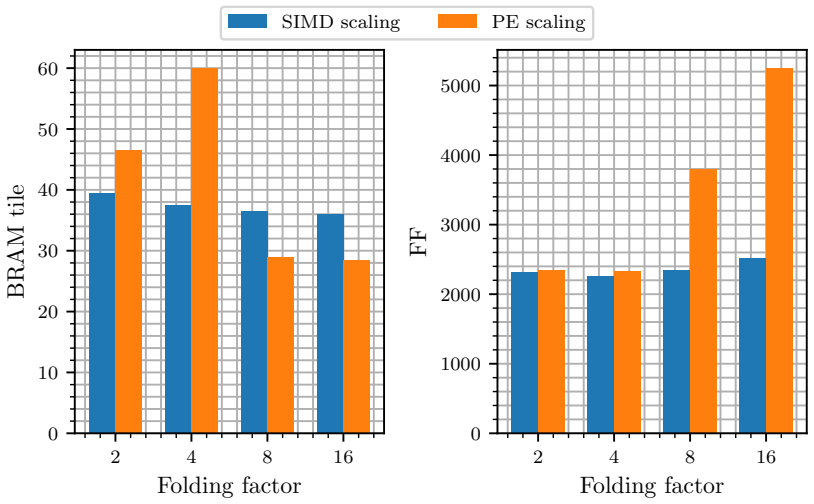


Figure 4.10: Memory utilisation for MVAU with decoupled weights in BRAM obtained from RTL synthesis post-implementation and route reports.

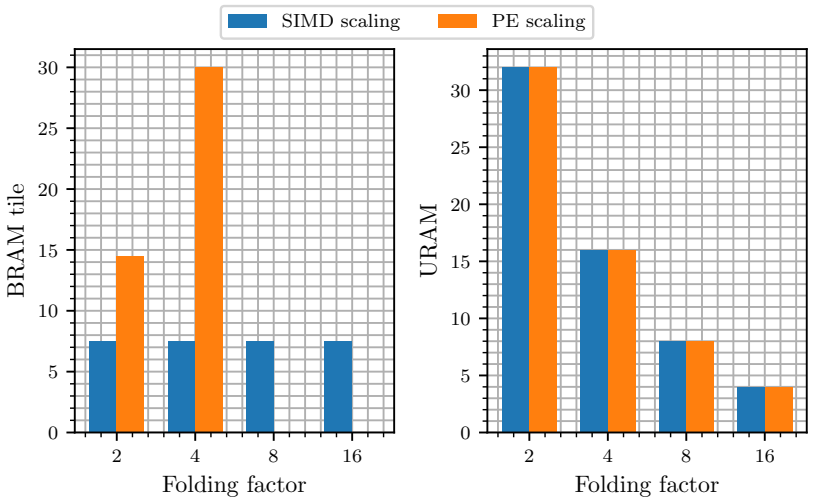
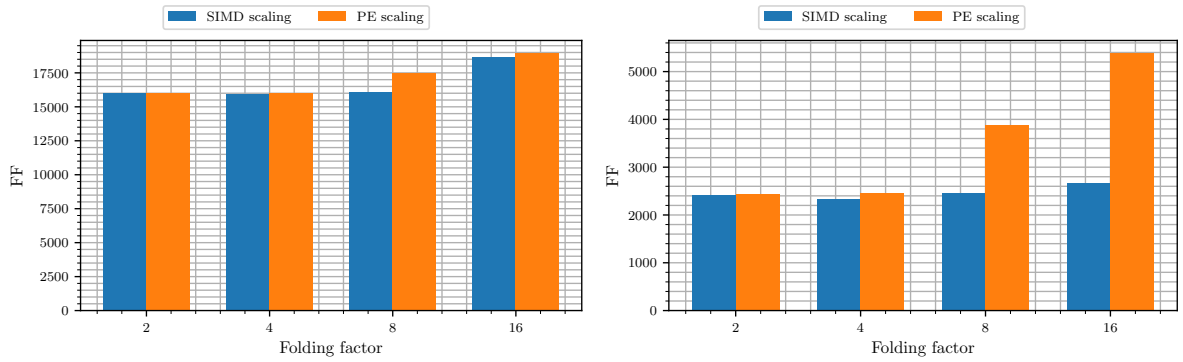


Figure 4.11: Memory utilisation for MVAU with decoupled weights in URAM obtained from RTL synthesis post-implementation and route reports.



(a) FF utilisation for MVAU with decoupled weights in LUTRAM.

(b) FF utilisation for MVAU with decoupled weights in URAM.

Figure 4.12: FF utilisation obtained from RTL synthesis post-implementation and route reports.

4.4.3. PE versus SIMD folding for an MVAU

As noted, there are multiple ways to unfold the matrix computation to achieve a specific latency target. Early experiments and estimations have shown that the VVAUs in the QuartzNet model achieve the lowest BRAM utilisation at a specific folding rate that results in a latency of (roughly) less than 600 000 cycles. Thus, for this experiment, a target latency of 600 000 cycles for the MVAU will be considered. By filling in Equation (4.4), there are seven different PE and SIMD configurations that achieve this latency target. However, each of them has different implications on the generated hardware. To simplify the visualisations, the following experiments are all performed with the MACs implemented in LUT logic. Figure 4.13 visualises for each of the seven configurations the differences in LUT logic utilisation and BRAM tile utilisation. Note that the graph is obtained for an MVAU with weights implemented in LUTRAM. Similar observations are made for an MVAU with weights in BRAM or URAM. Furthermore, there are small differences in other resource utilisation metrics, such as FF and DSP utilisation in case MACs are implemented with DSP blocks. However, based on experiments and previous FINN accelerator builds, these blocks are not expected to be as scarce as LUT logic and BRAM tiles. Similar graphs for cases where the weights are implemented in either BRAM or URAM, as well as detailed tables with other types of resource utilisation, can be found in Appendix A.3.

Figure 4.13 shows that BRAMs are utilised only for cases where PE<8. This is in line with our previous observation regarding the Vivado HLS tool inferring BRAMs or LUTROMs for the thresholds depending on the parallelism factor. Furthermore, note that there are significant differences in terms of LUT logic utilisation. As explained before, one of the reasons is caused by the fact that Vivado is reporting the LUTs used for implementing the threshold memory as LUT logic instead of LUTRAM. From the synthesis report, it was however observed that the LUTs used for the thresholds remained roughly constant for higher values of PE and that mainly the FF utilisation increased for scaling PE. In fact, the synthesis report showed that the main factor contributing towards the increased LUT utilisation for scaling PE is caused by an increase in the number of LUTs used to implement logic such as comparators. Note that by design, the number of MACs implemented should not differ as long as the product $PE \cdot SIMD$ remains equal. However, note that in the current form of the implementation, each PE performs an activation in the form of threshold comparisons. Hence, by increasing PE, more logic is required to implement and execute these activations in parallel. That explains why scaling PE increases the LUT logic utilisation. Furthermore, increasing SIMD also has consequences on the partitioning of the threshold array, as well as implications on the stream width which might increase routing congestion.

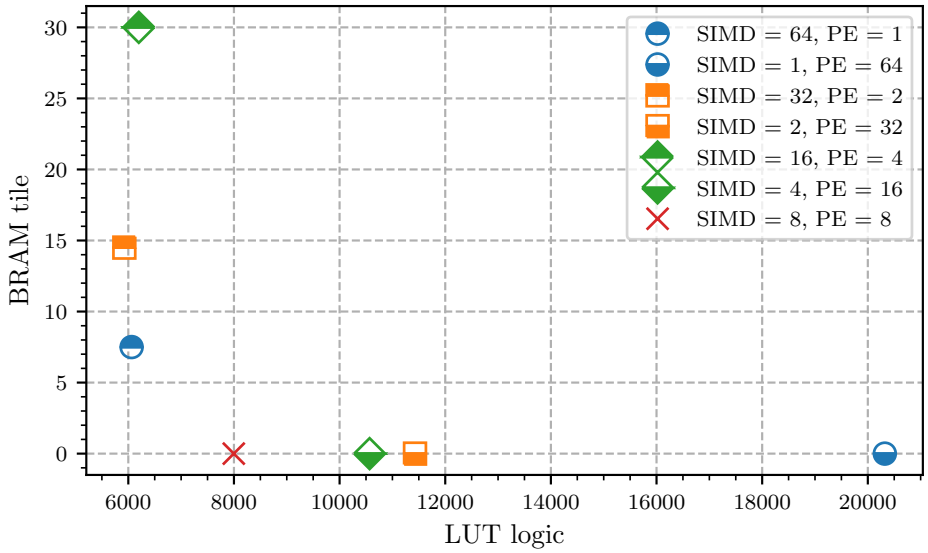


Figure 4.13: BRAM tile and LUT logic utilisation for MVAU with weight in LUTRAM with various folding factors achieving ≈ 500000 cycles latency. Results are obtained from RTL synthesis post-implementation and route reports.

Note that Figure 4.13 shows a Pareto-front, where the most optimal configuration is the closest to the origin. Based on this, we can draw three design rules:

1. Firstly, PE should be scaled large enough to ensure that the threshold array gets partitioned in such a way that the Vivado tool infers LUTs for its implementation. This is required as experiments have shown that BRAM utilisation is most likely to become a bottleneck in fitting the design on an FPGA. Another workaround to this problem is to decouple the threshold array by performing the threshold comparison functionality in a Thresholding_Batch layer following the MVAU.

2. Secondly, the number of PEs should be minimised as the required LUTs to implement the logic increases significantly.

3. Lastly, SIMD should be kept within reasonable bounds as it affects the width of the weight and input stream. Note that, depending on the hardware resources chosen to implement the weight, a user might also take into consideration to optimise the width of the weights to match an integer multiple of the width of the chosen hardware resource. This is especially required for more coarse-grained hardware memory types such as URAM.

4.5. FIFO analysis

A skip connection, as introduced in Section 2.5, is a simple mathematical operation; the outputs from two specific layers are added to each other produce. For example, tensor A, which is the output from the layer residing on the skip connection, gets added to tensor B, producing a new tensor $C = A + B$. In terms of hardware implementations, this might cause problems. Note that the data path along the skip connection is much smaller in terms of latency as the skip connection consists of a pointwise convolution and activation function only. In other words, it takes much fewer cycles to produce output values for tensor A compared to output values being produced for tensor B. Thus, from the perspective of the adder, the stream of input values from the operands A and B are highly unbalanced. Experiments showed that this causes a deadlock in the implementation, which could be resolved by inserting appropriately sized FIFOs at the output of the layer on the skip connection. The FINN compiler provides a transform that solves this problem. By performing RTL simulation of the design with very deep FIFOs inserted between all layers, the transform keeps track of the occupancy of each FIFO and resizes the final FIFO configuration accordingly. The downside of the current transformation is that it has a very long execution time for models that have a high end-to-end latency; for QuartzNet, it takes ≈ 9 days of runtime. However, this transformation is required to run only once for a specific folding configuration. On top of that, inserting FIFOs can balance out the entire design and can lead to a higher throughput in general.

The same design idea applies with specifying which resource type a FIFO should use; i.e. spreading the hardware resource utilisation as much as possible to stay within recommended limits posed by the Vivado tool in an attempt to ease timing closure and achieve potentially a higher frequency design. To achieve this, either LUTRAM, BRAM, or URAM is used accordingly. As the FIFOs are well understood, extensive tests are not performed.

4.6. Strategy for QuartzNet

Taking into account all of the aforementioned challenges and limitations, finding the right configuration with regards to folding is not a trivial task. During experiments with a highly folded QuartzNet configuration, it was found that inefficient memory utilisation resulted in a model that could not be placed on the target FPGA board due to a shortage of memory (BRAM and LUTRAM) resources. Therefore, the first objective is to ensure that the memory resources are utilised efficiently and that the utilisation of the various components is balanced. In order to achieve this, note that only two layers can be specified to utilise URAMs: SWU and MVAU. However, due to the limitation with regards to utilising URAMs to store the weights of MVAUs, only an SWU can utilise URAMs for the intermediate buffer. Therefore, URAMs might become underutilised. Note further that for the SWU, the URAMs do not need to be initialised and hence the IP block does not require an AXI-lite interface. As mentioned before, Vitis supports each Xilinx Object file (i.e. each IP) to have at most 1 outgoing AXI-lite interface. This limitation is thus not playing a role for the intermediate buffer of the SWU and therefore makes the SWU suited for utilising

URAMs. Therefore, to ensure that the memory resources are spread evenly, the intermediate buffer of all SWU layers will be implemented in URAM.

From experiments, it was found that the weights of the VVAU are implemented in BRAM. Thus, to spread the resource utilisation, the MVAUs will have the weights implemented in LUTRAM. This is not particularly ideal as the MVAU has relatively large weights which would be more suited to be placed in URAM.

In terms of computational resources, the goal is to spread the logic utilisation between the DSPs and LUTs evenly. Therefore, all of the multiplication operations for the VVAU and MVAU will be implemented in DSPs as DSPs are expected to be underutilised.

Furthermore, note that for simplicity, the threshold activation function is embedded within the MVAU and VVAU component, similar to the layer analysed in the previous sections.

From the analysis presented in Section 4.2, Section 4.3, and Section 4.4, it was found that the number of PEs for the VVAU and MVAU should be increased to at least 4, 8, or 16, depending on the size and complexity of the layer, in order to ensure that Vivado implements the thresholds in LUTROM. Note that the parallelism of the SWU must be the same as the parallelism of the VVAU, as described in Section 4.1. With this in mind, it is expected that it is feasible to obtain a baseline design with a critical latency of roughly 500000 cycles; i.e. the latency of the largest node in the model is roughly 500000 cycles.

5

Profiling & baseline comparison

This chapter will discuss the profiling of the quantized QuartzNet model on a high-end CPU and GPU device in Section 5.1. Subsequently, the proposed FPGA accelerator is analysed in terms of throughput, latency, and energy efficiency in Section 5.2. A comparison between the CPU, GPU, and FPGA based implementation of the quantized QuartzNet model is also presented.

5.1. Profiling QuartzNet baseline CPU & GPU

In order to provide a fair comparison, a high-performance CPU and GPU are considered to profile the quantized version of QuartzNet presented by the Xilinx Research Lab [16]. The CPU type of interest is the Intel Xeon CPU E5-2667 rated at 3.192 GHz with 6 cores and 112 GB of memory. The GPU type of interest is the Tesla V100 rated at 1.230 GHz with 16 GB of memory. Both hardware devices are reserved with Microsoft Azure. As mentioned before in Chapter 2, the reported accuracy of the original (floating-point) QuartzNet and quantized QuartzNet models on the LibriSpeech dev-other dataset are:

- QuartzNet NVIDIA [34]: 10.78% WER
- Quantized QuartzNet Xilinx Research Lab [16]: 12.00% WER

Table 5.1 shows for both the CPU and GPU implementation the execution time as well as the accuracy of evaluating the model on the LibriSpeech dev-other dataset. Note that there is a minor difference with respect to the previous presented speech-to-text pipeline in Section 2.5. The *pre-processing* step refers to extracting Mel-filterbank features from the raw audio waveform. However, the *QuartzNet model* step refers to the entire QuartzNet model as presented in Section 2.5, except for the final fully-connected layer. The *FC layer & decoding* step relates to the final fully connected layer followed by a softmax and logarithm operation. Finally, the *post-processing* step refers to the CTC decoding step that converts the predicted sequence of characters to a sensible sentence. Note that the pre-processing, decoding, and post-processing step are only a fraction of the total execution time. The main factor contributing to the execution time is, as expected, the bulk of the QuartzNet model.

First, notice that the GPU achieves roughly $25\times$ improvement in execution time over the CPU. Furthermore, note that the WER of both the CPU and GPU implementation of the quantized QuartzNet model are slightly larger than the original (floating point) QuartzNet model. This is an expected consequence of quantizing the model to lower precision weights and activations. Interesting to note is that the WER of the CPU and GPU implementation of the quantized QuartzNet model, as well as the reported WER of the quantized QuartzNet model, are all different. This is assumed to be caused by the unstable/non-deterministic sorting function in PyTorch. More precisely, when a list of numbers with duplicate entries gets sorted, the order of arrangement is non-deterministic. For quantized models, the output consists of a finite sequence of numbers and hence, it is more likely to end up with duplicate entries in the final prediction scores.

For the CPU implementation, power measurements could not be performed on an Azure virtual machine. However, a rough estimation can be given. Note that all cores were active in parallel during inference execution. The so-called thermal design power of the device, which represents the average

	CPU	GPU	CPU	GPU
Batch size	100		250	
Pre-processing	26.63 s	-	31.95 s	-
QuartzNet model	1496.03 s	-	1818.74 s	-
FC layer & Decoder	1.98 s	-	2.36 s	-
Post-processing	0.241 s	-	0.305 s	-
Total	1525.85 s	53.78 s	1855.14 s	55.83 s
WER	12.22 %	12.08 %	12.20%	12.09 %
Maximum power	130 W [1]	211 W	130 W [1]	242 W

Table 5.1: Accuracy and execution time of the quantized QuartzNet model performing inference on LibriSpeech dev-other dataset on both a CPU and GPU.

power dissipated when operating at base frequency with all cores active, is 130 W [1]. Thus, the power utilisation is assumed to be near 130 W. For the GPU implementation, the command line tool *nvidia-smi* was used to obtain power measurements by sampling the power consumption several times during the execution of the model.

Interesting to note is that dividing the dataset into larger batch sizes takes longer to process than a smaller batch size for both the CPU and GPU implementation.

5.2. QuartzNet FPGA

Due to the size of the QuartzNet model, the FPGA card of interest is the Alveo U250 as it is the largest one of the family of FPGAs suited for data centre workloads. Two implementations of an FPGA accelerator for the QuartzNet model are tested. For the first implementation, referred to as the *baseline* FPGA implementation, deep FIFOs are inserted between each of the layers to overcome the problems introduced in Section 4.5. The second implementation, also referred to as the *FIFO optimised* design, has been obtained by running the automatic FIFO sizing transformation from FINN. This transformation resizes the FIFOs according to their occupancy that has been tracked during RTL simulation. The downside of this transformation is that it takes roughly 9 days to complete for a model as large and relatively highly folded as the proposed QuartzNet implementation. The advantage compared to a baseline implementation with deep FIFOs between every layer is that a more area-efficient and higher-throughput implementation can be obtained.

Figure 5.1 visualises the entire speech-to-text pipeline in more detail. Note that grey boxes indicate steps that are executed on the CPU and the red boxes indicate steps that are executed on the FPGA. As noted before, the first convolution and activation are executed on a CPU to convert the data to an integer format to be streamed to the FPGA. To minimise the latency and increase the throughput of the entire speech-to-text pipeline, the first depthwise convolution and activation should also be mapped to an FPGA. However, this is expected to degrade the WER due to round-off errors caused by quantizing the input data. Besides, as the QuartzNet model contains 78 convolutional layers in total, the benefits in terms of throughput and latency of accelerating the first depthwise convolution and threshold layer are expected to be small as well. Especially considering the fact that deeper convolutional layers are larger in terms of computational effort. Thus, the proposed design executes the first convolutional layer and activation on a CPU, the bulk of the model is executed on the FPGA, and the final prediction and post-processing step are also executed on a CPU.

[1] Estimated quantity, has not been measured.

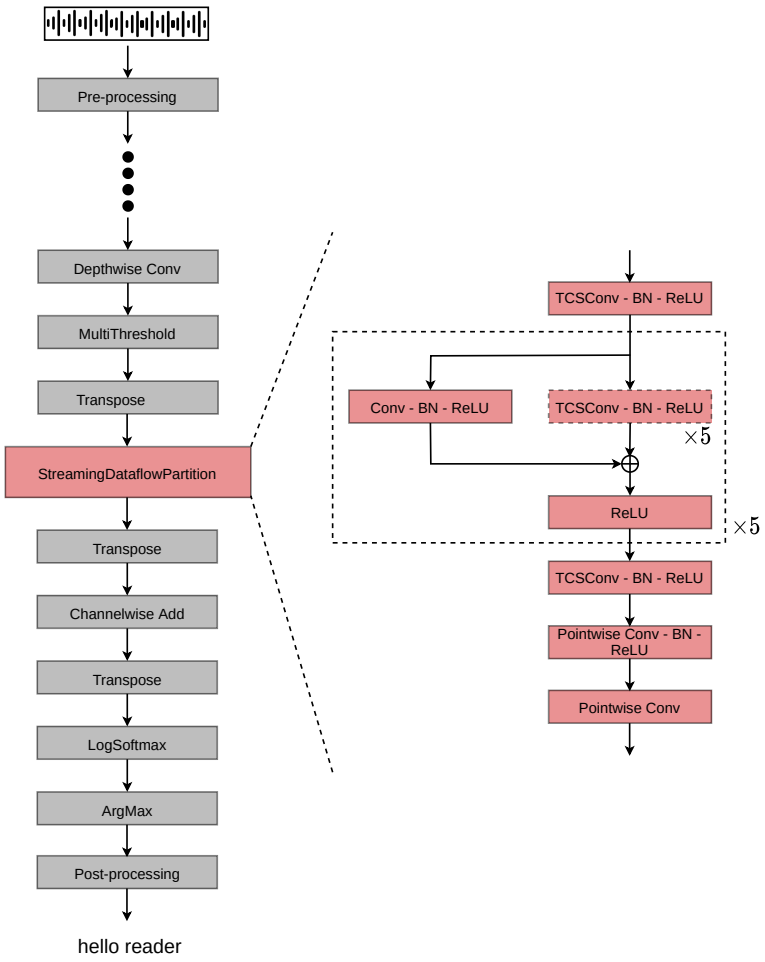
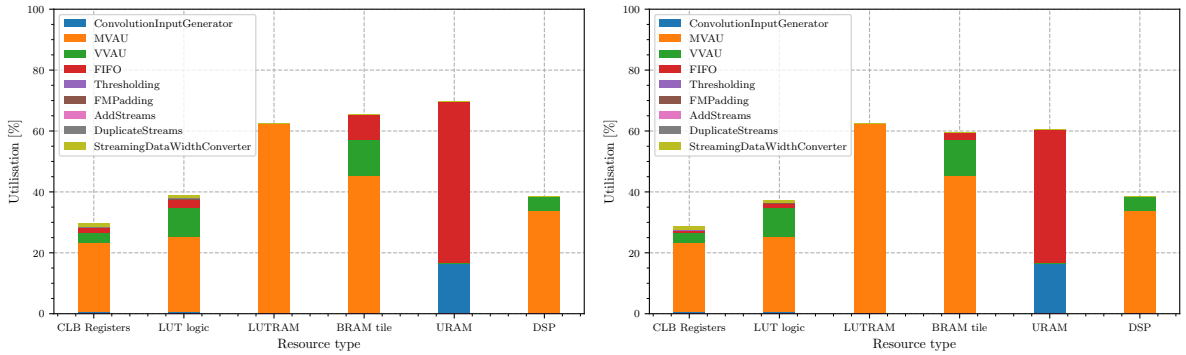


Figure 5.1: Overview of the accelerated quantized QuartzNet model.

5.2.1. Area utilisation for baseline and FIFO optimised design

Figure 5.2 visualises the resource utilisation for various hardware components categorised by the layer type. As noted before, the QuartzNet model consists of mainly depthwise and pointwise convolutions. The pointwise convolutions contain more weights and computations compared to the depthwise convolutions and as expected, the MVAU consumes the largest fraction of resources followed by the VVAU. Furthermore, note that the ConvolutionInputGenerator mainly utilises memory; in this case URAM as explained in Section 4.6. Comparing Figure 5.2a and Figure 5.2b, notice that the utilisation of LUTs, BRAMs and URAMs for the FIFO layer is also smaller. More precisely, the FIFO optimised implementation has a 5.8% and 9.0% reduction in total BRAM and URAM utilisation respectively. Note further that the recommended maximum resource utilisation by the Vivado tool is 50% for registers, 70% for LUTs, 25% for LUTRAM, and 80% for BRAM, URAM and DSP blocks. Both of the obtained FPGA implementations stay within these limits, with the only exception of LUTRAM utilisation peaking at roughly 62.6%. LUTRAMs are primarily used for storing the weights of the MVAU component, which are relatively large (in the order of $0.2 - 0.5$ MB). As can be seen from Figure 5.2b, the memory utilisation of LUTRAM, BRAM and URAM is spread evenly around 60%. Thus, in order to lower the memory utilisation, it is worth investigating to implement some of the weights of the MVAU in URAM and some of the intermediate buffers of the SWU and FIFOs in LUTRAM. Note that the intermediate buffer of the SWU ranges from $0.03 - 0.15$ MB, while some FIFOs are as small as 0.02 MB. However, as mentioned in Section 4.1, the weights of the MVAU could not be implemented in URAM due to an address range problem and hence, this investigation is left for future work.

The Alveo U250 device utilisation for the quantized QuartzNet model with optimised FIFOs can be found in Figure C.1 in Appendix C.



(a) Resource utilisation for the baseline FPGA implementation.

(b) Resource utilisation for the FIFO optimised FPGA implementation.

Figure 5.2: Resource utilisation for two different FPGA implementations of the quantized QuartzNet model.

5.2.2. Comparison between CPU, GPU, and FPGA implementation

To provide a fair comparison between the CPU, GPU, and FPGA-based implementation of the QuartzNet model, the same exact part of the QuartzNet model should be implemented on all three hardware platforms. The FPGA implementation is taken as a point of reference here. As shown in Figure 5.1, this implementation encapsulates the entire QuartzNet model except for the first depthwise convolution and activation and the final decoder and post-processing block. To simplify the experiments, the CPU and GPU implementations of the QuartzNet model encapsulate all of the residual blocks similar to the FPGA implementation, except that the CPU/GPU implementations also account for the first depthwise convolution and activation. This means that for the results presented below, the CPU and GPU implementation are slightly disadvantaged due to the fact that the tested model contains two more layers. However, as reasoned above, this is deemed acceptable due to the negligible size of the first two layers compared to the bulk of the QuartzNet model. Hence, the impact on the results and the conclusions derived from the results is deemed negligible.

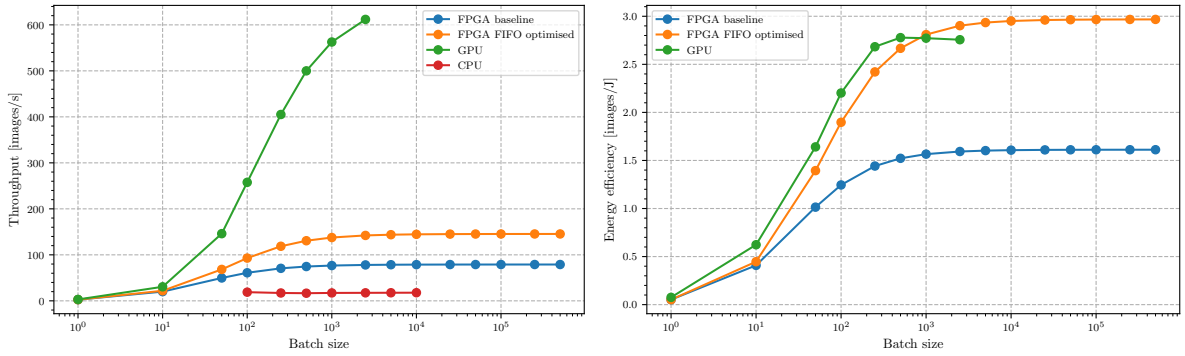
The experiments below are performed in the following way. A random tensor of the appropriate size and format is constructed; $[B, 64, 256]$ for the CPU/GPU implementation in floating point format, and $[B, 64, 128]$ for the FPGA in integer format. B refers to the batch size, which is increased until the point where the system would run out of memory. The idea is to isolate the inference pass of the model as much as possible and neglect other types of overhead such as memory transfers. For this reason, a single batch size is chosen and fed to the model, instead of processing a larger dataset in several chunks. For the CPU and GPU implementation, several runs are performed, which are then averaged to obtain the final runtime. During experiments, it was noticed that the first batch processed by the GPU had a large overhead in runtime which is assumed to be due to the fact that the GPU is in sleep-mode initially. For the FPGA, a single run was performed for each batch size up until out-of-memory bound would be reached.

Figure 5.3a visualises the throughput in images per second at various batch sizes for a CPU, GPU, and two FPGAs implementations. As expected, the CPU implementation achieves the lowest throughput. Furthermore, note that the FIFO optimised FPGA implementation of the QuartzNet model is able to achieve almost a $2\times$ improvement in throughput compared to the baseline FPGA implementation. Comparing the two FPGA implementations, it was observed that the number of FIFOs is halved in the FIFO optimised implementation compared to the baseline FPGA implementation. Secondly, several FIFOs in the FIFO optimised design were also $2 - 4\times$ larger compared to the sizes of the FIFOs in the baseline implementation. Thus, a possible explanation for the low throughput in the baseline design could be that the FIFOs are still too shallow to prevent stalls due to layer outputs being generated in a bursty instead of continuous way.

Lastly, note that the GPU implementation is able to achieve the highest throughput. This is partly expected, as GPUs are suited for achieving high throughput at highly parallelisable tasks. As explained in Section 2.3, FPGA implementations could potentially be favoured due to less power being consumed.

Figure 5.3b visualises the throughput in images per second normalised over the maximum power consumption at various batch sizes for a GPU and two FPGA implementations. In other words, the energy efficiency in images per Joule is visualised. Note that the gap between the FPGA implement-

ation and GPU implementation got considerably smaller. In fact, the FPGA implementation is able to achieve the highest achievable energy efficiency.



(a) Comparison of throughput in images per second at various batch sizes for a CPU, GPU, and FPGA implementation.

(b) Comparison of energy efficiency in images per Joule at various batch sizes for a GPU and FPGA implementation.

Figure 5.3: Throughput and energy efficiency of a CPU, GPU, and FPGA-based quantized QuartzNet implementation.

Table 5.2 summarises various metrics for the CPU, GPU, and best performing FPGA implementation. With the GPU implementation, the lowest latency and highest throughput are achieved. In terms of energy efficiency, the FPGA implementation is better than the GPU implementation. In terms of WER, the best obtained WER for the GPU model outperformed the best obtained WER for the CPU model. As explained before, this is assumed to be caused due to the non-deterministic sorting in PyTorch. With regards to the FPGA model, note that the WER on the LibriSpeech dev-other dataset has not been measured as the experiment setup has not been constructed. The WER of the FPGA-based model is assumed to be the same as the CPU and GPU-based models, because the FPGA-based model is by design functionally equivalent to the Brevitas model run on a CPU and GPU device.

	CPU	**GPU**	**FPGA**
Frequency	3.2 GHz	1.230 GHz	0.108 GHz
Highest achievable throughput [images/s]	18.86	611.86	145.40
Latency [ms]	56.42	1.54	6.88
Maximum power [W]	130 [1]	222	49
Highest achievable energy efficiency [images/J]	0.15 [1]	2.78	2.97
WER LibriSpeech dev-other	12.20%	12.08%	≈12%[1]

Table 5.2: Comparison between CPU, GPU and FPGA implementation in terms of throughput, latency, maximum power, and energy efficiency.

5.2.3. Comparison to other FINN-generated accelerators

Finally, to put this work in line with other accelerators developed by FINN, Table 5.3 compares various metrics among the MobileNetV1, ResNet-50, and QuartzNet models. In terms of novelty, the FPGA accelerator for QuartzNet is the first model generated with the FINN compiler that targets the speech-to-text recognition domain. Inherently coupled with the novel domain, the QuartzNet model is based on layers operating on 1D tensors, while the other models are based on layers operating on 2D tensors.

Furthermore, this work also showcases the largest FPGA implementation achieved so far with the FINN framework as well as the largest CNN for speech-to-text recognition implemented on an FPGA.

Note that in terms of model size, the ResNet-50 model is slightly larger than the QuartzNet model, while the QuartzNet model contains roughly three times more FINN nodes. This is in accordance with one of the motivations behind the design of the QuartzNet model. As explained in Section 2.5, the use of depthwise separable convolutions significantly lowers the number of parameters used. The ResNet-50 model is composed of regular convolutions, as opposed to the QuartzNet model, which

[1] Estimated quantity, has not been measured.

consists of solely depthwise separable (and pointwise) convolutions. For that reason, the architecture of the QuartzNet model is deep, while still containing a relatively low amount of learnable parameters.

Furthermore, note that MobileNetV1 is the smallest in terms of model size and FINN nodes. Interesting to note is that MobileNetV1 has, similar to QuartzNet, depthwise separable convolutions.

	MobileNetV1 [4]	**ResNet-50** [4]	**QuartzNet (this work)**
Application	Image classification	Image classification	Speech-to-text recognition
Precision	W4A4	W1A2	Inner layers: W4A4 Outer layers: W8A8
Model Size [MB]	2.1	11.25	9.95
GOps	1.1	6.8	4.8
FINN nodes	115	277	871

Table 5.3: Comparison of proposed accelerator with other accelerators generated with FINN. Note that the model size refers to the size in bytes of the weights and thresholds of the model.

6

Conclusion & future work

As deep neural networks are becoming increasingly large, alternative hardware platforms have gained attention to create more power-efficient, high-throughput, and low-latency accelerators. In this work, we have studied three research questions and addressed them as follows:

1. We have investigated the applicability of generating efficient CNN inference accelerators for speech-to-text applications on an FPGA by means of a dataflow-style compiler named FINN.
2. We profiled and analysed the throughput, latency, and power efficiency characteristics of the generated FPGA accelerator.
3. We compared the generated FPGA accelerator against mainstream CPU and GPU-based implementations of the CNN inference model.

6.1. Conclusion

The project has been divided into two parts. In the first part, I have extended the FINN compiler to support the mapping of non-square and 1D CNNs to dataflow-style FPGA realisations. These changes can be found in the open-source repositories of FINN, FINN-base, and FINN-hlslib [36, 37, 40]. In the second part, I have used and tailored the transformations in the FINN compiler to create an FPGA accelerator for a quantized version of the well-known QuartzNet model. This showcases the largest CNN topology for speech-to-text inference implemented on an FPGA.

Regarding the first research question, this work has shown that an energy-efficient, high-throughput, and low-latency FPGA accelerator can be created by means of the FINN compiler. An example has been demonstrated on a quantized version of a sub state-of-the-art CNN for speech-to-text recognition named QuartzNet. The network poses an implementation challenge due to its size in number of layers; creating an FPGA accelerator by hand in RTL is arguably even infeasible to do for such a large network. This work shows that, when the infrastructure in FINN has been established to support the layers in the network, creating an efficient FPGA accelerator can be done within a fraction of the implementation effort compared to a handwritten RTL design.

Regarding the second and third research questions, the highest throughput for the QuartzNet FPGA accelerator that could be achieved was 145.4 images per second with a latency of 6.88 ms. The highest achievable energy efficiency is 2.97 images per Joule. Compared to a high-end CPU implementation, the proposed FPGA accelerator achieves 7.7× higher peak throughput and 8.2× lower latency. Compared to a high-end GPU implementation, the proposed FPGA accelerator improves the achievable energy efficiency by 6.8% at the expense of lower throughput and higher latency.

6.2. Future work

In order to extend the FINN compiler to allow a wider range of hardware implementations to be realised and to lower the design burden, or to improve the proof-of-concept FPGA implementation of the quantized QuartzNet model, the following list of items are considered worth investigating:

1. Automate the streamlining procedure. Currently, this procedure in FINN has to be done by hand for each specific layer. For new users, this requires them to get acquainted with the streamlining transformations in FINN and possibly write custom transformations suited for their own network. Automating the streamlining procedure would lower the design effort to create an FPGA accelerator for new networks. However, this would imply that all the required transformations to streamline any particular graph are present in FINN. This is often not the case, which makes this problem hard to solve. A simpler solution is to consider automating the streamlining procedure for certain graphs that contain a sub-set of available ONNX nodes for which the graph transformations are well established; such as graphs consisting only out of batch normalisation layers, multiplications and additions, convolutions, and activations, similar to QuartzNet.

2. Automate the folding procedure. As shown in Chapter 4, the folding procedure is not straightforward and requires several iterations to obtain a feasible hardware configuration. The difficulty with the folding procedure is that it takes a significant amount of time to get reliable resource estimations; the most reliable estimations are from post-synthesis and implementation. This makes it infeasible to perform a thorough design space exploration for a complete model. What makes FINN powerful is that each layer has specific estimation functions that estimate the relevant compute and memory utilisation based on the configuration of the layer by means of a (simple) Python function. These estimation functions are essentially constructed by means of regression analysis on a large set of resource utilisation numbers from HLS synthesis, or are created by means of a rule-based system describing how Vivado HLS will most likely infer memory resources. Due to the fact that the non-square and 1D SWU implementations are (relatively) new, these estimation functions have not been constructed. Thus, in order to utilise FINN's resource estimation functions to allow exploring the design space of non-square and 1D models, the relevant estimation models for the SWU need to be extended in a similar way to how the estimation models are obtained for the older established FINN custom operators.

3. Optimise the folding of the quantized QuartzNet model. In order to improve the throughput and latency by $2\times$, all layers should be targeted to be folded by $2\times$. As shown in Chapter 5, the resource utilisation of the generated FPGA accelerator is within the limits proposed by the Vivado tool. It is worth exploring whether the layers can be folded by an additional factor. It is expected that this will lower the gap between the GPU and FPGA based implementation in terms of achievable throughput and latency. One possible downside is that the attainable frequency could suffer due to the very high utilisation of resources.

4. Quantize the input data to integer format as explained in Section 3.1.1. This would allow the first convolutional and activation layer to be implemented on an FPGA. This will improve the entire end-to-end throughput and latency of performing an inference task with an FPGA accelerator for the QuartzNet model. Note that the WER is expected to decrease when quantizing the input data to integer format.

5. Apply the floorplanning transformation proposed by Alonso et al. with QuartzNet [4]. This is expected to increase the attainable frequency of the design and therefore improve the throughput and latency. From the current quantized QuartzNet design, reset and clock signals are causing a worst negative slack of ≈ 2.24 ns on a target path of 7 ns. If the design can be implemented more efficiently and thereby attain the target clock period, an increase of approximately 35 MHz is expected. It is also interesting to explore whether the target clock period can be lowered further.

6. Store the weights of the MVAU component in URAM. For large weights stored in URAM in Alveo devices, there is at the time of writing a bug in the address range that is allocated for these weights. In order to increase the design space for the QuartzNet model, this problem should be alleviated as it would allow for implementing the weights of the MVAU component in URAM. In turn, exploring different hardware implementations of the quantized QuartzNet model could possibly lead to an implementation with more efficient memory utilisation.

7. Decouple the thresholding operation from the VVAU/MVAU to have more control over which hardware resources are utilised to store the thresholds. This would increase the number of nodes in the network, but would aid the design space exploration as the user has more control over which resources get instantiated. As shown in Chapter 4, the Vivado tool does not always infer the most efficient hardware memory resource for the thresholds.

8. Implement a custom FINN-ONNX MatMul operator for depthwise convolutions. In order to execute the depthwise convolutions more efficiently, it is worth investigating to create a custom FINN-ONNX MatMul operator that performs a dense matrix multiplication as shown in Figure 2.12. In the Python domain, depthwise convolutions are currently lowered to an Im2Col operator followed by a sparse matrix multiplication. For normal-sized models, this has shown to be a successful approach. However, for a model with a large number of depthwise convolutions, such as QuartzNet, this poses two particular problems. Firstly, storing the model exceeds the maximum size of an ONNX format file and secondly, executing the model by means of ONNX-runtime leads to large execution time due to large (sparse) matrix multiplications.

9. Enable FINN accelerators to be used in big data pipelines. Alternative hardware platforms, such as FPGAs, are becoming increasingly useful in big data applications as these devices can outperform CPUs in terms of throughput, latency, and power efficiency. However, programming and integrating FPGAs within big data pipelines is considered hard [25]. In order to aid in bridging the domain of machine learning inference, big data applications, and FPGA devices, integrating Fletcher [54, 55] within the FINN framework is considered to be fruitful. Fletcher is an open-source framework that can generate high-performance hardware interfaces to data in Apache Arrow format. Apache Arrow is a standardised in-memory data format for big data applications that can standardise data movement on hardware.

A

Memory analysis

A.1. Convolution Input Generator

Layer configuration			Resource utilisation						
RAM style	SIMD		LUT Logic	LUT Memory	FF	BRAM tile	URAM	DSP	Latency
LUTRAM	1	V1	3261	11392	545	0	0	0	3433472
		V2	1603	3317	848	0	0	0	3367886
	2	V1	3971	7120	589	0	0	0	1716736
		V2	1401	2093	805	0	0	0	1683918
	4	V1	2138	7120	656	0	0	0	858368
		V2	1309	2101	923	0	0	0	841934
	8	V1	1907	7120	578	0	0	0	429184
		V2	1298	2117	977	0	0	0	420942
	16	V1	1863	6586	593	0	0	0	214592
		V2	1342	2033	1236	0	0	0	210446
	32	V1	2191	6660	876	0	0	0	107296
		V2	1589	2097	1793	0	0	0	105198
	64	V1	2297	6808	1509	0	0	0	53648
		V2	1794	2372	2848	0	0	0	52574
BRAM	1	V1	136	0	216	12	0	0	3433472
		V2	691	53	725	4	0	0	3367886
	2	V1	142	0	231	12	0	0	1716736
		V2	706	53	749	4	0	0	1683918
	4	V1	140	0	261	12	0	0	858368
		V2	727	61	812	4	0	0	841934
	8	V1	156	0	324	11.5	0	0	429184
		V2	761	77	930	4	0	0	420942
	16	V1	198	0	451	11.5	0	0	214592
		V2	805	109	1159	4	0	0	210446
	32	V1	299	0	706	11.5	0	0	107296
		V2	999	173	1611	4	0	0	105198
	64	V1	486	0	1217	14.5	0	0	53648
		V2	1272	300	2510	4	0	0	52574
URAM	1	V1	198	0	220	0	23	0	3433472
		V2	741	46	723	0	7	0	3367886
	2	V1	155	0	234	0	12	0	1716736
		V2	751	47	747	0	4	0	1683918
	4	V1	154	0	264	0	6	0	858368
		V2	771	48	802	0	2	0	841934
	8	V1	166	0	326	0	3	0	429184
		V2	787	57	904	0	1	0	420942
	16	V1	202	0	452	0	2	0	214592
		V2	834	74	1102	0	1	0	210446
	32	V1	289	0	706	0	2	0	107296
		V2	970	107	1491	0	2	0	105198
	64	V1	481	0	1217	0	4	0	53648
		V2	1170	171	2263	0	4	0	52574

Table A.1: Post-synthesis and implementation hardware resource analysis for two implementations of a 1D SWU.

A.2. Vector Vector Activate Unit

Layer configuration		Resource utilisation						
PE	Resource type	LUT		FF	Bram tile	URAM	DSP	Latency
		Logic	Memory					
2	LUT	-	-	-	-	-	-	-
	DSP	-	-	-	-	-	-	-
4	LUT	907	0	351	29.5	0	0	835584
	DSP	735	0	297	29.5	0	3	835584
8	LUT	2554	0	1571	4	0	0	417792
	DSP	2160	0	1398	4	0	7	417792
16	LUT	3980	0	2942	8	0	0	208896
	DSP	3170	0	2547	8	0	15	208896
32	LUT	7464	0	4816	16	0	0	104448
	DSP	6083	0	4084	16	0	31	104448
64	LUT	13388	0	6519	32	0	0	52224
	DSP	10098	0	5151	32	0	63	52224

Table A.2: Post-synthesis and implementation hardware resource analysis for a VVAU.

A.3. Matrix Vector Activate Unit

A.3.1. Weight memory - LUTRAM

Layer configuration					Resource utilisation						
RAM style	SIMD	PE	Memory mode	Resource type	LUT Logic	LUT Memory	FF	BRAM tile	URAM	DSP	Latency
LUTRAM	2	1	Decoupled	LUT	5345	17000	16016	7.5	0	0	16777216
				DSP	5347	17000	16009	7.5	0	3	16777216
	4	1	Decoupled	LUT	5736	16996	16021	7.5	0	0	8388608
				DSP	5601	16996	15970	7.5	0	4	8388608
	8	1	Decoupled	LUT	5810	17000	16149	7.5	0	0	4194304
				DSP	5597	17000	16058	7.5	0	10	4194304
	16	1	Decoupled	LUT	5761	18426	18756	7.5	0	0	2097152
				DSP	5401	18426	18639	7.5	0	17	2097152
	1	2	Decoupled	LUT	5340	19228	16045	14.5	0	0	16777216
				DSP	5429	19228	16008	14.5	0	1	16777216
	1	4	Decoupled	LUT	5903	19288	16141	30	0	0	8388608
				DSP	5752	19288	16028	30	0	3	8388608
	1	8	Decoupled	LUT	7884	19520	17736	0	0	0	4194304
				DSP	7469	19520	17474	0	0	7	4194304
	1	16	Decoupled	LUT	10770	18704	19512	0	0	0	2097152
				DSP	9966	18704	18938	0	0	15	2097152
	8	2	Decoupled	LUT	6302	16004	16385	14.5	0	0	2097152
				DSP	5876	16004	16252	14.5	0	20	2097152
	8	4	Decoupled	LUT	6936	16132	16635	30	0	0	1048576
				DSP	6077	16132	16374	30	0	40	1048576
	8	8	Decoupled	LUT	9715	16136	18679	0	0	0	524288
				DSP	7997	16136	18195	0	0	80	524288
	16	8	Decoupled	LUT	11873	18360	22046	0	0	0	262144
				DSP	8798	18360	21669	0	0	136	262144
	8	16	Decoupled	LUT	14075	16058	21269	0	0	0	262144
				DSP	10655	16058	20395	0	0	160	262144

Table A.3: Post-synthesis and implementation hardware resource analysis for an MVAU with weights in LUTRAM.

Layer configuration					Resource utilisation						
RAM style	SIMD	PE	Memory mode	Resource type	LUT Logic	LUT Memory	FF	BRAM tile	URAM	DSP	Latency
LUTRAM	64	1	Decoupled	LUT	7647	18944	20203	7.5	0	0	524288
				DSP	6063	18944	19756	7.5	0	68	524288
	1	64	Decoupled	LUT	22147	18688	28591	0	0	0	524288
				DSP	20322	18688	27701	0	0	63	524288
	32	2	Decoupled	LUT	7543	18944	19766	14.5	0	0	524288
				DSP	5925	18944	19808	14.5	0	80	524288
	2	32	Decoupled	LUT	16073	17632	23298	0	0	0	524288
				DSP	11428	17632	23140	0	0	96	524288
	16	4	Decoupled	LUT	7703	18536	19646	30	0	0	524288
				DSP	6199	18536	19382	30	0	68	524288
	4	16	Decoupled	LUT	11889	16624	20000	0	0	0	524288
				DSP	10568	16624	19353	0	0	64	524288
	8	8	Decoupled	LUT	9715	16136	18679	0	0	0	524288
				DSP	7997	16136	18195	0	0	80	524288

Table A.4: Post-synthesis and implementation hardware resource analysis for an MVAU with weights in LUTRAM for a target of ≈ 500000 cycles latency.

A.3.2. Weight memory - BRAM

Layer configuration					Resource utilisation						
RAM style	SIMD	PE	Memory mode	Resource type	LUT Logic	LUT Memory	FF	BRAM tile	URAM	DSP	Latency
BRAM	2	1	Decoupled	LUT	1209	0	2325	39.5	0	0	16777216
				DSP	1210	0	2318	39.5	0	3	16777216
	4	1	Decoupled	LUT	1097	0	2328	37.5	0	0	8388608
				DSP	971	0	2255	37.5	0	4	8388608
	8	1	Decoupled	LUT	1207	0	2433	36.5	0	0	4194304
				DSP	1006	0	2342	36.5	0	10	4194304
	16	1	Decoupled	LUT	1455	0	2639	36	0	0	2097152
				DSP	1100	0	2522	36	0	17	2097152
	1	2	Decoupled	LUT	1482	0	2322	46.5	0	0	16777216
				DSP	1632	0	2338	46.5	0	1	16777216
	1	4	Decoupled	LUT	2019	0	2378	60	0	0	8388608
				DSP	1863	0	2333	60	0	3	8388608
	1	8	Decoupled	LUT	4000	0	3898	29	0	0	4194304
				DSP	3611	0	3797	29	0	7	4194304
	1	16	Decoupled	LUT	6930	0	5528	28.5	0	0	2097152
				DSP	6166	0	5250	28.5	0	15	2097152
	8	2	Decoupled	LUT	1643	0	2607	43	0	0	2097152
				DSP	1227	0	2456	43	0	20	2097152
	8	4	Decoupled	LUT	2342	0	2823	58.5	0	0	1048576
				DSP	1481	0	2562	58.5	0	40	1048576
	8	8	Decoupled	LUT	5136	0	4745	28.5	0	0	524288
				DSP	3426	0	4261	28.5	0	80	524288
	16	8	Decoupled	LUT	7119	0	5598	28.5	0	0	262144
				DSP	4192	0	5079	28.5	0	136	262144
	8	16	Decoupled	LUT	9426	0	7081	28.5	0	0	262144
				DSP	6046	0	6207	28.5	0	160	262144

Table A.5: Post-synthesis and implementation hardware resource analysis for an MVAU with weights in BRAM.

Layer configuration					Resource utilisation						
RAM style	SIMD	PE	Memory mode	Resource type	LUT Logic	LUT Memory	FF	BRAM tile	URAM	DSP	Latency
BRAM	64	1	Decoupled	LUT	3294	0	3473	36	0	0	524288
				DSP	1713	0	3024	36	0	68	524288
	1	64	Decoupled	LUT	18324	0	14698	28.5	0	0	524288
				DSP	16489	0	13774	28.5	0	63	524288
	32	2	Decoupled	LUT	3166	0	3039	43	0	0	524288
				DSP	1573	0	3099	43	0	80	524288
	2	32	Decoupled	LUT	11933	0	9312	28.5	0	0	524288
				DSP	7317	0	9210	28.5	0	96	524288
	16	4	Decoupled	LUT	3308	0	3300	58.5	0	0	524288
				DSP	1848	0	3027	58.5	0	68	524288
	4	16	Decoupled	LUT	7556	0	6309	28.5	0	0	524288
				DSP	6074	0	5421	28.5	0	64	524288
	8	8	Decoupled	LUT	5136	0	4745	28.5	0	0	524288
				DSP	3426	0	4261	28.5	0	80	524288

Table A.6: Post-synthesis and implementation hardware resource analysis for an MVAU with weights in BRAM for a target of ≈ 500000 cycles latency.

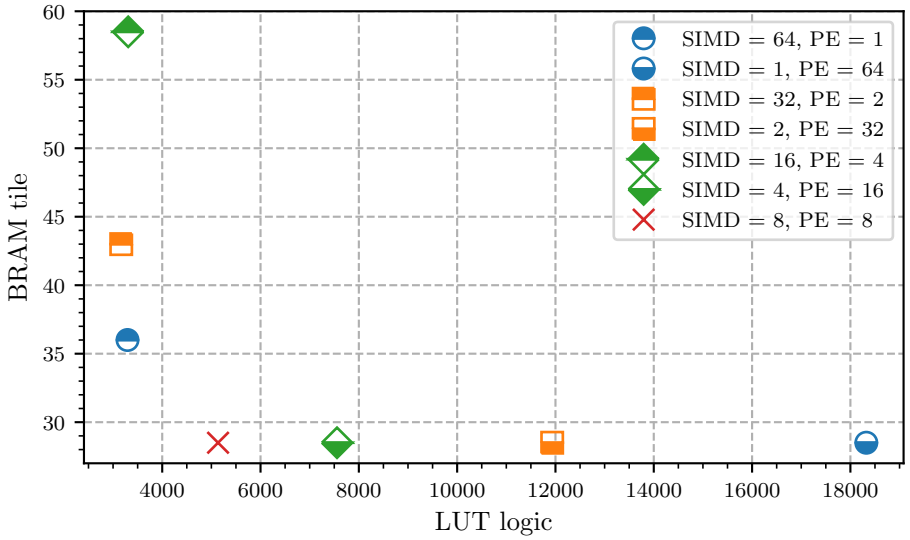


Figure A.1: BRAM tile and LUT logic utilisation for MVAU with weights in BRAM with various folding factors achieving ≈ 500000 cycles latency. Results are obtained from RTL synthesis post implementation and route reports.

A.3.3. Weight memory - URAM

Layer configuration					Resource utilisation						
RAM style	SIMD	PE	Memory mode	Resource type	LUT Logic	LUT Memory	FF	BRAM tile	URAM	DSP	Latency
URAM	2	1	Decoupled	LUT	1228	0	2428	7.5	32	0	16777216
				DSP	1227	0	2421	7.5	32	3	16777216
	4	1	Decoupled	LUT	1128	0	2400	7.5	16	0	8388608
				DSP	998	0	2327	7.5	16	4	8388608
	8	1	Decoupled	LUT	1222	0	2538	7.5	8	0	4194304
				DSP	1066	0	2458	7.5	8	10	4194304
	16	1	Decoupled	LUT	1463	0	2789	7.5	4	0	2097152
				DSP	1111	0	2672	7.5	4	17	2097152
	1	2	Decoupled	LUT	1502	0	2425	14.5	32	0	16777216
				DSP	1649	0	2441	14.5	32	1	16777216
	1	4	Decoupled	LUT	2045	0	2510	30	16	0	8388608
				DSP	1890	0	2465	30	16	3	8388608
	1	8	Decoupled	LUT	4015	0	4003	0	8	0	4194304
				DSP	3633	0	3884	0	8	7	4194304
	1	16	Decoupled	LUT	6940	0	5972	0	4	0	2097152
				DSP	6171	0	5382	0	4	15	2097152
	8	2	Decoupled	LUT	1657	0	2757	14.5	4	0	2097152
				DSP	1237	0	2615	14.5	4	20	2097152
	8	4	Decoupled	LUT	2421	0	3138	30	4	0	1048576
				DSP	1560	0	2877	30	4	40	1048576
	8	8	Decoupled	LUT	5256	0	5060	0	4	0	524288
				DSP	3547	0	4576	0	4	80	524288
	16	8	Decoupled	LUT	7297	0	6174	0	8	0	262144
				DSP	4399	0	5649	0	8	136	262144
	8	16	Decoupled	LUT	9616	0	7648	0	8	0	262144
				DSP	6235	0	6778	0	8	160	262144

Table A.7: Post-synthesis and implementation hardware resource analysis for an MVAU with weights in URAM.

Layer configuration					Resource utilisation						
RAM style	SIMD	PE	Memory mode	Resource type	LUT Logic	LUT Memory	FF	BRAM tile	URAM	DSP	Latency
URAM	64	1	Decoupled	LUT	3419	0	3788	7.5	4	0	524288
				DSP	1827	0	3339	7.5	4	68	524288
	1	64	Decoupled	LUT	18437	0	15323	0	4	0	524288
				DSP	16613	0	14093	0	4	63	524288
	32	2	Decoupled	LUT	3303	0	3354	14.5	4	0	524288
				DSP	1698	0	3396	14.5	4	80	524288
	2	32	Decoupled	LUT	10910	0	9987	0	4	0	524288
				DSP	7448	0	9525	0	4	96	524288
	16	4	Decoupled	LUT	3410	0	3615	30	4	0	524288
				DSP	1971	0	3342	30	4	68	524288
	4	16	Decoupled	LUT	7681	0	6624	0	4	0	524288
				DSP	6197	0	5736	0	4	64	524288
	8	8	Decoupled	LUT	5256	0	5060	0	4	0	524288
				DSP	3547	0	4576	0	4	80	524288

Table A.8: Post-synthesis and implementation hardware resource analysis for an MVAU with weights in URAM for a target of ≈ 500000 cycles latency.

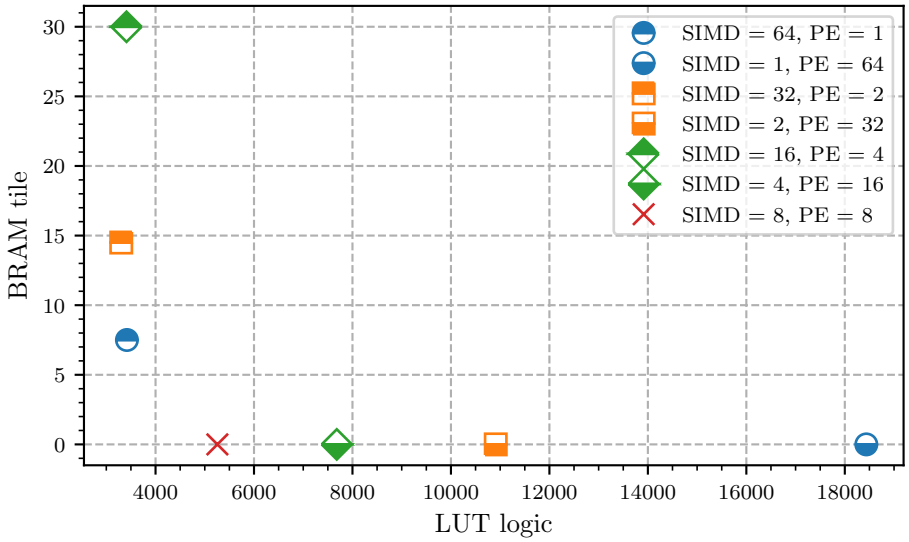


Figure A.2: BRAM tile and LUT logic utilisation for MVAU with weights in URAM with various folding factors achieving ≈ 500000 cycles latency. Results are obtained from RTL synthesis post implementation and route reports.

B

HLS implementation of custom Sliding Window Unit

```
/**
 * \brief Sliding Window unit that produces output vectors for feeding
 * a Matrix_Vector_Activate_Batch, implementing the im2col algorithm.
 * To be used when kernel is not square
 *
 * \tparam ConvKernelDim_x      Dimension of the convolutional kernel - x axis
 * \tparam IFMChannels          Number of Input Feature Maps
 * \tparam Input_precision      Number bits per pixel
 * \tparam IFMDim_x             Width of the Input Feature Map
 * \tparam OFMDim_x             Width of the Output Feature Map
 * \tparam SIMD                 Number of input columns computed in parallel
 * \tparam R                    Datatype for the resource used for FPGA implementation
 *                                of the SWG - safely deducible from the parameters
 *
 * \param in                    Input stream
 * \param out                   Output stream
 * \param numReps               Number of time the function has to be repeatedly executed
 *                              (e.g. number of images)
 * \param r                     Resource type for the hardware implementation of the memory
 *                              block
 */
template<unsigned int ConvKernelDim_x,
                 unsigned int IFMChannels,
                 unsigned int Input_precision,
                 unsigned int IFMDim_x,
                 unsigned int OFMDim_x,
                 unsigned int SIMD,
                 typename R>
void ConvolutionInputGenerator_1D_dws_lowbuffer(
                stream<ap_uint<SIMD*Input_precision> > & in,
                stream<ap_uint<SIMD*Input_precision> > & out,
                const unsigned int numReps,
                R const &r) {
        CASSERT_DATAFLOW(IFMChannels % SIMD == 0);
        const unsigned int multiplying_factor = IFMChannels/SIMD;
        const unsigned int buffer_size = ConvKernelDim_x * multiplying_factor;
        ap_uint<SIMD*Input_precision> inputBuf[buffer_size];
#pragma HLS ARRAY_PARTITION variable=inputBuf complete dim=1
        memory_resource(inputBuf, r);
        const unsigned int cycles_read_block = multiplying_factor*(ConvKernelDim_x-1)-
        (ConvKernelDim_x-1);
        const unsigned int baseIter = cycles_read_block + (OFMDim_x * buffer_size);
        unsigned int inp = 0, index_write=0, index_read = 0, j = 0, internal_counter = 0;
#pragma HLS reset variable=inp
        for (unsigned int count_image = 0; count_image < numReps; count_image++) {
                for (unsigned int i = 0; i < baseIter; i++) {
#pragma HLS PIPELINE II=1
```

```
                        if (inp < cycles_read_block) {// Initial buffer of ConvKernelDim lines
                                ap_uint<SIMD*Input_precision> inElem;
                                inElem = in.read();
                                inputBuf[inp] = inElem;
                                inp++;
                        }
                        else { // Read & write & update
                                // Read input buffer
                                if (inp < IFMDim_x*multiplying_factor){
                                        if (inp < buffer_size ||
                                    internal_counter >= buffer_size-multiplying_factor){
                                                ap_uint<SIMD*Input_precision> inElem;
                                                inElem = in.read();
                                                index_write = inp % (buffer_size);
                                                inputBuf[index_write] = inElem;
                                                inp++;
                                                internal_counter++;
                                                if (internal_counter==buffer_size){
                                                        internal_counter=0;
                                                }
                                        }
                                        else{
                                                internal_counter++;
                                        }
                                }

                                // Write output
                                ap_uint<SIMD*Input_precision> outElem = inputBuf[index_read];
                                out.write(outElem);

                                // Update read index pointer
                                if (j < ConvKernelDim_x-1){
                                        index_read = index_read + multiplying_factor;
                                        j++;
                                }
                                else{
                                        index_read = index_read + (multiplying_factor+1);
                                        j=0;
                                }
                                index_read = index_read%(buffer_size);

                                }
                } // End base_iter
        } // End count_image
} // End generator
```

C

Alveo U250 device utilisation

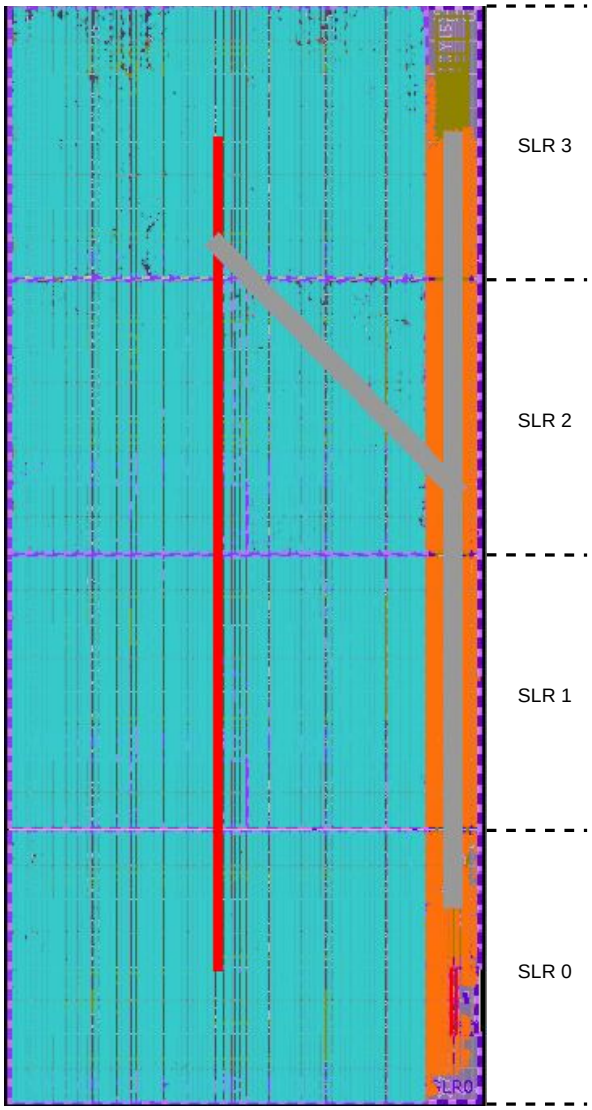


Figure C.1: Alveo U250 device utilisation of the quantized QuartzNet model with optimised FIFO sizing post-synthesis and implementation.

Bibliography

[1] Intel® xeon® processor e5-2667. URL https://ark.intel.com/content/www/us/en/ark/products/64589/intel-xeon-processor-e5-2667-15m-cache-2-90-ghz-8\protect\@normalcr\relax-00-gt-s-intel-qpi.html.

[2] Onnx operator schemas. URL https://github.com/onnx/onnx/blob/master/docs/Operators.md.

[3] Ossama Abdel-Hamid, Abdel rahman Mohamed, Hui Jiang, and Gerald Penn. Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition. *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4277–4280, 2012.

[4] Tobias Alonso, Lucian Petrica, Mario Ruiz, Jakoba Petri-Koenig, Yaman Umuroglu, Ioannis Stamelos, Elias Koromilas, Michaela Blott, and Kees Vissers. Elastic-df: Scaling performance of dnn inference in fpga clouds through automatic partitioning. 2021.

[5] Zhu Baozhou, Nauman Ahmed, Johan Peltenburg, Koen Bertels, and Zaid Al-Ars. Diminished-1 fermat number transform for integer convolutional neural networks. In *2019 IEEE 4th International Conference on Big Data Analytics (ICBDA)*, pages 47–52. IEEE, 2019.

[6] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning?, 2020.

[7] Michaela Blott, Thomas Preusser, Nicholas 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, 2018.

[8] Cameron Buckner and James Garson. Connectionism. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Fall 2019 edition, 2019.

[9] William Chan, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals. Listen, attend and spell, 2015.

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

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning, 2014.

[12] Christopher Cieri, David Miller, and Kevin Walker. The fisher corpus: A resource for the next generations of speech-to-text. 01 2004.

[13] Gianmarco Dinelli, Gabriele Meoni, Emilio Rapuano, Gionata Benelli, and Luca Fanucci. An fpga-based hardware accelerator for cnns using on-chip memories only: Design and benchmarking with intel movidius neural compute stick. *International Journal of Reconfigurable Computing*, 2019, 10 2019. doi: 10.1155/2019/7218758.

[14] Haytham Fayek. Speech processing for machine learning: Filter banks, mel-frequency cepstral coefficients (mfccs) and what's in-between. URL https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html.

[15] Linux Foundation. Open standard for machine learning interoperability. URL https://github.com/onnx/onnx.

[16] Giuseppe Franco. Quantized quartznet with brevitas for efficient speech recognition, 3 2020. URL https://xilinx.github.io/finn/2020/03/27/brevitas-quartznet-release.html.

[17] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference, 2021.

[18] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, and Ruoming Pang. Conformer: Convolution-augmented transformer for speech recognition, 2020.

[19] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. [dl] a survey of fpga-based neural network inference accelerators. *ACM Trans. Reconfigurable Technol. Syst.*, 12(1), March 2019. ISSN 1936-7406. doi: 10.1145/3289185. URL https://doi.org/10.1145/3289185.

[20] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William J. Dally. Ese: Efficient speech recognition engine with sparse lstm on fpga, 2017.

[21] Awni Hannun. Sequence modeling with ctc. URL https://distill.pub/2017/ctc/.

[22] Awni Hannun, Ann Lee, Qiantong Xu, and Ronan Collobert. Sequence-to-sequence speech recognition with time-depth separable convolutions, 2019.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[24] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9: 1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

[25] Joost Hoozemans, Johan Peltenburg, Fabian Nonnemacher, Akos Hadnagy, Zaid Al-Ars, and H. Peter Hofstee. Fpga acceleration for big data analytics: Challenges and opportunities. *IEEE Circuits and Systems Magazine*, 21(2):30–47, 2021. doi: 10.1109/MCAS.2021.3071608.

[26] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014. doi: 10.1109/ISSCC.2014.6757323.

[27] 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.

[28] Y. Bengio & A. Courville I. Goodfellow. *Deep Learning*. MIT Press, 2017.

[29] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[30] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[31] R. Kastner, J. Matai, and S. Neuendorffer. Parallel Programming for FPGAs. *ArXiv e-prints*, May 2018.

[32] Sehoon Kim, Amir Gholami, Zhewei Yao, Anirudda Nrusimha, Bohan Zhai, Tianren Gao, Michael W. Mahoney, and Kurt Keutzer. Q-asr: Integer-only zero-shot quantization for efficient speech recognition, 2021.

[33] Allison Koenecke, Andrew Nam, Emily Lake, Joe Nudell, Minnie Quartey, Zion Mengesha, Connor Toups, John R. Rickford, Dan Jurafsky, and Sharad Goel. Racial disparities in automated speech recognition. *Proceedings of the National Academy of Sciences*, 117(14):7684–7689, 2020. ISSN 0027-8424. doi: 10.1073/pnas.1915768117. URL https://www.pnas.org/content/117/14/7684.

[34] 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.

[35] M.I. Kroes. Optimizing memory mapping for dataflow inference accelerators. Master's thesis, Delft University of Technology, 2020.

[36] Xilinx Research Labs. Finn: Dataflow compiler for qnn inference on fpgas, . URL https://github.com/Xilinx/finn.

[37] Xilinx Research Labs. Open source compiler framework using onnx as frontend and ir, . URL https://github.com/Xilinx/finn-base.

[38] Xilinx Research Labs. Dataflow qnn inference accelerator examples on fpgas, . URL https://github.com/Xilinx/finn-examples.

[39] Xilinx Research Labs. Finn-experimental, . URL https://github.com/Xilinx/finn-experimental.

[40] Xilinx Research Labs. Vivado hls library for finn, . URL https://github.com/Xilinx/finn-hlslib.

[41] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989. doi: 10.1162/neco.1989.1.4.541.

[42] Minjae Lee, Kyuyeon Hwang, Jinhwan Park, Sungwook Choi, Sungho Shin, and Wonyong Sung. Fpga-based low-power speech recognition with recurrent neural networks. In *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 230–235, 2016. doi: 10.1109/SiPS.2016.48.

[43] Jason Li, Vitaly Lavrukhin, Boris Ginsburg, Ryan Leary, Oleksii Kuchaiev, Jonathan M. Cohen, Huyen Nguyen, and Ravi Teja Gadde. Jasper: An end-to-end convolutional neural acoustic model, 2019.

[44] Zhe Li, Caiwen Ding, Siyue Wang, Wujie Wen, Youwei Zhuo, Chang Liu, Qinru Qiu, Wenyao Xu, Xue Lin, Xuehai Qian, and Yanzhi Wang. E-rnn: Design optimization for efficient recurrent neural networks in fpgas. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 69–80, 2019. doi: 10.1109/HPCA.2019.00028.

[45] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing*, 461:370–403, 2021. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2021.07.045. URL https://www.sciencedirect.com/science/article/pii/S0925231221010894.

[46] Tatiana Likhomanenko, Qiantong Xu, Vineel Pratap, Paden Tomasello, Jacob Kahn, Gilad Avidov, Ronan Collobert, and Gabriel Synnaeve. Rethinking evaluation in asr: Are our models robust enough?, 2021.

[47] Mozilla. Mozilla common voice. URL https://commonvoice.mozilla.org/en/datasets.

[48] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization, 2021.

[49] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning, 2018.

[50] Dimitri Palaz, Mathew Magimai.-Doss, and Ronan Collobert. Convolutional neural networks-based continuous speech recognition using raw speech signal. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4295–4299, 2015. doi: 10.1109/ICASSP.2015.7178781.

[51] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An asr corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210, 2015. doi: 10.1109/ICASSP.2015.7178964.

[52] Alessandro Pappalardo. Xilinx/brevitas. URL https://doi.org/10.5281/zenodo.3333552.

[53] Douglas B. Paul and Janet M. Baker. The design for the Wall Street Journal-based CSR corpus. In *Speech and Natural Language: Proceedings of a Workshop Held at Harriman, New York, February 23-26, 1992*, 1992. URL https://aclanthology.org/H92-1073.

[54] J. Peltenburg, J. van Straten, L. Wijtemans, L. van Leeuwen, Z. Al-Ars, and P. Hofstee. Fletcher: A framework to efficiently integrate fpga accelerators with apache arrow. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 270–277, Sep. 2019. doi: 10.1109/FPL.2019.00051.

[55] Johan Peltenburg, Jeroen van Straten, Matthijs Brobbel, H. Peter Hofstee, and Zaid Al-Ars. Supporting Columnar In-memory Formats on FPGA: The Hardware Design of Fletcher for Apache Arrow. In Christian Hochberger, Brent Nelson, Andreas Koch, Roger Woods, and Pedro Diniz, editors, *Applied Reconfigurable Computing*, pages 32–47, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17227-5.

[56] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

[57] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge, 2015.

[58] Laurent Sifre and Prof Stéphane Mallat. Ecole polytechnique, cmap phd thesis rigid-motion scattering for image classification author, 2014.

[59] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[60] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

[61] Piotr Szymański, Piotr Żelasko, Mikolaj Morzy, Adrian Szymczak, Marzena Żyła Hoppe, Joanna Banaszczak, Lukasz Augustyniak, Jan Mizgajski, and Yishay Carmiel. Wer we are and wer we think we are, 2020.

[62] Yaman Umuroglu and Magnus Jahre. Streamlined deployment for quantized neural networks, 2018.

[63] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn. *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb 2017. doi: 10.1145/3020078.3021744. URL http://dx.doi.org/10.1145/3020078.3021744.

[64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[65] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions, 2018.

[66] Nils Voss, Pablo Quintana, Oskar Mencer, Wayne Luk, and Georgi Gaydadjiev. Memory mapping for multi-die fpgas. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 78–86, 2019. doi: 10.1109/FCCM.2019.00021.

[67] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Yanzhi Wang, Qinru Qiu, and Yun Liang. C-lstm: Enabling efficient lstm using structured compression techniques on fpgas, 2018.

[68] Shinji Watanabe, Michael Mandel, Jon Barker, Emmanuel Vincent, Ashish Arora, Xuankai Chang, Sanjeev Khudanpur, Vimal Manohar, Daniel Povey, Desh Raj, David Snyder, Aswin Shanmugam Subramanian, Jan Trmal, Bar Ben Yair, Christoph Boeddeker, Zhaoheng Ni, Yusuke Fujita, Shota Horiguchi, Naoyuki Kanda, Takuya Yoshioka, and Neville Ryant. CHiME-6 Challenge: Tackling Multispeaker Speech Recognition for Unsegmented Recordings. In *Proc. The 6th International Workshop on Speech Processing in Everyday Environments (CHiME 2020)*, pages 1–7, 2020. doi: 10.21437/CHiME.2020-1. URL http://dx.doi.org/10.21437/CHiME.2020-1.

[69] Dong Wen, Jingfei Jiang, Yong Dou, Jinwei Xu, and Tao Xiao. An energy-efficient convolutional neural network accelerator for speech classification based on fpga and quantization. *CCF Transactions on High Performance Computing*, 3, 01 2021. doi: 10.1007/s42514-020-00055-4.

[70] Ran Wu, Xinmin Guo, Jian Du, and Junbao Li. Accelerating neural network inference on fpga-based platforms—a survey. *Electronics*, 10(9), 2021. ISSN 2079-9292. doi: 10.3390/electronics10091025. URL https://www.mdpi.com/2079-9292/10/9/1025.

[71] *Large FPGA Methodology Guide*. Xilinx, 10 2012.

[72] *UltraScale Architecture Configurable Logic Block*. Xilinx, 2 2017.

[73] *Introduction to FPGA Design with Vivado High-Level Synthesis*. Xilinx, 1 2019.

[74] *Alveo U200 and U250 Data Center Accelerator Cards Data Sheet*. Xilinx, 5 2020.

[75] *UltraScale Architecture Memory Resources User Guide*. Xilinx, 3 2021.

[76] *Vivado Design Suite User Guide*. Xilinx, 5 2021.

[77] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8bert: Quantized 8bit bert, 2019.

[78] Neil Zeghidour, Nicolas Usunier, Gabriel Synnaeve, Ronan Collobert, and Emmanuel Dupoux. End-to-end speech recognition from the raw waveform, 2018.

[79] Albert Zeyer, André Merboldt, Wilfried Michel, Ralf Schlüter, and Hermann Ney. Librispeech transducer model with internal language model prior correction, 2021.

[80] Yu Zhang, James Qin, Daniel S. Park, Wei Han, Chung-Cheng Chiu, Ruoming Pang, Quoc V. Le, and Yonghui Wu. Pushing the limits of semi-supervised learning for automatic speech recognition, 2020.

[81] Baozhou Zhu, Zaid Al-Ars, and H Peter Hofstee. Nasb: Neural architecture search for binary convolutional neural networks. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020.

[82] Baozhou Zhu, Zaid Al-Ars, and Wei Pan. Towards lossless binary convolutional neural networks using piecewise approximation. In *European Conference on Artificial Intelligence (ECAI)*, 2020.