AUTOMATICALLY GENERATING FAST HARDWARE IMPLEMENTATIONS OF DEEPLY QUANTIZED NEURAL NETWORKS

Jure Vreča

Doctoral Dissertation
Jožef Stefan International Postgraduate School
Ljubljana, Slovenia

Supervisor: Asst. Prof. Anton Biasizzo, Jožef Stefan International Postgraduate School, Ljubljana, Slovenia

Evaluation Board:
Prof. Dr. Aleš Švigelj, Chair, Jožef Stefan Institute, Ljubljana, Slovenia
Prof. Dr. Andrej Žemva, Member, Faculty of Electrical Engineering - University of Ljubljana, Ljubljana, Slovenia
Dr. Vladimir Lončar, Member, European Organization for Nuclear Research (CERN), Geneva, Switzerland

MEDNARODNA PODIPLOMSKA ŠOLA JOŽEFA STEFANA
JOŽEF STEFAN INTERNATIONAL POSTGRADUATE SCHOOL



Jure Vreča

AUTOMATICALLY GENERATING FAST HARDWARE IMPLEMENTATIONS OF DEEPLY QUANTIZED NEURAL NETWORKS

Doctoral Dissertation

SAMODEJNO TVORJENJE HITRIH STROJNIH IMPLEMENTACIJ MOČNO KVANTIZIRANIH NEVRONSKIH MREŽ

Doktorska disertacija

Supervisor: Asst. Prof. Anton Biasizzo

Ljubljana, Slovenia, August 2025

Acknowledgments

Writing this thesis and researching this topic has been a long journey that I am glad has finally come to an end. This journey was made much easier by the support of my girlfriend, my family, and my friends.

I would also like to thank my supervisor for his guidance and thorough reading of the text. Your detailed comments and questions forced me to be exhaustive and helped me grow as an academic. I would also like to thank my colleagues for the lively discussions that generated many insightful thoughts and ideas.

I thank the hls4ml team for their openness and for answering my many questions in the FastML Slack channel.

I would also like to thank the Slovenian Research and Innovation Agency for the financial support (research core funding No. P2-0098). This work was also part of a project funded by the ECSEL Joint Undertaking under grant agreement No 101007273 (DAIS).

Abstract

This thesis studies the automatic design of hardware for ultra-low latency inference of neural networks. Such hardware fills a niche in the application of neural networks where results on the order of a few microseconds are required. An example use case for such hardware is the trigger system at the CERN Large Hadron Collider, which has to react in microseconds to prevent a data buffer overflow due to a data stream of terabytes per second. This is unachievable for most of today's hardware, as it is not optimized for ultra-low latency, but rather for high throughput. In addition, modern hardware accelerators almost universally use various forms of DRAM memory to store the parameter data of neural networks. Such external memories use busses that incur a high latency penalty.

The key to achieving ultra-low latency is to remove the external memory, store all data on-chip, and process the neural network in a highly parallel manner. However, this is problematic because neural networks typically have a large number of parameters and the storage capacity of the on-chip memory is limited. To address this issue, two neural network optimization techniques are investigated: quantization and pruning. The quantization techniques use advanced training methods that can quantize neural networks to use parameters with very small bitwidth. The second optimization technique is pruning. It focuses on removing unnecessary connections from the neural network and thus eliminating redundant parameters. By using these two methods together, non-trivial neural networks can be implemented on modern FPGA circuits in a highly parallel manner.

In this work, we develop a methodology for implementing highly parallel hardware implementations of quantized neural networks based on hardware generators that use structural descriptions. One of the goals of this approach is to provide an alternative to solutions based on high-level synthesis algorithms that have difficulty scaling to larger neural networks, since such algorithms have polynomial time complexity. We develop an open source tool based on the Chisel Hardware Construction Language that is a practical implementation of our method and is capable of generating direct circuit implementations of quantized and pruned neural networks. We compare our tool with similar tools based on high-level synthesis technology.

Povzetek

Tema doktorske naloge je samodejno snovanje strojnih implementacij za računanje nevronskih mrež, ki dosegajo ultra nizke zakasnitve. Take strojne implementacije nevronskih mrež, kjer so zakasnitve v redu nekaj mikrosekund, so uporabne v določenih nišnih aplikacijah. Primer uporabe takšnih strojnih implementacij je prožil sistem v CERN-u, kjer podatkovni toki iz senzoričnih sistemov dosegajo hitrosti terazlogov na sekundo. Ta sistem mora prožiti z mikrosekundno zakasnitvijo, da prepreči preplavljanje medpomnilnika. Takšne zakasnitve so izven dosega večine strojne opreme, saj so po večini le-te optimizirane za visoko prepustnost. Velika večina sodobne strojne opreme uporablja razne oblike dinamičnega pomnilnika za hranjenje parametrov nevronskih mrež. Takšni zunanji pomnilniki uporabljajo vodila, ki imajo že sama visoko zakasnitev.

Za dosego ultra nizkih zakasnitev je ključno, da sistem ne uporablja zunanjih pomnilnikov in da so vsi parametri shranjeni lokalno na čipu. Pomembno je tudi, da nevronsko mrežo procesiramo na zelo vzporeden način. Tega pa ni lahko doseči, ker imajo nevronske mreže tipično veliko število parametrov; lokalni pomnilnik pa je omejen. Dve tehniki se pogosto uporabljata v izogib temu problemu: kvantizacija in rezanje. Z uporabo naprednih tehnik učenja nevronskih lahko naučimo mreže, katerih parametri so kvantizirani na zelo majhne bitne širine. Druga tehnika optimizacije nevronskih mrež pa se imenuje rezanje sinaps. Pri tej tehniki odstranjujemo odvečne povezave v nevronskih mrežah, in s tem odstranimo tudi odvečne parametre. Skupaj ti tehniki optimizacije omogočata implementacije nevronskih mrež na sodobnih programirljivih vezjih na zelo vzporeden način.

V tem delu razvijemo metodologijo za razvoj zelo vzporednih strojnih implementacij kvantiziranih nevronskih mrež na osnovi generatorjev, ki uporabljajo strukturne opise vezij. Eden izmed ciljev je ustvariti alternativno rešitev orodjem, ki so osnovana z uporabo metode visoko nivojske sinteze. Te metode imajo težave pri večjih nevronskih mrežah, saj visoko nivojska sinteza uporablja algoritme, ki imajo polinomsko časovno kompleksnost. V tem delu razvijemo orodje na osnovi jezika za opis strojne opreme Chisel. To orodje je sposobno tvoriti direktne strojne implementacije kvantiziranih in rezanih nevronskih mrež. Svoje orodje tudi primerjamo s podobnimi orodji, ki pa so osnovana na tehnologiji visoko nivojske sinteze.

Contents

List of Figures **xv**

List of Tables **xix**

List of Algorithms **xxi**

Abbreviations **xxiii**

Symbols **xxv**

1 Introduction **1**
1.1 Hardware for Accelerating ANN Processing . . . 1
1.1.1 Throughput Optimized Hardware . . . 2
1.1.2 Ultra-Low Latency Hardware . . . 2
1.2 Hypothesis . . . 3
1.3 Scientific Contributions . . . 3
1.4 Thesis Structure . . . 4

2 Artificial Neural Networks **5**
2.1 The Neuron Model . . . 5
2.2 Neural Network Layers . . . 6
2.2.1 Fully Connected Layer . . . 6
2.2.2 Convolution Layer . . . 7
2.2.3 Pooling Layer . . . 8
2.2.4 Batch-Normalization Layer . . . 8
2.3 Feed-Forward Neural Networks . . . 9
2.3.1 Neural Network Training . . . 9
2.3.2 Folding Batch Normalization Layers . . . 10
2.4 Quantization . . . 11
2.4.1 Basics of Quantization . . . 11
2.4.2 Quantizing Neural Networks . . . 13
2.4.2.1 Post-Training Quantization . . . 13
2.4.2.2 Quantization-Aware Training . . . 13
2.4.2.3 Scale Factors . . . 14
2.4.3 Binarized Neural Networks . . . 15
2.5 Pruning . . . 16
2.6 DQNN Training Frameworks and Graph Representations of Neural Networks 17
2.6.1 Open Neural Network eXchange Format . . . 17
2.6.2 Quantized Open Neural Network eXchange Format . . . 18
2.6.3 Tensor Layouts . . . 19

3 Hardware **23**

3.1 Field-Programmable Gate Array . . . 23
3.2 High-Level Synthesis . . . 24
3.3 Chisel . . . 24

4 State of the Art **29**
4.1 hls4ml . . . 29
4.1.1 Backend HLS Implementations . . . 31
4.1.1.1 Fully Connected Layer Implementations . . . 31
4.1.1.2 Convolution Layer Implementations . . . 33
4.1.1.3 Pooling Layer Implementations . . . 35
4.2 FINN . . . 36
4.2.1 Matrix-Vector-Activation Unit . . . 37
4.2.2 Sliding Window Unit . . . 38
4.2.3 Vector-Vector-Activation Unit . . . 39
4.2.4 Pooling Unit . . . 40

5 chisel4ml **41**
5.1 Low-Bitwidth Intermediate Representation Format . . . 41
5.2 Python Frontend . . . 43
5.2.1 Transforming DQNN to LBIR . . . 45
5.3 Chisel Backend . . . 46
5.3.1 Representing Quantization Information . . . 47
5.3.2 A Combinational Neuron Implementation . . . 48
5.3.3 Combining Neurons into Layers . . . 49
5.3.4 Implementing Maximum Pooling Layers . . . 51
5.3.5 Generating the Entire Circuit . . . 52

6 Results **53**
6.1 Individual Layer Experiments . . . 55
6.1.1 Fully Connected Layer . . . 55
6.1.2 Convolution Layer . . . 58
6.1.3 Maximum Pooling Layer . . . 61
6.2 Jet Tagging Multilayer Perceptron . . . 64
6.3 MNIST Digit Recognition . . . 67

7 Conclusions **71**
7.1 Hypothesis . . . 71
7.2 Synopsis . . . 72
7.3 Scientific Contributions . . . 73
7.4 Future Work . . . 73

Appendix A Plots of All Experiment Results **75**
A.1 Fully Connected Layer - Input Features . . . 76
A.2 Fully Connected Layer - Output Features . . . 77
A.3 Fully Connected Layer - Input Bitwidth . . . 78
A.4 Fully Connected Layer - Weight Bitwidth . . . 79
A.5 Convolution Layer - Input Channels . . . 80
A.6 Convolution Layer - Output Channels . . . 81
A.7 Convolution Layer - Input Bitwidth . . . 82
A.8 Convolution Layer - Weight Bitwidth . . . 83
A.9 Maximum Pooling Layer - Input Size . . . 84

A.10 Maximum Pooling Layer - Channels . . . 85
A.11 Maximum Pooling Layer - Kernel Size . . . 86
A.12 Maximum Pooling Layer - Input Bitwidth . . . 87
A.13 Jet Tagging MLP - Bitwidth . . . 88
A.14 Jet Tagging MLP - Prune Rate . . . 89
A.15 MNIST CNN - Bitwidth . . . 90
A.16 MNIST CNN - Prune Rate . . . 91

Appendix B Table of All Experiment Results **93**
B.1 Fully Connected Layer - Input Features . . . 94
B.2 Fully Connected Layer - Output Features . . . 95
B.3 Fully Connected Layer - Input Bitwidth . . . 96
B.4 Fully Connected Layer - Weights Bitwidth . . . 97
B.5 Convolution Layer - Input Channels . . . 98
B.6 Convolution Layer - Output Channels . . . 99
B.7 Convolution Layer - Input Bitwidth . . . 100
B.8 Convolution Layer - Weights Bitwidth . . . 101
B.9 Maximum Pooling Layer - Input Size . . . 102
B.10 Maximum Pooling Layer - Channels . . . 103
B.11 Maximum Pooling Layer - Kernel Size . . . 104
B.12 Maximum Pooling Layer - Input Bitwidth . . . 105
B.13 Jet Tagging MLP - Bitwidth . . . 106
B.14 Jet Tagging MLP - Prune Rate . . . 107
B.15 MNIST CNN - Bitwidth . . . 108
B.16 MNIST CNN - Prune Rate . . . 109

References **111**

Bibliography **115**

Biography **117**

List of Figures

Figure 1.1: A typical hardware configuration. . . . 2
Figure 1.2: Internal architectures of PPUs. . . . 2
Figure 1.3: Block diagram of an hypothetical DFPU design. . . . 3

Figure 2.1: A rectified linear unit (ReLU) activation function on the left and the sigmoid activation function on the right. . . . 6
Figure 2.2: A fully connected layer with two neurons and three inputs 6
Figure 2.3: A simple convolution operation. . . . 8
Figure 2.4: A simple maximum pooling operation. . . . 8
Figure 2.5: A graph of a fully connected layer followed by a batch normalization layer in (*a*) and the same graph with the batch normalization layer folded into the $Weights$ in (*b*). . . . 10
Figure 2.6: Plot of the function $y = x$ without quantization, with uniform quantization and with power of two quantization (non-uniform). . . . 12
Figure 2.7: Computational graphs of a convolution layer. . . . 14
Figure 2.8: Efficient binarized values encoding. . . . 15
Figure 2.9: A fully connected layer with four inputs and two neurons. . . . 16
Figure 2.10: An ONNX graph of a fully connected layer. . . . 18
Figure 2.11: An ONNX graph representation of a simple CNN. . . . 18
Figure 2.12: A quantized convolution layer. . . . 19
Figure 2.13: A 2 by 3 RGBA image, where alpha equals 1 for all pixels. . . . 20
Figure 2.14: RGBA image stored in a 32 bit byte-addressable memory using (a) HWC layout or (b) CHW layout. . . . 20

Figure 3.1: Typicall FPGA architecture. . . . 23
Figure 3.2: CLB architecture. . . . 23
Figure 3.3: Chisel type hierarchy [34]. . . . 25
Figure 3.4: A simple moving average FIR filter. . . . 26
Figure 3.5: A reduce operation on a sequence of four elements using the addition as the reduction operator. . . . 27

Figure 4.1: The hls4ml workflow [2]. . . . 30
Figure 4.2: The hls4ml software architecture. . . . 31
Figure 4.3: The processing element [38]. . . . 38
Figure 4.4: Matrix-Vector-Activation Unit [38]. . . . 38
Figure 4.5: The operation of the Sliding Window Unit [39]. . . . 39

Figure 5.1: High-level software architecture of chisel4ml. . . . 41
Figure 5.2: A QONNX graph of a quantized convolution layer. . . . 46
Figure 5.3: Partially transformed QONNX graph with Quant nodes transformed into QTensor nodes. . . . 46
Figure 5.4: The final LBIR model. . . . 46

Figure 5.5: Mapping the inputs to the appropriate neurons for a fully connected layer in (a) and a two-dimensional convolution layer in (b). 50

Figure 6.1: The workflow of the experiments. 54
Figure 6.2: Single fully connected layer experiment in which the number of input features is varied. The number of output features equals 32. The input, output and weights are quantized to 4 bits and the bias is quantized to 8 bits. 56
Figure 6.3: Single fully connected layer experiment in which the number of output features is varied. The number of input features equals 16. The input, output and weights are quantized to 4 bits and the bias is quantized to 8 bits. 56
Figure 6.4: Single fully connected layer experiment in which the number of bits used to quantize the input features is varied. The number of input and output features equals 32. The weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits. 57
Figure 6.5: Single fully connected layer experiment in which the number of bits used to quantize the weight parameters is varied. The number of input and output features equals 32. The inputs and outputs are quantized to 4 bits and the bias is quantized to 8 bits. 57
Figure 6.6: Single convolution layer experiment in which we vary the number of input channels of the convolution layer. The input size is set to an 8 by 8 square and the kernel size is set to 3 by 3. The number of output channels is set to 1. The inputs, weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits. 59
Figure 6.7: Single convolution layer experiment in which the number of output channels of the convolution layer is varied. The input size is set to an 16 by 16 square and the kernel size is set to 3 by 3. The number of input channels is set to 1. The inputs, weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits. 59
Figure 6.8: Single convolution layer experiment in which the number of bits used to quantize the input is varied. The input size is set to an 16 by 16 square and the kernel size is set to 3 by 3. The number of input and output channels is set to 1. The weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits. 60
Figure 6.9: Single convolution layer experiment in which the number of bits used to quantize the weight parameters is varied. The input size is set to an 16 by 16 square and the kernel size is set to 3 by 3. The number of input and output channels is set to 1. The inputs and outputs are quantized to 4 bits, and the bias is quantized to 8 bits. 60
Figure 6.10: Linear versus tree style reduction. 61
Figure 6.11: Single maximum pooling layer experiment in which the input size is varied whilst keeping it square. The size of the kernel is 3 by 3 and the number of channels is set to 3. The inputs to the layer are quantized to 4 bits. 62
Figure 6.12: Single maximum pooling layer experiment in which the number of channels of the input tensor is varied. The input size is set to an 8 by 8 square and the kernel size is set to 3 by 3. The inputs to the layer are quantized to 4 bits. 62

Figure 6.13: Single maximum pooling layer experiment in which the size of the square kernel is varied. The number of channels is set to 3 and input size to an 8 by 8 square. The inputs to the layer are quantized to 4 bits. . . . 63
Figure 6.14: Single maximum pooling layer experiment in which the number of bits used to quantize the input is varied. The number of channels is set to 3 and the input size to an 8 by 8 square. The kernel size is set to 2 by 2. 63
Figure 6.15: Accuracies of the MLP models. 65
Figure 6.16: The multilayer perceptron experiment in which the number of bits used to quantize the weights and activations is varied. The inputs to the MLP are quantized to 8 bits. The global prune rate is set to 0.5. 66
Figure 6.17: The multilayer perceptron experiment in which the global prune rate is varied. The weights and activations are quantized to 4 bits. The inputs to the MLP are quantized to constant 8 bits. 66
Figure 6.18: Accuracies of the CNN models on the MNIST test dataset. 68
Figure 6.19: The convolutional neural network experiment in which the number of bits used to quantize the weights and activations is varied. The inputs to the CNN are quantized to constant 8 bits. The global prune rate is set to 0.5. 69
Figure 6.20: The convolutional neural network experiment in which the global prune rate is varied. The weights and activations are quantized to 3 bits. The inputs to the CNN are quantized to constant 8 bits. 69

List of Tables

Table 3.1: Width inference rules in Chisel [35]. 26

Table 6.1: Topology of multilayer perceptron models after folding the batch normalization layers. 64
Table 6.2: Topology of the convolutional neural networks after folding the batch normalization layers. 67

Table A.1: Quantity description table. 75

Table B.1: Description of shorthands. 93
Table B.2: Single fully connected layer experiment in which the number of input features is varied. The number of output features equals 32. The input, output and weights are quantized to 4 bits and the bias is quantized to 8 bits. 94
Table B.3: Single fully connected layer experiment in which the number of output features is varied. The number of input features equals 16. The input, output and weights are quantized to 4 bits and the bias is quantized to 8 bits. 95
Table B.4: Single fully connected layer experiment in which the number of bits used to quantize the input features is varied. The number of input and output features equals 32. The weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits. 96
Table B.5: Single fully connected layer experiment in which the number of bits used to quantize the weight parameters is varied. The number of input and output features equals 32. The inputs and outputs are quantized to 4 bits and the bias is quantized to 8 bits. 97
Table B.6: Single convolution layer experiment in which the number of input channels of the convolution layer is varied. The input size is set to an 8 by 8 square and the kernel size is set to 3 by 3. The number of output channels is set to 1. The inputs, weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits. 98
Table B.7: Single convolution layer experiment in which the number of output channels of the convolution layer is varied. The input size is set to an 16 by 16 square and the kernel size is set to 3 by 3. The number of input channels is set to 1. The inputs, weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits. 99
Table B.8: Single convolution layer experiment in which the number of bits used to quantize the input is varied. The input size is set to an 16 by 16 square and the kernel size is set to 3 by 3. The number of input and output channels is set to 1. The weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits. 100

Table B.9: Single convolution layer experiment in which the number of bits used to quantize the weight parameters is varied. The input size is set to an 16 by 16 square and the kernel size is set to 3 by 3. The number of input and output channels is set to 1. The inputs and outputs are quantized to 4 bits, and the bias is quantized to 8 bits. 101
Table B.10: Single maximum pooling layer experiment in which the input size whilst keeping it square is varied. The size of the kernel is 3 by 3 and the number of channels is set to 3. The inputs to the layer are quantized to 4 bits. 102
Table B.11: Single maximum pooling layer experiment in which the number of channels of the input tensor is varied. The input size is set to an 8 by 8 square and the kernel size is set to 3 by 3. The inputs to the layer are quantized to 4 bits. 103
Table B.12: Single maximum pooling layer experiment in which the size of the square kernel is varied. The number of channels is set to 3 and input size to an 8 by 8 square. The inputs to the layer are quantized to 4 bits. 104
Table B.13: Single maximum pooling layer experiment in which the number of bits used to quantize the input is varied. The number of channels is set to 3 and the input size to an 8 by 8 square. The kernel size is set to 2 by 2. . 105
Table B.14: The multilayer perceptron experiment in which the number of bits used to quantize the weights and activations is varied. The inputs to the MLP are quantized to 8 bits. The global prune rate is set to 0.5. 106
Table B.15: The multilayer perceptron experiment in which the global prune rate is varied. The weights and activations are quantized to 4 bits. The inputs to the MLP are quantized to constant 8 bits. 107
Table B.16: The convolutional neural network experiment in which the number of bits used to quantize the weights and activations is varied. The inputs to the CNN are quantized to constant 8 bits. The global prune rate is set to 0.5. 108
Table B.17: The convolutional neural network experiment in which the global prune rate is varied. The weights and activations are quantized to 3 bits. The inputs to the CNN are quantized to constant 8 bits. 109

List of Algorithms

Algorithm 2.1: The Batch Normalization procedure [11]. 9

Abbreviations

ANN	...	Artificial Neural Network
AXI	...	Advanced eXtensible Interconnect
BNN	...	Binarized Neural Network
CERN	...	European Organization for Nuclear Research
Chisel	...	Constructing Hardware In a Scala Embedded Language
CLB	...	Configurable Logic Block
CNN	...	Convolutional Neural Network
CPU	...	Central Processing Unit
DDR	...	Double Data Rate
DFPU	...	Data Flow Processing Unit
DQNN	...	Deeply Quantized Neural Network
DRAM	...	Dynamic Random Access Memory
EDA	...	Electronic Design Automation
FF	...	Flip-Flop
FIR	...	Finite Impulse Response
FPGA	...	Field-Programmable Gate Array
GPGPU	...	General-Purpose Graphics Processing Unit
HCL	...	Hardware Construction Language
HDL	...	Hardware Description Language
HLS	...	High-Level Synthesis
IOB	...	Input-Output Buffer
LBIR	...	Low-Bitwidth Intermediate Representation
LHC	...	Large Hadron Collider
LUT	...	Look-Up Table
MVAU	...	Matrix-Vector-Activation Unit
NPU	...	Neuron Processing Unit
ONNX	...	Open Neural Network eXchange
PE	...	Processing Element. A part of MVAU.
PPU	...	Programmable Processing Unit
PU	...	Processing Unit. A part of DFPU.
PTQ	...	Post-Training Quantization
QAT	...	Quantization-Aware Training
QNN	...	Quantized Neural Network
QONNX	...	Quantized Open Neural Network eXchange
ReLU	...	Rectified Linear Unit
SB	...	Switch Box
SIMD	...	Single Instruction Multiple Data
SWU	...	Sliding Window Unit
TPU	...	Tensor Processing Unit
VVAU	...	Vector-Vector-Activation Unit

Symbols

y ... Scalar output of a neuron.
$\vec{y}$ ... Vector output of a layer, e.g., a fully connected layer.
$\vec{x}$ ... Input vector to a layer, neuron, etc.
b ... A scalar bias constant.
$\vec{b}$ ... A vector of bias values.
$\vec{w}$ ... The weight vector of a neuron
M ... The size dimension of vectors, e.g., length of input vector.
N ... The size of the batch dimension.
W ... The weight matrix of a fully connected layer.
Y ... Output tensor of a layer.
X ... Input tensor of a layer.
F ... The filter weight tensor of a convolution layer.
C ... The channel dimension, i.e., number of channels.
K ... The number of kernels of a convolution layer.
H_F ... Height dimension of the filter.
W_F ... Width dimension of the filter.
H_I ... The height of the input tensor X.
W_I ... The width of the input tensor X.
G ... A function representing a feed-forward neural network.
g_i ... A function representing a particular layer of a feed-forward neural network.
L ... The loss function.
$\hat{Y}_i$ ... The expected output of a neural network G when the input to the network is X_i.
T ... The training dataset.
q ... A quantization function.
q_u ... A uniform quantization function.
$\mathbb{U}_n$ ... The set of all n bit unsigned integers.
$\mathbb{S}_n$ ... The set of all n bit signed integers.
$\mathbb{I}_n$ ... The set of all n bit integers without a specified signedness.
$\hat{q}$ ... A fake quantization function.
$\mathbb{F}_{32}$ ... The set of all 32 bit floating-point numbers.
$\mathbb{F}_{32}^{[\alpha,\beta]}$ ... The set of all 32 bit floating-point numbers in the inclusive range from α to β.
$\mathbb{Q}_n^{[\alpha,\beta]}$ ... A subset of $\mathbb{F}_{32}^{[\alpha,\beta]}$ that has the same cardinality as some set $\mathbb{I}_n$ and where the values of the set are roughly equidistant.
S ... Scale factor.

Chapter 1

Introduction

Artificial Neural Networks (ANN) are machine learning algorithms with a plethora of applications in science and industry. They allow us to solve both classification and regression problems when a sufficient amount of data is available.

The general trend in the development of ANN topologies is to design larger and larger models that contain a vast amount of parameters. Larger models require more computing power, and so the progress of ANN models in this way is linked to the available computing power of the hardware. A major leap in the performance of neural network models occurred in 2012, when the first deep neural networks were trained with GPGPUs [1]. Deep neural networks are ANNs that consist of multiple hidden layers, allowing them to learn complex patterns in data. By using GPGPUs, much larger models could be trained, which in turn increased the performance of the models. Since then, the industry has focused on increasing the performance of their hardware for deep neural networks, allowing us to train ever larger models.

The GPGPU implementation of ANNs is almost exclusively optimized for throughput. However, there are specific applications that require ultra-low latency inference of ANNs. An example of such an application is the trigger system at the CERN's Large Hadron Collider (LHC) [2]. The sensory system at the LHC generates data at a rate of terabytes per second. This presents a problem from a processing point of view, as it becomes impractical to store so much data or process it in real-time.

To mitigate this problem, a neural network is used to filter out the majority of data that does not exhibit interesting physical phenomena. Since the temporary storage buffers are limited, the neural network trigger system must be processed with a very low latency in the order of microseconds. These latency requirements necessitated the development of special hardware that can cope with this.

Other use cases for ultra-low latency hardware include real-time network intrusion detection [3], radio modulation classification [4], low latency optical tracking in a tokamak [5], and more [6].

Ultra-low latency hardware for accelerating neural network computation and the process of its development is the topic of this dissertation.

1.1 Hardware for Accelerating ANN Processing

ANN algorithms are highly parallelizable. This means that hardware can be designed to take advantage of the available parallelism and reduce the number of steps required to process ANNs compared to the steps required by a serial processor. In recent years, a wide range of hardware platforms have become available that take advantage of this fact and enable high speed processing of ANNs.

1.1.1 Throughput Optimized Hardware

Typically, ANNs are processed using either CPUs, GPGPUs or some kind of Tensor Processing Unit (TPU). Regardless of the type of processing unit used, they use an external memory to store ANN parameters and they are programmable. For convenience, let us refer to this group of processing units as Programmable Processing Units (PPU). The typical PPU is connected to an external memory via some kind of memory interface, e.g., DDR in the case of DRAM. This configuration is shown in Figure 1.1. This ubiquitous hardware

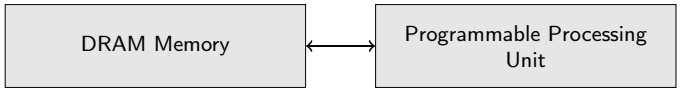


Figure 1.1: A typical hardware configuration.

configuration hinders the ability to use it for ultra-low latency inferencing. The reason for this is that every DRAM memory interface has a relatively high read latency [7]. Therefore, solutions that achieve ultra-low latency must avoid DRAM memory latency penalty and use only on-chip memory. Figure 1.2 illustrates the internal architecture of a typical PPU. They typically consist of cache memories and a number of Processing Units (PUs). Depending on the type of PPU, there are more or fewer PUs, which represent different types of hardware. For a CPU, a PU can be seen as parallel lanes of a single-instruction multiple data (SIMD) accelerator, for GPGPUs it can refer to a set of streaming multiprocessors, and for TPUs it could refer to a systolic array structure.

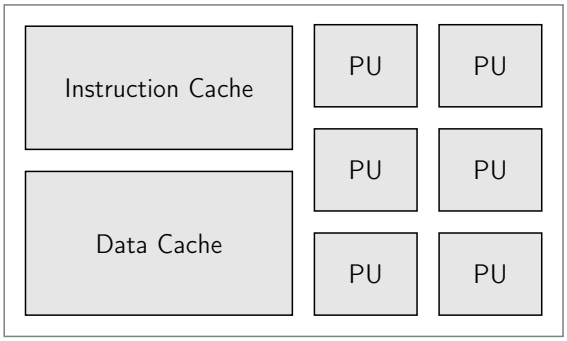


Figure 1.2: Internal architectures of PPUs.

The PPU approach is preferred in most applications because it is flexible and can be reprogrammed for different topologies by reprogramming the instruction and data memories. The PPU approach is also very scalable, as large external memories allow for very large model sizes.

1.1.2 Ultra-Low Latency Hardware

To achieve ultra-low latency, all data must be stored on-chip and a high level of parallelization is required. The Data Flow Processing Unit (DFPU) approach makes this possible by connecting a number of specialized PUs via parallel streaming interfaces. Each PU processes a subset of the neural network, typically a single layer. This is illustrated in Figure 1.3. The DFPU approach allows for layer-wise parallelism, where each PU can run in parallel on different inputs. In addition, the DFPU approach allows the PU to be specialized for the layer it processes. This significantly increases performance, but also binds

the structure of the design to the topology of the neural network. This means that such designs cannot be easily reused and are therefore best implemented on FPGA circuits that can be reprogrammed.

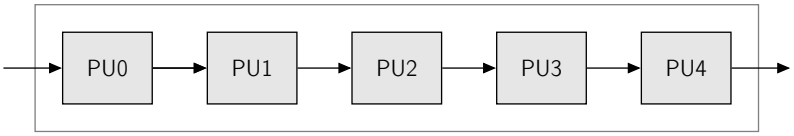


Figure 1.3: Block diagram of an hypothetical DFPU design.

In order to achieve the desired ultra-low latencies, the ANNs should be implemented in a highly parallel fashion. However, this presents a problem, as ANNs are usually processed in floating-point representation. The adders and multipliers for floating-point numbers are complex and can require a significant number of look-up tables to implement on an FPGA. The use of neural network optimization techniques such as quantization and pruning enables higher parallelism in ANN implementation. With these techniques, we can implement limited (but still useful) neural networks on FPGAs in a highly parallel fashion or even as a direct logic circuit. The latter is the most extreme case of parallelization, where the structure of the neural network is fully implemented as a combinational logic circuit.

This work focuses on DFPU approaches with a very high degree of parallelism. The goal is to design circuits that achieve ultra-low latency.

1.2 Hypothesis

The aim of this thesis is to develop a method for the automatic generation of highly parallel direct logic circuit implementations of deeply quantized neural networks using structural descriptions of the hardware. Previous work in this field uses high-level synthesis to implement ANNs. These solutions often do not produce optimal results. Furthermore, the size of the neural networks for which the hardware structure can be generated is limited due to the complexity of the algorithms used to infer hardware from algorithms. This has motivated us to develop a solution using hardware generators based on structural descriptions of ANNs.

We form three hypothesis for our work:

H1: Deeply quantized neural networks can be efficiently and automatically converted to highly parallel hardware implementations using generators based on structural descriptions of the hardware.

H2: The automatic conversion of deeply quantized neural networks using generators based on structural descriptions of the hardware can perform the conversion in significantly less time, then conversions based on algorithmic descriptions.

H3: The automatic conversion of deeply quantized neural networks using generators based on structural descriptions of the hardware can generate pipelined hardware circuits that can achieve very high throughput and ultra-low latency, when implemented on field-programmable gate array circuits.

1.3 Scientific Contributions

The main scientific contributions of this work are as follows:

SC1: A methodology for implementing highly parallel hardware implementations of quantized neural networks suitable for ultra-low latency inference using hardware generators that use structural descriptions.

SC2: An implementation of an intermediate representation for representing a wide range of quantized neural networks. This representation is the basis on which hardware generators operate.

SC3: An open source chisel4ml tool that utilizes the presented methodology and generate direct logic hardware implementations.

SC4: An in-depth analysis of chisel4ml tool and comparisons with other tools based on high-level synthesis technology that also generate highly parallel hardware implementations of quantized neural networks.

1.4 Thesis Structure

The thesis begins with an introductory chapter that provides a high-level overview of the problem. This is followed by two chapters that provide the background knowledge required to understand the thesis.

The first of these gives an introduction to artificial neural networks, especially from the computational point of view. It also discusses the process of optimizing the neural network for the DFPU approach using quantization and pruning. The practical aspects of training such quantized and pruned neural networks are also discussed.

The second background chapter focuses on hardware. This includes a brief introduction to field-programmable gate array circuits and an introduction to the languages used to describe the circuits of interest in this thesis. A basic introduction to high-level synthesis techniques is provided, as well as an introduction to the Chisel language, which is used to describe hardware generators.

Chapter 4 describes two state of the art solutions in the field: hls4ml and FINN. Both solutions are based on high-level synthesis technologies, but have some differences that we examine.

Chapter 5 describes our solution chisel4ml, which is based on Chisel Generators. The chapter discusses in detail how chisel4ml uses Chisel to efficiently generate neural networks in hardware.

Chapter 6 provides a wide range of comparisons of fully parallel hardware structures generated by chisel4ml, hls4ml, and FINN tools.

Finally, Chapter 7 concludes the thesis.

Chapter 2

Artificial Neural Networks

Artificial Neural Networks (ANNs) are a class of machine learning algorithms loosely based on the behavior of biological neurons. They are commonly used in a variety of applications and can be used to train classification and regression models.

ANNs are constructed from a series of sub-functions called layers. In general, a layer consists of a set of neurons, where the inputs of the neurons are connected to the inputs of the layer and the outputs of the neurons represent the output of the layer. Not all layers consist of neurons. Examples of such layers are the pooling layer and the batch normalization layer.

2.1 The Neuron Model

A single neuron takes a series of inputs and produces a single output. The inputs of the neuron are also called synapses. The neuron operates by first summing weighted inputs and a bias. This sum is called the pre-activation and is passed to a non-linear activation function, which produces the neuron's output, also called the activation value. A non-linear activation function is required to enable the modeling of non-linear relationships [8]. The weights and the bias are constant parameters during inference and are determined during the training of the neural network. The above description is concisely expressed in Equation 2.1:

$$y = f(b + \sum_{i=1}^{M} w_i \cdot x_i) = f(\vec{w} \cdot \vec{x} + b), \tag{2.1}$$

where y is a scalar output or activation value, b is the scalar bias value, $\vec{w}$ and $\vec{x}$ are vectors of size M that represent the weight parameters and inputs, respectively. Finally, f is a non-linear activation function. Two examples of activation functions are the rectified linear unit and the sigmoid function. They are given in the following equations, and their graphs are shown in Figure 2.1.

$$f(x) = ReLU(x) \quad = \begin{cases} x & if\, x > 0, \\ 0 & otherwise. \end{cases}$$

$$f(x) = sigmoid(x) \quad = \frac{1}{1 + e^{-x}}$$

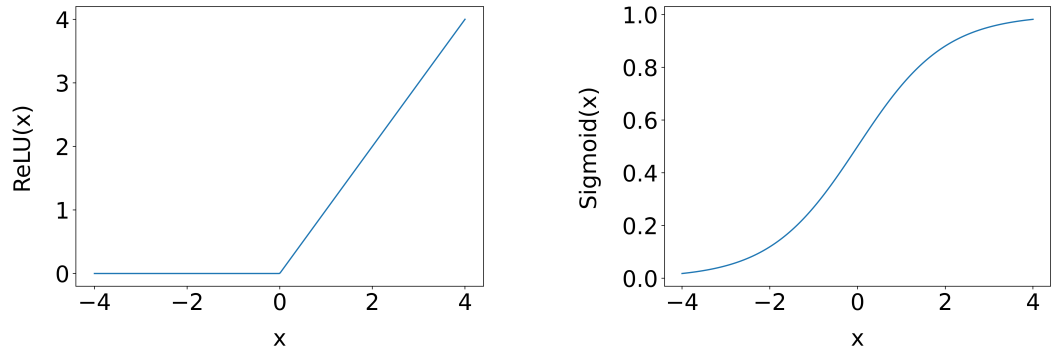


Figure 2.1: A rectified linear unit (ReLU) activation function on the left and the sigmoid activation function on the right.

2.2 Neural Network Layers

A single neuron alone is not very useful as a machine learning model. To get useful structures, many neurons are combined into larger blocks called layers. These layers differ mainly in how the inputs of the layer are connected to the neurons in the layer. Note that not all layers use neurons. In general, a layer can be defined as a function that has one or more tensors as input and produces one or more tensors as output.

Neural network layers can also be viewed as some groupings of operations in a computational graph. They are not meant to be atomic operations, but rather a convenience for people to understand the structure of neural networks. Layers therefore consist of mathematical operations such as matrix multiplication, addition, and activation functions.

2.2.1 Fully Connected Layer

A basic type of neural network layer is a fully connected or dense layer. The fully connected layer connects all inputs of the layer to every neuron in the layer. This is illustrated in Figure 2.2. A fully connected layer with N inputs and M outputs can be expressed as a matrix-vector product as shown in Equation 2.2:

$$\vec{y} = f(W \cdot \vec{x} + \vec{b}), \tag{2.2}$$

where W is an $M \times N$ weight parameter matrix, $\vec{x}$ is an N long input vector, $\vec{y}$ and $\vec{b}$ are the output and bias vectors, both of length M. f is a non-linear activation function that is applied to each pre-activation separately.

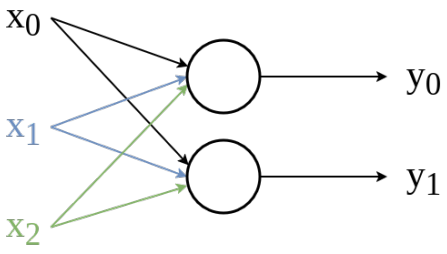


Figure 2.2: A fully connected layer with two neurons and three inputs

2.2.2 Convolution Layer

Another common neural network layer is a convolution layer. The basic operation is the convolution of input features with a predefined kernel, which is usually relatively small compared to the size of the input feature set. If the input features represent multidimensional data, both the kernel and the convolution operation are also multidimensional. Image processing is a domain where multidimensional data is used. Although 2D color images can be represented by 3D input data, it is often advantageous to separate 2D images by color, i.e., by channel. Usually, there are three channels (RGB), but any number of channels can be used, e.g., when using a multispectral camera. In contrast to the fully connected layer, where each neuron operates on all inputs, the convolution layer assigns a local receptive field to each neuron. This means that each individual neuron operates on a small portion of the input tensor. Therefore, the convolution layers use fewer parameters for a given input size than fully connected layers and are well suited to the task of image recognition [9]. The operation of a two-dimensional convolution layer is described in Equation 2.3:

$$Y[k,i,j] = f(\vec{b}[k] + \sum_{c=1}^{C}\sum_{x=1}^{H_F}\sum_{y=1}^{W_F} X[c,i+x,j+y] \cdot F[k,c,x,y]), \tag{2.3}$$

where X is a three-dimensional input tensor of dimensions $[C, H_I, W_I]$, e.g., an RGB image has $C = 3$, the output Y is a three-dimensional tensor of dimensions $[K, H_O, W_O]$. The weight parameter tensor F has the dimensions $[K, C, H_F, W_F]$ and consists of a set of K filters that span all channels C of the input and a subset of the height and width of the input: $H_F \leq H_I$, $W_F \leq W_I$. The bias vector $\vec{b}$ has K elements, a scalar constant for each kernel. And the function f is a non-linear activation function. Note that, if $H_F = H_I$ and $W_F = W_I$ the convolution operation becomes equivalent to a fully connected layer, as each neuron is connected to the entire input.

Figure 2.3 shows a diagram of a very simple convolution operation with an input size of $[1, 4, 4]$, a filter of size $[1, 1, 2, 2]$, and an output size of $[1, 3, 3]$. Equation 2.3 is only the simplest form of the convolution layer operation. Apart from the number of channels and the filter dimensions, several additional parameters are usually used for the convolution layer:

- stride - the size of the step between the receptive fields of the neurons (in the above equation, the value one is assumed),
- padding - adds additional zeros (or other values) around the image to increase the input size (in the above equation, the value zero is assumed),
- groups - controls the connections between inputs and outputs by selecting which channels are visible to which kernel.

Equation 2.3 assumes that there is only one group of channels, so that each kernel operates on all channels. The other extreme is when the group setting is equal to the number of input channels C, so that each input channel is convolved with its own set of filters. This is also known as depthwise convolution and is often used to minimize the memory footprint of a convolution layer [10].

In the remainder of the thesis, convolution layer will be understand to mean the two-dimensional convolution layer.

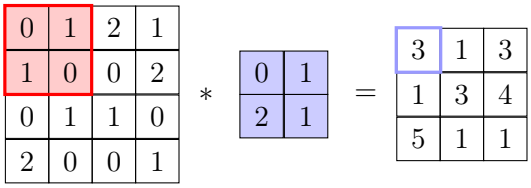


Figure 2.3: A simple convolution operation.

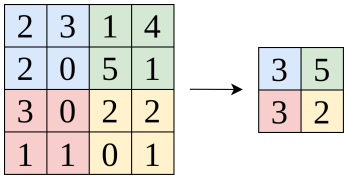


Figure 2.4: A simple maximum pooling operation.

2.2.3 Pooling Layer

Pooling layers are layers that do not use neurons to perform computations. Instead, as the name implies, they perform different types of pooling operations. Two commonly used examples are the two-dimensional average pooling layer and the two-dimensional maximum pooling layer. Similar to the two-dimensional convolution layer, the pooling layer also operates on local receptive fields of a three-dimensional input tensor. Instead of calculating the neuron model according to Equation 2.1, the average and maximum pooling layers calculate either the average of the receptive field or its maximum. Figure 2.4 shows how a maximum pooling layer operates on an input of size $[1, 4, 4]$ and a kernel of size $[1, 2, 2]$. The result is an output tensor of size $[1, 2, 2]$. The colors in the input tensor highlight the active local receptive field of the corresponding output of the output tensor. For pooling layers, the stride is typically the size of the receptive field, which equals two in the given example. This differs from convolution layers, which typically have a stride of one. This is due to the fact that these layers are typically used to downsample the input tensors.

In the remainder of the thesis, maximum pooling layer is assumed to be two-dimensional maximum pooling layer.

2.2.4 Batch-Normalization Layer

Batch-normalization is a normalization technique that accelerates the learning process. The authors of [11] suggest that batch normalization layers prevent a phenomenon termed *internal covariate shift*. This is the case when the outputs of the previous layers keep shifting during training, which means that the subsequent layers must keep adapting to the new outputs.

To efficiently compute ANN on GPGPUs, batching is used to reuse data. In effect this gives each tensor in an ANN an extra batch dimension N. For an input tensor X to the batch normalization layer with the dimensions $[N, M]$, an output tensor Y with the same dimensions is calculated. For each element of a non-batch dimension M, an average value is calculated along the batch dimension N. The mean value is then used to calculate the variance and subsequently to normalize the elements. Two learnable parameters *gamma* and *beta* are then introduced for each element, which control the scaling and shifting of the distribution. This is shown in Algorithm 2.1. Here, x_i represent the individual input

elements and y_i the outputs. This algorithm is therefore repeated M times to compute the entire output tensor Y.

Algorithm 2.1: The Batch Normalization procedure [11].

Input : Values of x over a mini-batch: x_i; $i = 1..N$
Parameters to be learned: γ, β
Output: Values y over a mini-batch: y_i; $i = 1..N$

(1) $\mu \leftarrow \frac{1}{N}\sum_{i=1}^{N} x_i$ // mini-batch mean
(2) $\sigma^2 \leftarrow \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$ // mini-batch variance
(3) $\hat{x}_i \leftarrow \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$ // normalize
(4) $y_i \leftarrow \gamma \cdot \hat{x}_i + \beta$ // scale and shift

During training, a running mean and variance are calculated so that after training, the mean and variance are set to a constant that represent the mean and variance of the entire training dataset. In other words, steps 1 and 2 are omitted at inference.

Since the batch normalization layer is a linear transformation, it can be merged with the surrounding layers as long as there is no non-linear transformation in between. The process of folding batch normalization layers is described in Section 2.3.2.

2.3 Feed-Forward Neural Networks

In the thesis only stateless layers are considered. That is, layers whose output depends solely on the input and not on the internal state of the network. Neural networks that consist only of stateless layers are termed feed-forward neural networks. A feed-forward neural network consists of a set of n layers $g_i(X_i, \theta_i)$ that successively perform a transformation of their input tensors X_i using their parameters θ_i to produce the output tensor X_{i+1} and finally the output of the neural network Y. Not all layers necessarily have parameters.

$$Y = G(X_1, \theta) = g_n(\ldots g_2(g_1(X_1, \theta_1), \theta_2), \theta_n) = g_n(\theta_n) \circ \ldots \circ g_2(\theta_2) \circ g_1(X_1, \theta_1).$$

The ordering and configuration of the layers, e.g., the kernel size, determine the architecture or topology of the neural network. The performance of a neural network is influenced by the topology of the neural network, but also by many other factors, such as the way it is trained and the quality and quantity of the training data.

Models that consist only of fully connected layers are called multilayer perceptrons (MLPs), and models that contain convolution layers are called convolutional neural networks (CNNs).

2.3.1 Neural Network Training

Training an ANN is the process of determining the weight parameters θ to perform a certain task using a neural network $G(X, \theta)$. For this process, a loss function is defined that measures the difference between ground truth and ANN output values. Training can be expressed as an optimization problem that minimizes a loss function for a given set of parameters. Almost exclusively, gradient based methods are used for training ANNs. Gradients are used to correct the weight parameters θ in order to minimize the loss. The

training of a neural network can be defined as the following minimization problem:

$$min \sum_{i=1}^{|T|} L(\hat{Y}_i, G(X_i, \theta)). \tag{2.4}$$

Where L is the loss function and T is the training dataset containing a set of inputs X_i and the ground truth values $\hat{Y}_i$. The cardinality of T is typically too large to compute all at once. For this reason, the loss function is computed in so-called mini-batches, that contain a random subset of T. The loss function depends on the type of problem to be solved. For example, regression problems typically use a mean squared error loss function, while the cross-entropy loss function is more suitable for classification problems.

The minimization of the loss function is performed iteratively using the gradient descent method:

$$\theta_t \leftarrow \theta_{t-1} - \eta \frac{\partial L}{\partial \theta_{t-1}}$$

where η is the step size, θ_{t-1} are the weight parameters from the previous iteration, and θ_t are the updated weight parameters. Note that actual optimization algorithms can be somewhat more complicated [12]. In the process of neural network training $G(X_i, \theta)$ and L are calculated first and then the gradients $\frac{\partial L}{\partial \theta}$. The former is called the forward pass and the latter the backward pass.

2.3.2 Folding Batch Normalization Layers

As discussed in Section 2.2.4, batch normalization layers are important for training deep neural networks. However, they are also difficult to quantize. Fortunately, they are typically inserted before activation functions and can therefore be folded into preceding layers [13].

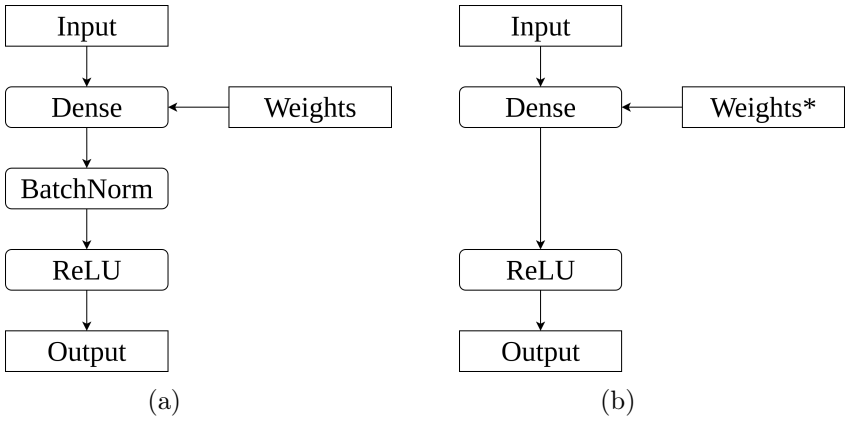


Figure 2.5: A graph of a fully connected layer followed by a batch normalization layer in (a) and the same graph with the batch normalization layer folded into the $Weights$ in (b).

The typical arrangement of layers with batch normalization is shown in Figure 2.5a. The batch normalization layer is usually inserted before an activation function in fully connected or convolution layers. The computational graph in Figure 2.5 can be written

with the following equations:

$$\begin{aligned}\vec{p} &= W \cdot \vec{x} + \vec{b} \\ \vec{p}_{mod} &= \gamma \cdot \frac{\vec{p} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \\ Output &= ReLU(\vec{p}_{mod}) \\ Output &= ReLU(\gamma \cdot \frac{W \cdot \vec{x} + \vec{b} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta)\end{aligned}$$

The above equation can be rewritten by combining the parameters of the batch normalization layer into the parameters W and b and replacing them with W^* and $\vec{b}^*$:

$$\begin{aligned}\vec{b}^* &= \gamma \cdot \frac{\vec{b} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \\ W^* &= \gamma \cdot \frac{W}{\sqrt{\sigma^2 + \epsilon}} \\ Output &= ReLU(W^* \cdot \vec{x} + \vec{b}^*).\end{aligned}$$

In practice, a model with batch normalization layers is trained first, then the batch normalization layers are folded into the preceding fully connected or convolution layer. However, since practical implementations work with finite numbers, the transformed model is not exactly equivalent, which often leads to a deterioration in accuracy. This can be mitigated by training the folded model for a few additional epochs.

2.4 Quantization

One of the implicit assumptions of gradient-based optimization is that the parameters θ of a neural network G are real numbers. However, a digital computer cannot store or compute real numbers. Usually, the parameters θ are stored as 32 bit floating-point numbers according to the IEEE 754 standard [14]. The 32 bit floating-point number representation is effective, however, it comes at the price of high memory consumption and complex arithmetic circuits. This motivated the use of various lower-precision floating-point and fixed-point formats.

2.4.1 Basics of Quantization

Quantization is the process of subdividing a quantity into a discrete set of values. In general, we are interested in quantizing from sets with larger cardinality to sets with smaller cardinality. Therefore, a generic quantization function or quantizer q can be defined as follows:

$$q : A \to B; \quad |A| \geq |B|$$

where A and B are two sets and the cardinality of A is greater than or equal to the cardinality of B. Sets of equal cardinality could also be interesting, as computing in 32 bit integers can be more efficient than computing in 32 bit floating-point numbers[1].

For convenience let us define $\mathbb{F}_{32}$ as the set of 32 bit floating-point numbers, $\mathbb{S}_n$ as the set of all signed n bit integers, $\mathbb{U}_n$ as the set of all unsigned n bit integers, and $\mathbb{I}_n$ as the set of n bit integers that do not have signedness specified. In addition, let us define subsets

[1] Strictly speaking, the set of 32 bit integers is larger than the set of 32 bit floating-point numbers, since the floating-point set contains two representations of zero, a not-a-number bit combination, etc.

in the following form: $\mathbb{F}_{32}^{[\alpha,\beta]}$, which denotes all floating-point numbers in the range $[\alpha, \beta]$. In neural network quantization the set A is typically the set of floating-point numbers $\mathbb{F}_{32}$ and B is some smaller set of integers, for example the set of 8 bit integers $\mathbb{I}_8$.

A quantization function is said to be uniform if the resulting quantized values are spaced equally. A uniform quantization function is defined by:

$$q_u(x) = \left\lfloor \frac{x}{S} \right\rceil - Z \tag{2.5}$$

where x is a value in some finite continuous range, S is a real-valued scale factor and Z is an integer that defines the zero-point. The notation $\lfloor x \rceil$ signifies the rounding of x to an integer. The rounding function is intentionally not precisely defined, as there are several options.

A quantization function is symmetric if $q(-x) = -q(x)$ holds for every x of the input range, otherwise the quantization function is said to be asymmetric.

The quantization function in Equation 2.5 is dynamic if the values of S and Z depend on the input x. If these are constants, the quantization is said to be static. Dynamic quantization can potentially result in less information loss, but also negates some of the performance benefits as the values of S and Z must be dynamically recalculated.

Figure 2.6 shows the identity function, which corresponds to no quantization, and two types of symmetric quantization functions. The middle plot shows a symmetric uniform quantization of the function $y(x) = x$ for $\beta = -\alpha = 1$, which is the most commonly used type of quantization.

In this thesis, only static symmetric uniform quantization schemes are considered. Uniform quantization is particularly interesting because general-purpose digital hardware can be used directly to perform computations with uniformly quantized numbers. In most cases, special hardware is required to efficiently compute mathematical operations on non-uniformly quantized values. However, non-uniformly quantized values could preserve more information of a given signal, which means that fewer bits are required to store each parameter. The power of two quantization shown in the right plot of Figure 2.6 is an example of non-uniform quantization [15].

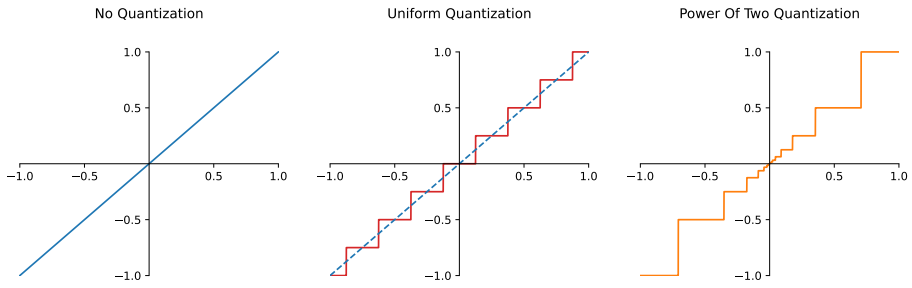


Figure 2.6: Plot of the function $y = x$ without quantization, with uniform quantization and with power of two quantization (non-uniform).

Fake quantization functions are similar to quantization functions, but do not change the data type. A fake quantization function $\hat{q}$ is defined by:

$$\hat{q} : \mathbb{F}_{32}^{[\alpha,\beta]} \to \mathbb{Q}_n^{[\hat{\alpha},\hat{\beta}]};$$
$$\mathbb{Q}_n^{[\hat{\alpha},\hat{\beta}]} \subseteq \mathbb{F}_{32}^{[\hat{\alpha},\hat{\beta}]} \wedge |\mathbb{Q}_n^{[\hat{\alpha},\hat{\beta}]}| = |\mathbb{I}_n|.$$

Here a new set $\mathbb{Q}_n^{[\hat{\alpha},\hat{\beta}]}$ is defined. It has the same cardinality as some set $\mathbb{I}_n$ and is a subset of $\mathbb{F}_{32}^{[\hat{\alpha},\hat{\beta}]}$. In the context of uniform quantization, $\mathbb{Q}_n^{[\hat{\alpha},\hat{\beta}]}$ is a set of 2^n approximately equidistant floating-point numbers in the interval $[\hat{\alpha}, \hat{\beta}]$. Note that $\alpha \approx \hat{\alpha}$ and $\beta \approx \hat{\beta}$. They differ because of the possibility of rounding at the edge of the interval. An example of a symmetric uniform fake quantization function is given by:

$$\hat{q}_u(x) = S \cdot \left\lfloor \frac{x}{S} \right\rceil. \tag{2.6}$$

It is typical to multiply the function by the scale factor so that the quantized variables have similar values to unquantized variables. Fake quantization functions are used in the training of quantized neural networks. This is because they are easy to implement in today's widely used frameworks for training neural networks that are based on floating-point numbers.

2.4.2 Quantizing Neural Networks

Traditional quantization techniques from digital signal processing are trying to solve the problem of minimizing the reconstruction error. The quantization problem in neural networks is different because the main performance criterion is the performance metric of our model (e.g., accuracy) rather than the actual reconstruction error of quantizing the parameters from floating-point to another format. In other words, the goal is to find neural networks with low-bitwidth parameters that are easy to compute and can still minimize the loss function defined by Equation 2.4. It is possible that quantizing a neural network increases its performance metrics, but in general it degrades them. There are two main methods to solve this problem: post-training quantization and quantization-aware training.

2.4.2.1 Post-Training Quantization

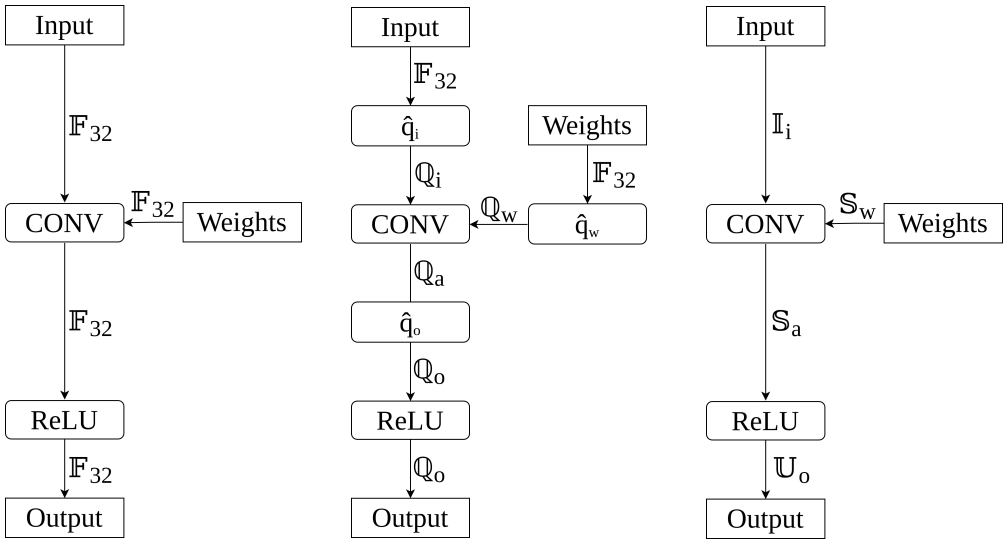
Post-Training Quantization (PTQ) takes pre-trained floating-point models and outputs quantized models. Sometimes these approaches also require a minimal amount of training data for calibration purposes. This method can directly use floating-point models trained in standard environments and convert these models into quantized models. Such frameworks usually target 8 bit quantization [16], but have recently been able to improve results down to 4 bits [17]. In general, the success of PTQ approaches depends on the network topology and the dataset used. Neural networks quantized with PTQ methods often suffer from poor performance when quantized with less than 8 bits.

2.4.2.2 Quantization-Aware Training

Models whose weight parameters and/or activations are quantized bellow 8 bits are referred to as Deeply Quantized Neural Networks (DQNNs). A different quantization technique called Quantization-Aware Training (QAT) is usually used to create DQNN models. In QAT methods, fake quantization functions are inserted into the computational graph of the neural network. During the forward pass of the training, these functions constrain the floating-point values to a set of fixed-point values. However, during the backward pass, when the derivatives are computed, such quantization functions present a problem since they are not continuous. This problem is solved by a version of the Straight-Through Estimator (STE) method that simply ignores the derivative contributions of the fake quantization functions and truncates the gradients if they are greater than one [18].

QAT can successfully produce very low-bitwidth models and even models with binary weight parameters and activations [18]. Figure 2.7 shows three computational graphs of a

convolution layer. The left figure shows the computational graph of a normal floating-point convolution layer. The middle figure shows the computational graph of a convolution layer during QAT and the right figure shows a quantized computational graph of a convolution layer that can be obtained after QAT. The fake quantization functions $\hat{q_i}$, $\hat{q_w}$, and $\hat{q_o}$ quantize the input, weight constants and outputs. The indices of the fake quantization functions indicate the bitwidth used to represent the given tensor values.



(a) Floating-point graph. (b) QAT graph. (c) Quantized graph.

Figure 2.7: Computational graphs of a convolution layer.

2.4.2.3 Scale Factors

A trained neural network G with n layers and parameters $\theta = \{\vartheta |\ \vartheta \in \mathbb{F}^{[\alpha,\beta]}\}$ is a composition of functions (layers) g_i. The parameters of layer i are denoted as θ_i such that $\theta = \theta_1 \cup \theta_2 \cup \cdots \cup \theta_n$. One approach to quantize the network G would be to simply transform all learned weight parameters with a single quantization function. By setting $\alpha = min\,\theta$ and $\beta = max\,\theta$ in the uniform quantizer from Equation 2.5, a function G', which is an approximation of G, is obtained. However, such an approach leads to poor performance at lower bitwidths because the parameter ranges vary greatly throughout the neural network, especially between layers [16]. Instead, the different chunk of θ can be quantized independently by selecting the bitwidth of each chunk separately and introducing a scale factor S. For example, if the tensor θ_i is considered as a chunk for the case of symmetric uniform quantization, the following equation is obtained:

$$\theta_i \approx S \cdot \hat{\theta}_i \tag{2.7}$$

where $\hat{\theta}_i$ is the quantized variant of θ_i. The selection of these chunks determines the *granularity* of the quantization. For instance, per-layer quantization quantizes each θ_i independently. A common approach is per-tensor quantization, where each tensor is quantized independently. In the case of the fully connected layer described in Equation 2.2, this means that W and $\vec{b}$ are quantized separately. You could go even further and do per-channel quantization or use even smaller groups of parameters. There is, however, a trade-off to consider. Because if different groups of parameters are quantized on different

ranges, they need to be scaled in order to compute with other groups of parameters. This can lead to additional computational and memory requirements on the deployment system.

The use of scale factors is crucial to obtain high-accuracy quantized neural network models. The scale factor can also be determined as part of the training process [19]. If the scale factor is limited to a power of two, the scaling can be efficiently implemented in hardware with a simple shift operation. By combining Equations 2.7 and 2.1, the following equation is obtained:

$$y = f(\vec{w} \cdot \vec{x} + b) \approx f(S \cdot \hat{\vec{w}} \cdot \vec{x} + b) \tag{2.8}$$

This can be calculated using only efficient integer operations if $\hat{\vec{w}}$, $\vec{x}$, b are all integers and S is a constant power of two. Neural networks that are quantized in this way are also called dyadic neural networks [20].

2.4.3 Binarized Neural Networks

Binarized neural networks (BNNs) represent the extreme case of quantization, where both weights and activations are restricted to the set $\{-1, +1\}$ [18], which can be represented with a single bit. Nevertheless, BNNs can still achieve acceptable accuracies, at least for simpler tasks. The authors of [18] have shown that for datasets such as the CIFAR-10 dataset [21], BNNs can achieve an accuracy that is less than 1 % below the accuracy of a floating-point neural network. However, for more complex datasets, the difference tends to grow. Note that BNNs do not binarize the inputs to the neural networks in the first layer, as this would potentially discard a large amount of information.

The activation function in BNN is the sign function:

$$f(x) = Sign(x) = \begin{cases} +1 & x \geq 0, \\ -1 & otherwise. \end{cases} \tag{2.9}$$

An efficient implementation of BNNs encodes the values -1 and +1 into the binary values 0 and 1, as shown in Figure 2.8. An encoded element v^* that has a value in the

$\mathbf{w}$	$\mathbf{x}$	$\mathbf{w} \cdot \mathbf{x}$
-1	-1	1
-1	1	-1
1	-1	-1
1	1	1

$\mathbf{w}^*$	$\mathbf{x}^*$	$\mathbf{w}^* \odot \mathbf{x}^*$
0	0	1
0	1	0
1	0	0
1	1	1

Figure 2.8: Efficient binarized values encoding.

set $\{0, 1\}$ can be transformed into an element v that has a value in the set $\{-1, +1\}$:

$$v = 2v^* - 1.$$

With this encoding, the multiplication operation is converted into a simple XNOR operation and the vector addition operation is converted into a population count operation[2]. The above statements can be verified by inserting the sign activation function into Equation 2.1, which represents the neuron operation:

$$y = f(b + \sum_{i=1}^{M} w_i \cdot x_i) = \begin{cases} +1 & \sum_{i=1}^{M} w_i \cdot x_i + b \geq 0, \\ -1 & otherwise. \end{cases}$$

[2]A population count operation counts the number of ones in a binary vector.

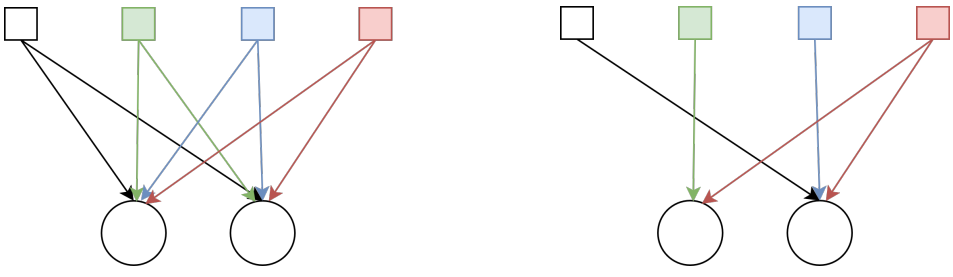
Using the above equations, the following inequality must hold:

$$\begin{aligned}
\sum_{i=1}^{M} w_i \cdot x_i + b &\geq 0 \\
\sum_{i=1}^{M} p_i + b &\geq 0 \\
\sum_{i=1}^{M} (2p_i^* - 1) + b &\geq 0 \\
2 \cdot \sum_{i=1}^{M} p_i^* &\geq M - b \\
\sum_{i=1}^{M} w_i^* \odot x_i^* &\geq \frac{M-b}{2} \\
PopCount(\vec{w}^* \odot \vec{x}^*) &\geq \frac{M-b}{2} \\
PopCount(\vec{w}^* \odot \vec{x}^*) &\geq T
\end{aligned} \tag{2.10}$$

Where $\odot$ signifies the XNOR operation, p and p^* represents the normal and encoded product, and w^* and x^* represent the encoded weight and input vectors. Since both b and M are constants, a threshold T can be computed in advance. Thus, the neuron computation is performed using only bitwise XNOR operation, a population count operation, and a comparison operation.

2.5 Pruning

Pruning is a technique for reducing the computational complexity of ANNs by removing insignificant synapses from the neural network. This is achieved by setting small weight parameters to zero. Since most ANNs are highly over-parameterized, pruning can in some cases remove up to 90% of the synapses and still retain good accuracy of the neural network [22]. The pruning process is illustrated in Figure 2.9 using an example of a fully connected layer. The left figure shows an unpruned fully connected layer and the right figure shows the layer after pruning.



(a) Without pruning. (b) With pruning.

Figure 2.9: A fully connected layer with four inputs and two neurons.

Pruning can be categorized into: unstructured and structured pruning. Unstructured pruning is not limited by any constraint. Any connection or weight parameter can be

pruned. Structured pruning, on the other hand, attempts to prune structures in a certain way. For example, structured pruning attempts to remove an entire convolution kernel. Unstructured pruning typically achieves higher prune rates than structured pruning. However, unstructured pruning results in the weight parameters being stored in sparse matrices. This makes acceleration more difficult on certain compute platforms such as GPGPUs. On the other hand, the direct logic circuit implementation is able to take full advantage of unstructured pruning. The size of a direct logic circuit implementation can be reduced by omitting the hardware part corresponding to the pruned connections.

The inefficiency of parallel accelerators on unstructured sparse matrices can be mitigated by semi-structured sparsity and block sparsity techniques [23]. These techniques attempt to modify the parameter matrix of the neural network in such a way that parallel accelerators such as GPGPUs can compute the neural network efficiently. However, this is beyond the scope of this work since this work focuses on the direct logic circuit implementations.

A simple pruning procedure is to remove a certain percentage of the smallest weight parameters. When this is done for a single tensor or layer, it is called local pruning. Global pruning, on the other hand, involves removing a certain percentage of the smallest weight parameters from the entire neural network. This means that some layers can be pruned significantly more than others. The pruning technique is independent of the quantization technique and they can be combined.

2.6 DQNN Training Frameworks and Graph Representations of Neural Networks

Modern frameworks for training neural networks offer simple interfaces in Python that allow neural network developers to define the topology of the neural network and the training parameters. The two most commonly used frameworks today are PyTorch [24] and Tensorflow [25]. However, these frameworks do not have native support for quantization to an arbitrary bitwidth. Various extensions have been added to these frameworks to enable arbitrary bitwidth quantization. This functionality is provided by a library called Brevitas [26] in the PyTorch ecosystem and in Tensorflow by the QKeras library [27]. Both Brevitas and QKeras provide implementations of fake quantization functions, allowing developers to perform quantization-aware training of neural networks with arbitrary bitwidths.

Originally, both Brevitas and QKeras had a custom serialization format. However, to enable interoperability, a new format QONNX: Quantized Open Neural Network eXchange was developed [28], [29]. QONNX is an extension of the Open Neural Network eXchange (ONNX) representation that aims to standardize the general ANN representation [30].

2.6.1 Open Neural Network eXchange Format

The Open Neural Network eXchange format represents neural networks in a form of a graph, where the nodes represent computational operations and the links represent tensors. ONNX defines a wide range of computational operators, including:

- a matrix-matrix product (Gemm),
- a convolution operation (Conv),
- a maximum pooling operation (MaxPool),
- a matrix addition operation (Add),

- activation functions (e.g., ReLU),
- data processing operations (e.g., transpose tensor).

The above operators do not have to correspond exactly to the definition of layers from Section 2.2. Instead, layers in these representations can be considered as subgraphs. For example, a fully connected layer as defined in Section 2.2 could be considered as a subgraph consisting of a matrix-matrix product node and an activation function operation node. An ONNX graph of a fully connected layer is shown in Figure 2.10. An ONNX graph of a

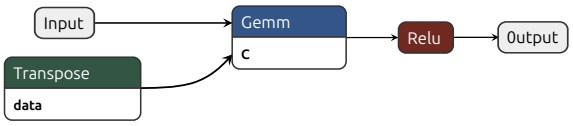


Figure 2.10: An ONNX graph of a fully connected layer.

simple convolutional neural network is shown in Figure 2.11.

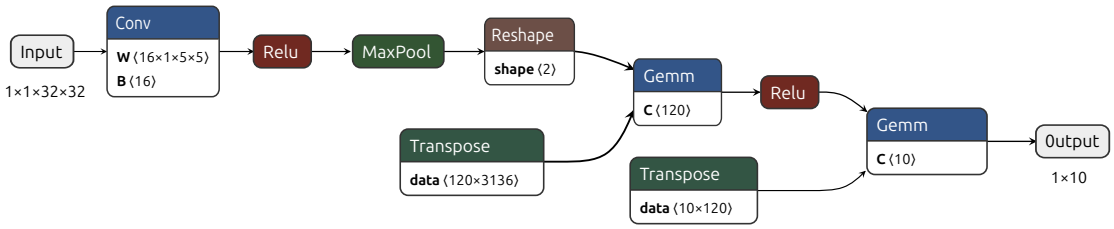


Figure 2.11: An ONNX graph representation of a simple CNN.

2.6.2 Quantized Open Neural Network eXchange Format

Quantized Neural Network eXchange (QONNX) is an extension of the ONNX standard, [28]. QONNX adds a standard way to represent quantization functions by adding custom ONNX operators. These operators allow QONNX to represent neural networks quantized to an arbitrary bitwidth. Figure 2.12 depicts a single convolution operation with an input tensor quantized to 2 bits, weights quantized to 4 bits, bias quantized to 8 bits, and an output tensor quantized to 4 bits.

QONNX defines several additional custom operations:

- **Quant**
  The Quant operator is a fake quantization function that performs uniform quantization operations, [28]. It is defined similarly to Equation 2.6, with the difference that it adds clamping. The purpose of clamping is to limit the maximum and minimum values that the output of the function can take. The following equation describes the behavior of the QONNX Quant operation:

$$Y = Quant(X) = \begin{cases} S \cdot (y_{min} - Z) & if \left\lfloor \frac{X}{S} + Z \right\rceil \leq y_{min}, \\ S \cdot (y_{max} - Z) & if \left\lfloor \frac{X}{S} + Z \right\rceil \geq y_{max}, \\ S \cdot (\left\lfloor \frac{X}{S} + Z \right\rceil - Z) & otherwise, \end{cases}$$

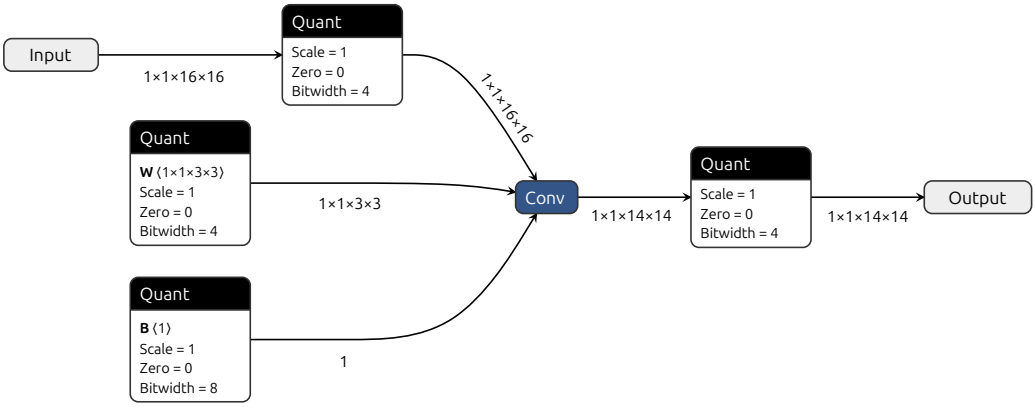


Figure 2.12: A quantized convolution layer.

where X is the input tensor, S is the scale factor, Z is the zero-point, Y is the output tensor, and $y_{\min}/y_{\max}$ represent the maximum and minimum values that can be represented by a certain quantization bitwidth. Equations bellow show $y_{\min}$ and $y_{\max}$ for a certain quantization bitwidth k, depending on whether signed or unsigned numbers are used.

$$y_{\min} = \begin{cases} -2^{k-1} & \textit{if signed,} \\ 0 & \textit{otherwise.} \end{cases}$$

$$y_{\max} = \begin{cases} 2^{k-1} - 1 & \textit{if signed,} \\ 2^{k} - 1 & \textit{otherwise.} \end{cases}$$

Note that in newer versions of QONNX the Quant operator has been renamed to IntQuant.

- **BipolarQuant**
  The BipolarQuant operator is an operator used to implement binary quantization [28]. It implements a scaled sign operation:

$$\mathit{BipolarQuant}(X) = \mathit{Sign}(X) \cdot S$$

  Where the $\mathit{Sign}(X)$ function is the function defined in Equation 2.9, and S is the scale factor.

- **Trunc**
  The Trunc operator truncates the least significant bits of a quantized value [28]. This work does not use the Trunc operator, as it is possible to represent the neural networks of interest without it.

2.6.3 Tensor Layouts

The layout of a tensor in memory is a mapping of a multidimensional array into a linear memory. The tensor layout can more generally be thought of as a linearization of a multi-dimensional array. In this way, it can also be used in other contexts, e.g., for the order of data on a streaming bus interface. The layout of the tensor is an important consideration as it has a large impact on performance and most computational kernels are written with only a single layout in mind.

A layout can be viewed as a mapping from a multidimensional space to a one-dimensional space. For a D-dimensional tensor, a layout is a function f that has D inputs and produces a single value:

$$f(x_1, x_2, \ldots, x_D) = x_1 \cdot stride_1 + x_2 \cdot stride_2 + \cdots + x_D \cdot stride_D,$$

where f represents the memory location of the tensor value and x_i represents the i-th index of the tensor value.

A widely used multidimensional tensor in ANN is the color 2D image with RGB or RGBA channels (red, green, blue, and alpha). There are two common layouts for such tensors:

- The HWC layout, where channel dimension (C) is iterated first, following the width dimension (W) is iterated, and finally, the height dimension (H) is iterated.
- The CHW layout, where the weight dimension is iterated first, followed by the height dimension (H), and finally, the channel dimension (C) is iterated.

In neural networks, the notation NHWC and NCHW is more commonly used, where N indicates the batch dimension. As discussed previously, the batching technique accelerates computation, especially on GPGPU accelerators.

Each channel (red, green, blue, and alpha) in 2D image is usually represented by one byte, e.g., pure red color is #FF0000FF, pure yellow color is #FFFF00FF, where # denotes hexadecimal value. Let us consider small image with dimensions 2x3 shown in Figure 2.13.

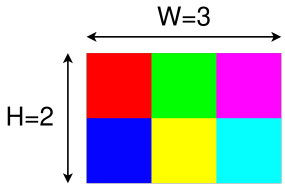


Figure 2.13: A 2 by 3 RGBA image, where alpha equals 1 for all pixels.

The 32 bit addressable HWC layout of this image is given in Figure 2.14a, where each pixel is represented by one line. The CHW layout of the same image is given in Figure 2.14b



Figure 2.14: RGBA image stored in a 32 bit byte-addressable memory using (a) HWC layout or (b) CHW layout.

The HWC layout mapping for byte-addressable memory is:

$$f_{HWC}(h, w, c) = h \cdot 12 + w \cdot 4 + c,$$

where h, w, and c are the indices of the tensor's height, width, and channel dimension, respectively.

Similarly, the CHW layout mapping is:

$$f_{CHW}(h, w, c) = c \cdot 6 + h \cdot 3 + w.$$

Chapter 3

Hardware

3.1 Field-Programmable Gate Array

A Field-Programmable Gate Array (FPGA) is a type of integrated circuit that can be reconfigured repeatedly after fabrication to emulate a variety of digital circuits.

The most important elements of the FPGA are the Configurable Logic Blocks (CLBs). They are arranged in a grid and connected to each other via a configurable interconnect. Switch-Boxes (SB) are used to configure the interconnect connections and Input/Output Buffers (IOB) to enable communication with the outside world. A simplified FPGA architecture is shown in Figure 3.1.

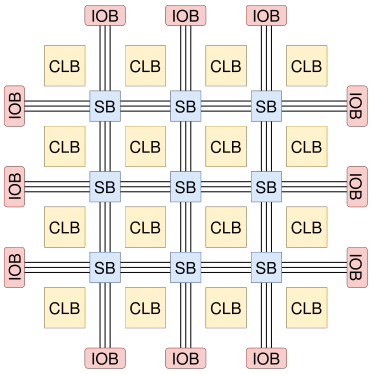


Figure 3.1: Typicall FPGA architecture.

CLBs consist of sub-elements called slices, which are further composed of look-up tables (LUT) and flip-flops (FF). A CLB architecture is depicted in Figure 3.2.

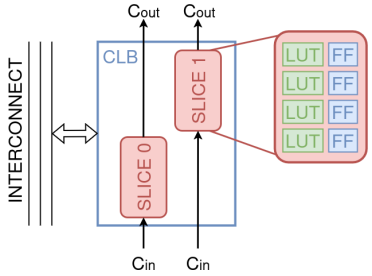


Figure 3.2: CLB architecture.

FPGAs can also include specialized function blocks that perform most common tasks more efficiently. These blocks include: static RAM memories, DSP blocks for basic arithmetic operations, specialized high speed input/output interfacing blocks (SERDES), and even embedded processors. These specialized blocks are not shown in Figure 3.1.

FPGAs emulate digital logic by configuring the combinational logic into the look-up tables (LUTs), which are specialized memories. To create more complex combinatorial functions, multiple LUTs can be combined by routing them together using FPGA routing resources. A sequential circuit is constructed by inserting flip-flops into the logic chain. Programming FPGAs by manually configuring each element would be tedious, so vendors provide Electronic Design Automation (EDA) tools that can be used to configure FPGAs using various hardware description languages (HDL). The most commonly used HDL languages are Verilog and VHDL. EDA tools transform the HDL circuit description into a bitstream file, which holds the configurations of the FPGA elements. However, languages such as Verilog or VHDL still require considerable manual effort to describe complex circuits. To overcome this challenge, new languages and techniques have been developed. These include hardware construction languages such as Chisel [31] or high-level synthesis approaches, which are described in the following section.

3.2 High-Level Synthesis

High-Level Synthesis (HLS) is a family of algorithms that aim to raise the level of abstraction in hardware development. In general, HLS tools allow us to describe the hardware using untimed behavioral descriptions, e.g., a C program. The HLS tool then transforms these descriptions into a timed structural description of the synchronous digital logic, e.g., a Verilog description. This is a challenging task because procedural languages such as C describe fundamentally sequential descriptions of data manipulation. On the other hand, the description of synchronous digital logic is fundamentally a parallel description. This dichotomy means that any such HLS algorithm must be able to identify data dependencies and determine which operations can be parallelized. Such information is usually represented by a graph containing information about data and control dependencies [32]. Another issue with procedural descriptions is that they usually do not provide constructs for specifying an arbitrary bitwidth of variables. However, this can be solved by using a library of custom types. The task of HLS can be partitioned into the following interdependent subtasks [32]:

- Scheduling - Schedule operations into clock cycles.
- Resource allocation - Assign the operations to hardware.
- Control generation - Generate the control logic for the hardware.

Finding a general solution for HLS is difficult as some of its subtasks are known to be NP-hard [32]. For this reason, practical HLS implementations use suboptimal algorithms that run in polynomial time [33]. The user can instruct the HLS algorithms by providing special comments in the code called HLS pragmas. These comments instruct the HLS algorithm whether a particular loop should be unrolled, pipelined, etc.

3.3 Chisel

The Chisel Hardware Construction Language is a domain-specific language for the design of synchronous digital circuits [31]. It is not a form of high-level synthesis, but a language

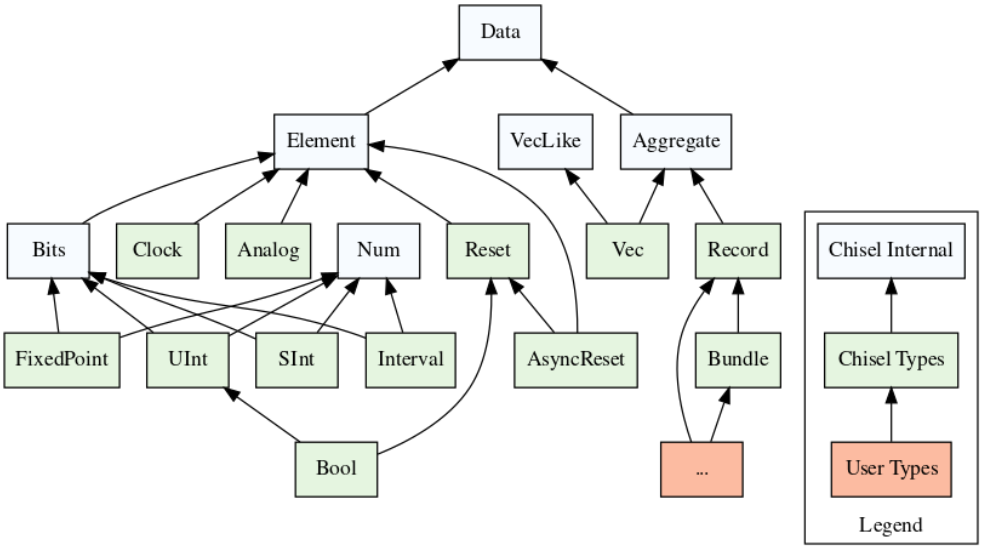


Figure 3.3: Chisel type hierarchy [34].

at the register-transfer level, which also has powerful meta-programming features. It is implemented as an embedded language in Scala, a general-purpose functional programming language with object-oriented features. The main advantage of Chisel over other hardware description languages (HDLs) such as Verilog or VHDL is that Chisel is conducive to writing reusable parameterizable generators.

Chisel hardware description is actually a Scala program that, when executed, constructs an interconnected graph of Chisel defined hardware primitives, e.g., registers, multiplexers, adders, etc., which can be exported to the Verilog hardware description. Chisel has a number of built-in data types that differ from the data types in Scala. Chisel provides a greater variety of types than most HDLs, which generally provide only a logical bit type and some sort of an array container. Chisel provides types for signed and unsigned integers, the boolean type, and aggregate types for bundles and vectors. Bundles are used to create groupings of signals, e.g., they could be used to create an AXI-Stream interface. There are also types for clocks and resets. These types are implemented in the form of a class hierarchy, which is shown in Figure 3.3. These classes also contain various logical and arithmetic operations that can be performed on the contained data, e.g., integers can be added, but bundles can only be connected.

All registers in Chisel must be explicitly created and connected to other elements by the user. Registers also have a clock signal and a reset signal. The clock and reset signals are automatically wired by Chisel to the default clock and reset signals. Chisel also allows the user to wire the clock and reset signals manually, e.g., to create a circuit with multiple clock domains.

Chisel has inference rules for inferring the width of wires. In this way, the user does not have to explicitly specify the width of each wire. Table 3.1 shows the inference rules for some common arithmetic operations in Chisel.

A Chisel description of a simple Finite Impulse Response (FIR) filter shown in Figure 3.4 is presented in Listing 3.1.

The hardware description in Chisel must be contained in classes that extend (inherit) the `Module` class. The input and output interface is created with the `IO` object. Chisel has several options for creating registers, one of which is the `RegNext` object, which is used in

Table 3.1: Width inference rules in Chisel [35].

Symbol	Operation	Width
x + y	Addition	max(w(x), w(y))
x +& y	Expanding Addition	max(w(x), w(y))+1
x - y	Subtraction	max(w(x), w(y))
x -& y	Expanding Subtraction	max(w(x), w(y)) + 1
x * y	Multiplication	w(x) + w(y)
x <<y	Dynamic left shift	w(x) + maxVal(y)
x <<n	Static left shift	w(x) + n

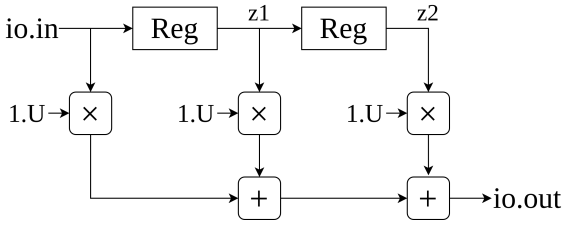


Figure 3.4: A simple moving average FIR filter.

the example shown in Listing 3.1. Line 7 declares that the inputs of `io.in` are stored in a register of the same size. The output of the register is connected to the `z1` wire.

```
1 class FirFilterSimple(bitWidth: Int) extends Module {
2   val io = IO(new Bundle {
3     val in = Input(UInt(bitWidth.W))
4     val out = Output(UInt((bitWidth + 3).W))
5   })
6
7   val z1 = RegNext(io.in) // Create a register whose input is connected to the argument io.in
8   val z2 = RegNext(z1) // Create a register whose input is connected to the argument z1
9
10  io.out := (io.in * 1.U) +& (z1 * 1.U) +& (z2 * 1.U) // ‘1.U‘ is an unsigned literal with value 1
11 }
```

Listing 3.1: A simple FIR filter [36].

A better way to describe an FIR filter is to write a generator for an arbitrary FIR filter. This way, if the parameters of the FIR filter are changed, the implementation of the FIR filter can be regenerated. Listing 3.2 shows such a parameterizable FIR filter implementation. This implementation uses more advanced Scala constructs. Its operation is as follows:

- Firstly, the `FirFilter` class has an additional parameter that corresponds to the coefficients of the FIR filter.
- Next, the input and output interface is created. The width of the output is not specified in this case, as Chisel automatically infers it based on the inference rules in Table 3.1.
- The number of provided coefficients determines the number of created registers. The `Reg` statement in line 7 instantiates these registers, but does not connect them. This is done in the for loop in lines 8-11.

- The multipliers are then created using the Scala tabulate method, which iterates through the indices of the coefficient sequence. On each iteration, a multiplier is created and connected to the register `zs(i)` and the corresponding coefficient constant `coeffs(i)`.
- Finally, the higher-order function `reduce` is used to produce an adder tree. The `reduce` function computes a single value from the collection by successively combining the elements of the collection one by one with a given binary operator.

```
1 class FirFilter(bitWidth: Int, coeffs: Seq[UInt]) extends Module {
2   val io = IO(new Bundle {
3     val in = Input(UInt(bitWidth.W))
4     val out = Output(UInt())
5   })
6   // Create the serial-in, parallel-out shift register
7   val zs = Reg(Vec(coeffs.length, UInt(bitWidth.W)))
8   zs(0) := io.in
9   for (i <- 1 until coeffs.length) {
10    zs(i) := zs(i-1)
11  }
12
13  // Do the multiplies
14  val products = VecInit.tabulate(coeffs.length)(i => zs(i) * coeffs(i))
15
16  // Sum up the products using an anonymous function
17  io.out := products.reduce(_ +& _)
18 }
```

Listing 3.2: Parameterizable FIR Filter design [36].

The operation of the `reduce` function is shown in Figure 3.5 for the case of a list of four integers and addition as the binary operator. The figure 3.5 visualizes the operation of the following Scala code snippet: `Seq(1,2,3,4).reduce((x,y) -> x + y))`.

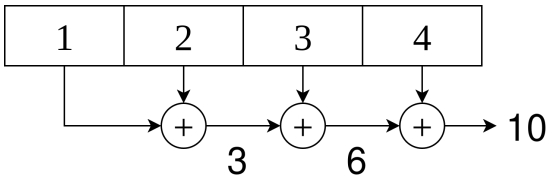


Figure 3.5: A reduce operation on a sequence of four elements using the addition as the reduction operator.

In Listing 3.2, an argument function for reduce is provided in line 17, which sums the two elements using the extending addition operation +&. The snippet "`_ +& _`" is an example of an anonymous function definition in Scala. If the position of the function arguments is irrelevant, i.e., if the function is commutative, the arguments can be denoted by an underscore. In the given example, the underscore refers to an element of the sequence `products`.

Chapter 4

State of the Art

Early attempts at deploying neural networks onto FPGA mostly used the PPU style of implementation. That is, they stored the weights off-chip in a separate memory and processed the neural network sequentially with programmable processing units. The reason for this is that quantization techniques were not yet well developed and the sizes of FPGAs devices were lower. Consequently, a highly parallel implementation would not have fitted on an FPGA device. Nevertheless, the PPU approach is still the most widely used today, as it can be scaled to larger neural network topologies. However, the focus of this work is on implementations that do not use off-chip memories and instead store the data entirely on the FPGA.

With the development of quantization techniques and the growth of available resources on FPGA devices, it has become possible to implement smaller neural networks directly on FPGAs. A common approach is to design a processing unit for each layer of the neural network. In this way, each processing unit can be tailored to the requirements of the respective layer. Processing each layer in a separate processing unit yields another level of parallelism which increases the throughput of the system. We call this approach the Data Flow Processing Unit (DFPU) approach. However, despite the use of quantization techniques, the DFPU approach is still limited to smaller neural network topologies. Larger neural network models that handle tasks such as object detection or language understanding are generally out of reach for such an approach, as even the largest FPGAs only have on-chip memory with a capacity in the order of megabytes. Much larger memories are required for these neural network models, which can currently only be realized with off-chip memories.

A common approach for the development of such processing units is to design them using HLS. In this approach, a set of parameterizable C++ kernels describing specific layers of a neural network is used together with the HLS pragma parameters to generate a hardware implementation. The result is a highly parallel digital circuit that can process a quantized neural network with very low latency and high throughput. Two of the most commonly used tools that implement this approach are hls4ml and FINN.

4.1 hls4ml

The hls4ml project arose from the need for high speed data processing at the CERN Large Hadro Collider (LHC). The sensor system at the CERN LHC acquires data at a frequency of 40 MHz, which leads to very high data rates in the range of terabytes per second. Such high data rates are challenging to process in real-time. Therefore, a machine learning based triggering system was proposed to filter out events that are not of interest for particle physics research. Due to the limited size of data buffers, the online machine

learning trigger must be computed with very low latencies in the order of microseconds [2].

Such low latency requirements rule out the possibility of using conventional accelerators such as GPGPUs, as these are not optimized for low latency but for high throughput. In general, it would be very difficult to achieve such low latencies with a solution that requires access to DRAM memory, as accessing memory alone introduces significant latency [7].

The hls4ml tool was developed to solve this problem. The hls4ml tool automates the translation of a trained neural network into HLS C++ code, which is then translated into Verilog. The parameters of the neural network parameters are contained entirely on the FPGA. Figure 4.1 shows a typical workflow when using hls4ml:

- The input to hls4ml is a trained neural network model, e.g., a Keras model.
- The model can then undergo various forms of compression, e.g., quantization.
- The compressed model is then first converted into an internal representation and then into HLS C++ code.
- The user can set various parameters that influence the trade-off between achieved hardware performance and resource consumption. These parameters are then used to configure the HLS pragmas in the HLS C++ code.
- Finally, the HLS C++ code is translated into Verilog code, which can then be synthesized for a specific FPGA.

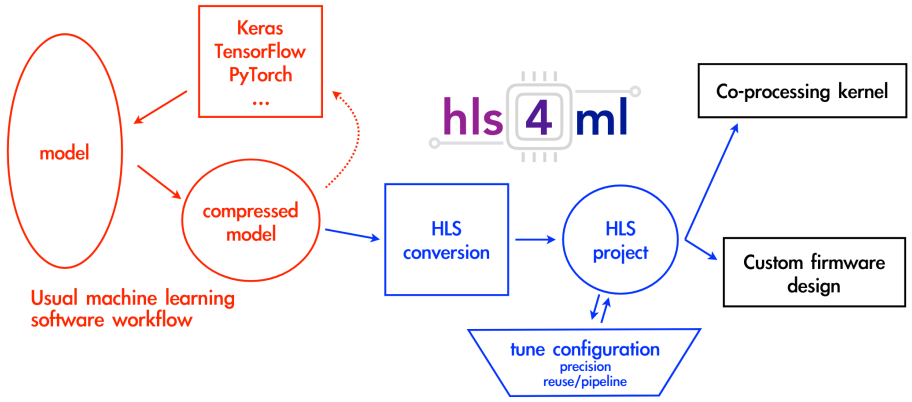


Figure 4.1: The hls4ml workflow [2].

The hls4ml tool supports both non-quantized and quantized models. hls4ml also offers the possibility to perform post-training quantization techniques in order to compress the model. This post-training quantization is a necessary step in the case of a non-quantized NN model, since it is almost impossible to fit the highly parallel hardware implementation of a floating-point NN model into an FPGA device. The hls4ml tool supports the following input formats:

- non-quantized: Keras, Pytroch, or ONNX models,
- quantized: QKeras or QONNX models.

All the model formats described above are then translated into an internal hsl4ml representation called `ModelGraph`. This representation is, as the name suggests, a graph-based representation of neural networks. It is implemented as a Python class hierarchy and can represent a wide range of deeply quantized neural networks.

The hls4ml internal representation can then be translated into HLS C++ code for a wide range of different HLS tools, including: AMD Vivado HLS, AMD Vitis HLS, Intel Quartus, Intel OneAPI, and Siemens Catapult. The software components responsible for converting the different types of neural network models into the `ModelGraph` representation are referred to as frontends, and the software components that translate the `ModelGraph` object into the specific HLS C++ code flavor are referred to as backends. This software architecture is shown in Figure 4.2.

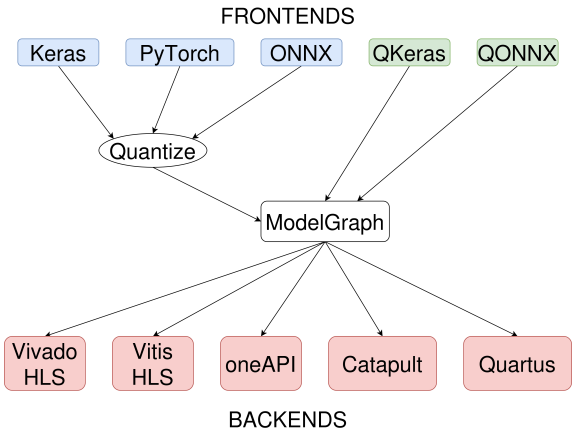


Figure 4.2: The hls4ml software architecture.

Listing 4.1 shows an example of using the hls4ml Python API to transform a trained QONNX model into a Verilog implementation using the Vitis backend.

```
from qonnx.core.modelwrapper import ModelWrapper
import hls4ml

qonnx_model = ModelWrapper('/path/to/model.onnx')
hls_config = hls4ml.utils.config_from_onnx_model(
    qonnx_model,
    granularity="name",
    backend="Vitis",
    default_reuse_factor=1,
)
hls_model = hls4ml.converters.convert_from_onnx_model(
    qonnx_model,
    output_dir=work_dir,
    backend="Vitis",
    io_type="io_parallel",
    hls_config=hls_config,
    part="xcvu9p-flga2104-2L-e",
)
hls_model.compile()
build_info = hls_model.build(csim=False, synth=True, cosim=True, vsynth=True)
```

Listing 4.1: Demonstration of hls4ml Python API.

4.1.1 Backend HLS Implementations

This subsection describes how the computational kernels in hls4ml are designed. The kernels are implemented in C++ and use the C++ templates feature to make the implementation generic with respect to the type of input, output, bias and weight tensors.

4.1.1.1 Fully Connected Layer Implementations

There are several implementations of the fully connected layer in hls4ml. One of them, called `dense_latency`, is geared towards high-parallelization and low latency, while three

others are aimed at lower resource consumption. The focus of this thesis is on the implementation with high-parallelization and low latency, as this is of interest for the comparison with chisel4ml. Nevertheless, it is briefly described how the resource-optimized implementations differ from the latency-optimized implementations. The selection of implementations is guided by the user by providing additional configurations to control this selection.

Listing 4.2 shows an abridged template-based C++ function that represents a high-parallelization and low latency implementation of the fully connected layer. For clarity, the HLS pragmas that steer the HLS algorithm in designing the hardware have been removed. Four arrays are parameters of the function: `data`, `res`, `weights`, and `biases`. The inputs of the layer, the outputs of the layer, the weights of the layer, and the biases of the layer are stored in these arrays. The C++ template feature is used to abstract the function over the input type `data_T` and the output type `res_T`. An additional template parameter is the `CFG_T` parameter[1]. It contains a plethora of information: various constants that define the size of the inputs, outputs, etc., the types of the weights and the accumulator, and it contains the implementation of the product function. The product function is abstracted so that it can be replaced by a specialized implementation, e.g., binarized neural networks use the XNOR function.

```
template <class data_T, class res_T, typename CFG_T>
void dense_latency(data_T data[CFG_T::n_in],
                   res_T res[CFG_T::n_out],
                   typename CFG_T::weight_t weights[CFG_T::n_in * CFG_T::n_out],
                   typename CFG_T::bias_t biases[CFG_T::n_out]) {
  data_T cache;
  typename CFG_T::accum_t mult[CFG_T::n_in * CFG_T::n_out];
  typename CFG_T::accum_t acc[CFG_T::n_out];

  // Product1 - Do the matrix multiply
  for (int ii = 0; ii < CFG_T::n_in; ii++) {
    cache = data[ii];
    // Product2
    for (int jj = 0; jj < CFG_T::n_out; jj++) {
      int index = ii * CFG_T::n_out + jj;
      mult[index] = CFG_T::template product<data_T, typename CFG_T::weight_t>
          ::product(cache, weights[index]);
    }
  }

  // ResetAccum - Initialize accumulator with input biases
  for (int iacc = 0; iacc < CFG_T::n_out; iacc++) {
    acc[iacc] = (typename CFG_T::accum_t)biases[iacc];
  }

  // Accum1 - Accumulate multiplication result
  for (int ii = 0; ii < CFG_T::n_in; ii++) {
    // Accum2
    for (int jj = 0; jj < CFG_T::n_out; jj++) {
      int index = ii * CFG_T::n_out + jj;
      acc[jj] += mult[index];
    }
  }

  // Result - Cast to "res_t" type
  for (int ires = 0; ires < CFG_T::n_out; ires++) {
    res[ires] = cast<data_T, res_T, CFG_T>(acc[ires]);
  }
}
```

Listing 4.2: A minimal latency implementation of the fully connected layer in hls4ml [37].

The basic steps of the function `dense_latency`, which calculates the fully connected layer, are:

- The `Product1` and `Product2` loops perform the multiplications of the input features with the weights and store the individual products in the `mult` array. The multi-

[1] In the hls4ml source code, the `CFG_T` is referred to as the `CONFIG_T` parameter, but it was shortened for presentation purposes.

plication is performed with a custom product function that depends on the type of quantization.

- The `ResetAccum` loop simply initializes the accumulator with the input biases.
- The `Accum1` and `Accum2` loops through the products stored in the `mult` array and add them to the previously initialized accumulator to obtain the the matrix-vector product.
- Finally, the `Result` loop casts the accumulator to the output quantization type.
- In this implementation, the activation function is missing as it is handled in a separate function.

The C++ implementation in Listing 4.2 is translated by an HLS tool into a Verilog code that describes a highly parallel digital circuit. In extreme cases, it can be converted into a fully parallel circuit, i.e., a direct logic circuit. Intuitively, this can be thought of as fully unrolling all loops in Listing 4.2.

Other implementations of the fully connected layer in hls4ml focus on reducing the resources used and do not fully unroll all loops. Instead, the `Product1` loop is pipelined, and the hls4ml user can control the degree of unrolling in the `Product2` loop. This is controlled with a parameter called the `ReuseFactor`. If the `ReuseFactor` equals 1, this corresponds to a fully parallel `Product2` implementation, if the `ReuseFactor` equals 2, the hardware implementation computes the `Product2` loop in two iterations, etc. In our experiments, the `ReuseFactor` is always set to 1 to produce highly parallel implementations.

4.1.1.2 Convolution Layer Implementations

There are also several implementations of the convolution layer in hls4ml. In a similar fashion to the fully connected layer, there is a highly parallelized and low latency implementation called `conv_2d_latency_cl`, as well as implementations that aim to reduce FPGA resource consumption.

Listing 4.3 gives an abridged version of the highly parallelized and low latency implementation of the convolution layer in hls4ml. The HLS pragmas have again been removed from the listing. The `conv_2d_latency_cl` function also has the same four array parameters: `data`, `res`, `weights`, and `biases`. These parameters correspond to the input, output, weight and bias tensors of the layer.

```
template <class data_T, class res_T, typename CFG_T>
void conv_2d_latency_cl(data_T data[CFG_T::in_height * CFG_T::in_width * CFG_T::n_chan],
                        res_T res[CFG_T::out_height * CFG_T::out_width * CFG_T::n_filt],
                        typename CFG_T::weight_t weights[CFG_T::filt_height * CFG_T::filt_width *
                             CFG_T::n_chan * CFG_T::n_filt],
                        typename CFG_T::bias_t biases[CFG_T::n_filt]) {

  constexpr unsigned mult_n_in = CFG_T::filt_height * CFG_T::filt_width * CFG_T::n_chan;
  constexpr unsigned mult_n_out = CFG_T::n_filt;

  data_T data_buf[CFG_T::n_pixels][mult_n_in];
  typename CFG_T::accum_t mult[mult_n_in * mult_n_out];
  typename CFG_T::accum_t acc[mult_n_out];

  // PartitionLoop
  for (int i_part = 0; i_part < CFG_T::n_partitions; i_part++) {
    CFG_T::template fill_buffer<data_T, CFG_T>::fill_buffer(data, data_buf, i_part);

    // PixelLoop
    for (unsigned i_pxl = 0; i_pxl < CFG_T::n_pixels; i_pxl++) {
      data_T cache;

      // Product1 - do the matrix multiply
      for (int i_in = 0; i_in < mult_n_in; i_in++) {
        cache = data_buf[i_pxl][i_in];
        // Product2
        for (int i_out = 0; i_out < mult_n_out; i_out++) {
```

```
27          mult[i_in * mult_n_out + i_out] = CFG_T::mult_config::template product<data_T, typename
                CFG_T::mult_config::weight_t>::product(cache, weights[i_in * mult_n_out + i_out]);
28        }
29      }
30
31      // ResetAccum - initialize accumulator with input biases
32      for (int i_acc = 0; i_acc < mult_n_out; i_acc++) {
33        acc[i_acc] = (typename CFG_T::accum_t)biases[i_acc];
34      }
35
36      // Accum1 - Accumulate multiplication result
37      for (int i_in = 0; i_in < mult_n_in; i_in++) {
38       // Accum2
39        for (int i_out = 0; i_out < mult_n_out; i_out++) {
40          acc[i_out] += mult[i_in * mult_n_out + i_out];
41        }
42      }
43
44      // Result - Cast to "res_t" type
45      for (int i_res = 0; i_res < mult_n_out; i_res++) {
46        *(res++) = cast<data_T, res_T, typename CFG_T::mult_config>(acc[i_res]);
47      }
48    } // PixelLoop
49  } // PartitionLoop
50}
```

Listing 4.3: A minimal latency implementation of the convolution layer in hls4ml [37].

The `conv_2d_latency_cl` function implements the convolution operation with a total of eight loops. Because convolution layers often work on high-dimensional data and can consume large amounts of FPGA resources, even this highly parallelized implementation of the convolution layer is not always fully unrolled. Instead, it is divided into a number of partitions, which are stored in the parameter `CFG_T::n_partitions`. However, a fully unrolled implementation is possible if the `CFG_T::n_partition` is equal to 1. The `CFG_T::n_partition` parameter is not configured directly by the user of hls4ml, instead the user configures a parameter called `ParallelizationFactor`. This parameter controls how many neurons, e.g., pixels, are computed simultaneously. If the `ParallelizationFactor` equals the number of neurons in the layer, then the `CFG_T::n_partitions` parameter is equal to 1.

Next, the `PixelLoop` loop computes the matrix-vector product in the same way as the fully connected implementation described in the previous Section 4.1.1.1. In the low latency implementation, the `PixelLoop` and all inner loops are fully unrolled. This includes the `Product1`, `Product2`, `ResetAccum`, `Accum1`, `Accum2`, and the `Result` loop.

The low resource consumption implementations of the convolution layers use two parameters to reduce FPGA resource consumption:

- `ParallelizationFactor`
  As described above, this controls how many neurons (or pixels) can be computed simultaneously. The minimum setting is 1, the maximum setting corresponds to the number of neurons in the layer.

- `ReuseFactor`
  Similar to the implementation of the fully connected layer, the `ReuseFactor` controls the level of folding of the matrix-vector product. So if using a low resource implementation, the `Product1` loop will be pipelined, and the `Product2` loop will be unrolled depending on the value of the `ReuseFactor`. If the `ReuseFactor` is 1, the `Product2` loop is implemented completely in parallel, and if the `ReuseFactor` is 2, the multiplication is implemented in two iterations, and so on.

For comparison with chisel4ml, the `ParallelizationFactor` is set to its maximum value and the `ReuseFactor` is set to its minimum value. These settings produce the most parallel circuit possible.

4.1.1.3 Pooling Layer Implementations

The implementation of pooling layers in hls4ml is written in such a way that the pooling function is abstracted by a template parameter. This allows the same implementation to be used for different types of pooling layers, e.g., maximum pooling layers or average pooling layers. The pooling layer implementations, as opposed to the convolution layer or fully connected layer implementations, are not split into low latency and low resource implementations. Instead, they are categorized by the size of the pooling window and the expected format of the input data. The following implementations exist: `pooling2d_cl`, `pooling2d_cf`, `global_pooling2d_cl`, and `global_pooling2d_cf`. The last two implementations listed only handle cases in which the size of the pooling window corresponds to the size of the input window. This is rarely the case, so we will not discuss them. The first two implementations listed: `pooling2d_cl`, `pooling2d_cf`, are similar. The difference is that the former implementation handles the case where the input data is in the channels last data layout format, and the latter is for channel first data layout format.

Listing 4.4 shows an abridge version of the `pooling2d_cl` pooling layer implementation. Some constant parameter defintions and the HLS pragmas were removed for brevity.

```
template <class data_T, class res_T, typename CFG_T>
void pooling2d_cl(data_T data[CFG_T::in_height * CFG_T::in_width * CFG_T::n_filt],
                  res_T res[CFG_T::out_height * CFG_T::out_width * CFG_T::n_filt]) {
  // FilterLoop
  for (int ff = 0; ff < CFG_T::n_filt; ff++) {
    // DataYLoop - loop over input image y in steps of stride
    for (int ii = 0; ii < restricted_padded_height; ii += CFG_T::stride_height) {
      // DataXLoop - loop over input image x in steps of stride
      for (int jj = 0; jj < restricted_padded_width; jj += CFG_T::stride_width) {
        data_T pool[CFG_T::pool_height * CFG_T::pool_width];
        unsigned overlap_pixel = 0;

        // KernelYLoop - loop over pool window y
        for (int kk = 0; kk < CFG_T::stride_height; kk++) {
          // KernelXLoop - loop over pool window x
          for (int ll = 0; ll < CFG_T::stride_width; ll++) {
            bool c1 = ii + kk >= CFG_T::pad_top && ii + kk < CFG_T::in_height + CFG_T::pad_top;
            bool c2 = jj + ll >= CFG_T::pad_left &&jj + ll < CFG_T::in_width + CFG_T::pad_left;
            if (c1 && c2) {
              unsigned data_idx = ((ii + kk - CFG_T::pad_top) * CFG_T::in_width +
                  (jj + ll - CFG_T::pad_left)) * CFG_T::n_filt + ff;
              pool[kk * CFG_T::stride_width + ll] = data[data_idx];
              overlap_pixel++;
            } else
              pool[kk * CFG_T::stride_width + ll] = pad_val<data_T, CFG_T::pool_op>();
          }
        }

        int patch_size = CFG_T::count_pad ? CFG_T::stride_width * CFG_T::stride_height :
            overlap_pixel;

        res[(ii / CFG_T::stride_height) * CFG_T::out_width * CFG_T::n_filt +
            (jj / CFG_T::stride_width) * CFG_T::n_filt + ff] =
            pool_op<data_T, CFG_T::pool_height * CFG_T::pool_width, CFG_T::pool_op,
                  typename CFG_T::accum_t>(pool, patch_size);
      }
    }
  }
}
```

Listing 4.4: Channel last pooling layer implementation in hls4ml [37].

The function `pooling2d_cl` implements a two-dimensional pooling operation. The input of the function is the `data` array and the output is the `res` array. The function uses templates in a similar fashion to the fully connected and convolution layer functions. It has three template parameters that represent the input data type `data_T`, the output data type `res_T`, and a configuration type `CFG_T` that contains a plethora of data, including the pooling function. The pooling implementation consists of five loops:

- `FilterLoop`
  Loops through the input channels of the tensor. The name of the loop is somewhat

misleading, however, a pooling operation can also be thought of as an operation of several kernels over each channel of the input tensor, in a similar fashion as in depthwise convolution.

- `DataYLoop` and `DataXLoop`
  These loops iterate through the height and width dimensions of the input tensor. They are incremented by a stride, which corresponds to the size of the kernel window. For each `ii` and `jj` pair, the kernel loops fill in the `pool` array, which represents the currently active window. After the `pool` array is filled by the kernel loops, it is passed to a pooling function that calculates the selected pooling operation.

- `KernelYLoop` and `KernelXLoop`
  These loops iterate through the kernel window. At each point, two conditions `c1` and `c2` are evaluated. These conditions check whether a certain index lies within the padded input window. If this is the case, the specified value is added to a newly created `pool` array. If the conditions `c1` and `c2` do not hold, a fill value is added to the `pool` array, usually a zero. These two loops fill the `pool` array with the currently active window. Once the pool array is complete, it is passed to a pooling function provided by the user. Listing 4.5 gives the `max` function, which is used for implementing the maximum pooling layer. The `max` function simply returns the largest element in the array.

```
template <typename T, int N, typename accum_t> accum_t max(T x[N]) {
    T y = x[0];
    for (int i = 1; i < N; i++) {
        y = x[i] > y ? x[i] : y;
    }
    return y;
}
```

Listing 4.5: A implementation of the maximum pooling function in hls4ml [37].

The parameters in Listing 4.4 are determined by the topology of the neural network:

- `restricted_padded_height` — Represents the height of the input including the padding added at the top and bottom.

- `restricted_padded_width` — Represents the width of the input including the padding added to the left and right.

- `stride_height` / `stride_width` — Represent the stride in the corresponding height and width dimension of the input tensor. These values do not have to be the same.

- `count_pad` — Whether the values that are added as padding should also be included in the pooling function. If not, padding is still a useful way to control the dimensionality of the output.

4.2 FINN

FINN is a framework developed by Xilinx Research Labs to automate the process of implementing deeply quantized neural networks on FPGAs. It is based on high-level synthesis technology, but unlike hls4ml, it is limited to Xilinx HLS tools only. FINN also has a backend implementation in SystemVerilog RTL. However, the RTL implementation has some limitations compared to the HLS implementations and is therefore not discussed further [38]–[40].

The FINN tool consists of a Python API that is used to read the quantized neural network model and generate the Verilog code. Listing 4.6 shows an example of using the FINN Python API to convert a trained QONNX model into a Verilog implementation.

```
import finn.builder.build_dataflow as build
import finn.builder.build_dataflow_config as build_cfg

cfg = build.DataflowBuildConfig(
    steps=build_cfg.default_build_dataflow_steps,
    output_dir="/my/output/dir",
    synth_clk_period_ns=10.0,
    fpga_part="xczu3eg-sbva484-1-e",
    verbose=True,
    generate_outputs=[
        build_cfg.DataflowOutputType.STITCHED_IP
    ]
    )
)
build.build_dataflow_cfg(
    "/path/to/qonnx/model.onnx",
    cfg
)
```

Listing 4.6: Demonstration of FINN Python API.

Instead of implementing separate kernels for each layer like hls4ml, FINN implements a number of different parameterizable processing units that can be combined to compute the different layers of the neural network. The most commonly used processing units are the:

- Matrix-Vector-Activation Unit (MVAU),
- Sliding Window Generator Unit (SWG) or Convolution Input Generator,
- Vector-Vector-Activation Unit (VVAU),
- Pooling Unit.

In the next subsections a brief description of the different processing units and how they can be combined to implement the different layers will be provided.

Due to the complexity of the implementations of the FINN processing units, the code is not presented. They are implemented as templated C++ functions similar to the hls4ml implementations. An interested reader can have a look at the FINN GitHub repository: `https://github.com/Xilinx/finn-hlslib`.

4.2.1 Matrix-Vector-Activation Unit

The Matrix-Vector-Activation Unit (MVAU) computes a matrix-vector product and then computes the activation function on the products. This corresponds to the computation of a fully connected layer.

An MVAU consists of subunits called *Processing Elements* (PE) that compute the dot product at different levels of parallelization. The PE can also be considered as a processing unit to compute the output of a neuron. The architecture of a PE is shown in Figure 4.3. The PE consists of a weight and threshold memory, which are indexed by the *index* signal. The weight memory has Q parallel lanes with bitwidth of W. These lanes are entered together with Q inputs with the bitwidth of A in a SIMD-type multiplication operation. In Figure 4.3 this is shown as a MUL box. The MUL operator does not necessarily have to be a normal multiplication, it could also be replaced by an XNOR operation for BNNs. The result of the MUL operation is then fed into an adder tree and added to the accumulator. When the total dot product is computed, the value of the accumulator is output to the threshold unit, which computes an activation function, e.g., the rectified linear unit. The accumulator is pre-loaded with the bias value.

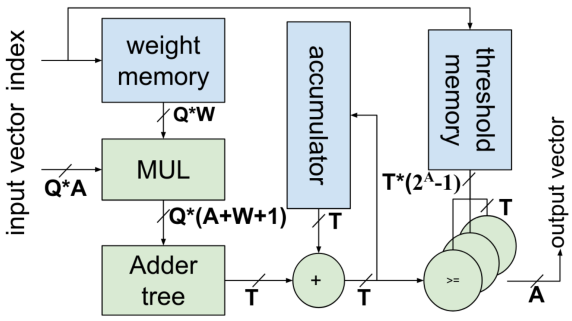


Figure 4.3: The processing element [38].

The architecture of the MVAU is shown in Figure 4.4. The MVAU consists of P processing elements, each of which has a SIMD lane size of S. In terms of a fully connected layer, the implementation with the highest degree of parallelization results if you set the lane size S equal to the size of the input features and P equal to the number of neurons in the layer or equal to the size of the output features. If, on the other hand, you set S and P equal to 1, you get the slowest implementation.

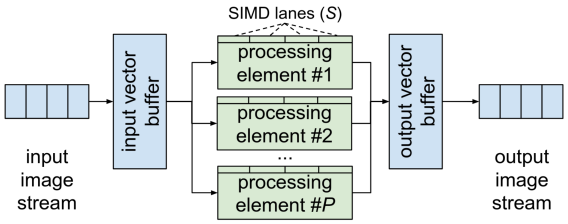


Figure 4.4: Matrix-Vector-Activation Unit [38].

An additional parameter that influences the MVAU implementation is the *mem_mode* parameter. It determines how the weight memory is implemented. The MVAU implementation supports three options:

- `internal_decoupled` — in this mode, FINN treats the weights as constants that are known at compile time. This means that they become part of the HLS C++ code and can be optimized. A side effect of this approach is that it reduces the the compilation time.
- `internal_encoded` — in this mode the weights are stored in an FPGA memory and streamed to the MVAU unit.
- `external` — allows the user to provide the weights from an external stream.

In order to achieve designs with the highest possible throughput the *mem_mode* parameter must be set to *internal_decoupled*.

4.2.2 Sliding Window Unit

The convolution operation operates over an input data window that slides over the input data. At each sliding window position its elements are multiplied by corresponding kernel elements and summed together. Thus, the convolution operation can also be represented as a matrix-vector product.

To illustrate this let us consider two-dimensional convolution operation over a single channel input matrix described by Equation 4.1, where the $*$ symbol represents the two-dimensional convolution operation.

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \end{bmatrix} * \begin{bmatrix} k_1 & k_2 \\ k_3 & k_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \tag{4.1}$$

The equivalent operation using the matrix-vector product, shown in Equation 4.2, is determined by identifying sliding window elements.

$$\begin{bmatrix} x_1 & x_2 & x_4 & x_5 \\ x_2 & x_3 & x_5 & x_6 \end{bmatrix} \cdot \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \tag{4.2}$$

This is a common technique to accelerate convolution on SIMD and GPGPU platforms. It is sometimes also referred to as an image-to-column or im2col operation.

Instead of a dedicated processing unit for convolution, a Sliding Window Unit (SWU) is implemented in FINN. The unit buffers an incoming data stream that uses the channels last data layout and transforms it into an output stream of windows in the channels first data layout format. In other words, it provides the vectors to compute dot products with the filters to compute the convolution. In this way, a convolution layer can be implemented in FINN with a combination of an SWU and an MVAU. The functionality of the SWU is shown in Figure 4.5. It shows how the input features, that have two channels, are transformed in the SWU before they are forwarded to an MVAU.

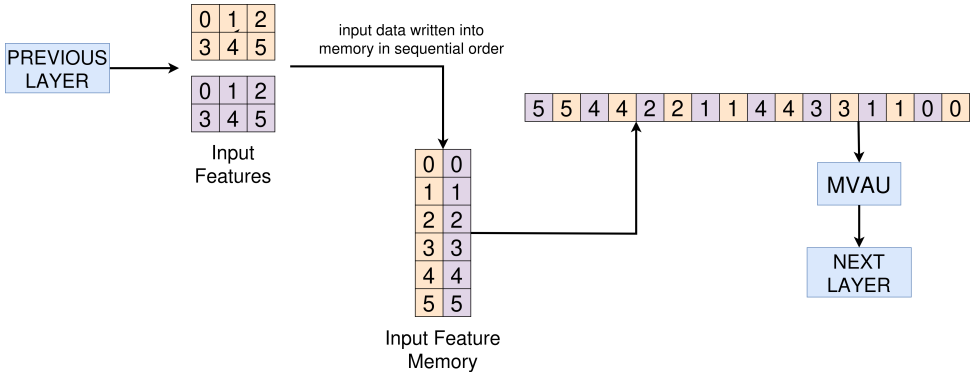


Figure 4.5: The operation of the Sliding Window Unit [39].

4.2.3 Vector-Vector-Activation Unit

The Vector-Vector-Activation Unit (VVAU) is used to compute dot products between two vectors. It is used to implement depthwise convolution. Although depthwise convolution can also be expressed as a matrix-vector product, it can be performed without changing the layout of the tensor from channel last data layout to channel first data layout. Because of this, it is more efficient to compute it as vector-vector multiplication, as the processing unit can already start the computation as soon as it receives the first channel. This is not the case with normal convolution, as each output is dependent on all channels.

To see why this is the case, consider the depthwise convolution shown in Equation 4.3. It shows a depthwise convolution of a two-by-three input image with two channels, where

a channel last data layout is assumed and where each channel has its own kernel of size 2 by 2. The $\star$ symbol stands for the depthwise convolution operation.

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \end{bmatrix} \star \begin{bmatrix} k_1 & k_2 \\ k_3 & k_4 \end{bmatrix} = \begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix} = \begin{bmatrix} x_1k_1 + x_2k_2 + x_4k_3 + x_5k_4 & x_2k_1 + x_3k_2 + x_5k_3 + x_6k_4 \\ x_7k_5 + x_8k_6 + x_{10}k_7 + x_{11}k_8 & x_8k_5 + x_9k_6 + x_{11}k_7 + x_{12}k_8 \end{bmatrix} \\ \begin{bmatrix} x_7 & x_8 & x_9 \\ x_{10} & x_{11} & x_{12} \end{bmatrix} \star \begin{bmatrix} k_5 & k_6 \\ k_7 & k_8 \end{bmatrix} \tag{4.3}$$

Equation 4.3 can be rewritten using a matrix multiplication, as shown in Equation 4.4. It is important to note that the elements of the input array x_n are streamed to the vector unit in the same order as they are numbered in the Equations 4.3 and 4.4. This means that processing can begin as soon as x_5 is received.

$$\begin{bmatrix} x_1 & x_2 & x_4 & x_5 \\ x_2 & x_3 & x_5 & x_6 \end{bmatrix} \cdot \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \\ \begin{bmatrix} x_7 & x_8 & x_{10} & x_{11} \\ x_8 & x_9 & x_{11} & x_{12} \end{bmatrix} \cdot \begin{bmatrix} k_5 \\ k_6 \\ k_7 \\ k_8 \end{bmatrix} = \begin{bmatrix} y_3 \\ y_4 \end{bmatrix} \tag{4.4}$$

The input vectors to the VVAU are generated by a variant of SWU. It works in exactly the same way as the SWU described above, with the difference that it does not change the order of the channels, i.e., the data layout.

The structure of the VVAU is almost identical to the MVAU. An interested reader can find more information in the FINN GitHub repository: `https://github.com/Xilinx/finn-hlslib`.

4.2.4 Pooling Unit

Similar to the other layers, the pooling layers are implemented as a combination of two processing units: the SWU and the Pool unit. The role of the SWU is identical to its role in the implementation of the convolution layer. That is, it converts channel last tensors into a series of windows of the input tensor.

This makes the Pool unit very simple. It operates in a streaming fashion, reading parts of a tensor on one side and streams the results on the other side. The input streams of an $H_I \times W_I \times C$ input tensor with a $K \times K$ filter size are stored in a line buffer with $H_I * C * K$ elements. The elements of the buffer are processed with a pooling function of choice, e.g., maximum element selection, and the result is streamed out [39].

Chapter 5

chisel4ml

The biggest disadvantage of HLS-based tools is the computational complexity of HLS. Even some subtasks of HLS are known to be NP-hard. As a result, HLS-based tools like hls4ml struggle with long synthesis times. It can take days to generate hardware implementations of even moderately sized neural networks. Moreover, HLS-based techniques do not necessarily produce optimal circuits [40]. To overcome this issue, we propose the Chisel-based chisel4ml tool [41], [42]. Since Chisel uses structural descriptions of circuits, the hardware generators are able to generate the hardware of the neural network in a time proportional to the number of neurons, i.e., linear time complexity. This greatly improves the scalability of the approach. The main disadvantage, however, is that this limits the possible hardware structures to those for which the generators were designed.

The chisel4ml tool consists of two main components: a Python frontend and the Chisel backend. The Python frontend provides a programming interface, as much of the machine learning infrastructure also has a programming interface in Python. The backend contains the Chisel hardware generators for the processing units used to implement various neural network layers. The Python frontend communicates with the Chisel backend via the services provided by the latter, e.g., generate, simulate, etc. In the generate service data exchange the neural network model is represented in Low-Bitwidth Intermediate Representation (LBIR) format. This is an internal description format for neural networks that was developed to simplify the development of Chisel hardware generators. Figure 5.1 shows the high-level software architecture of chisel4ml.

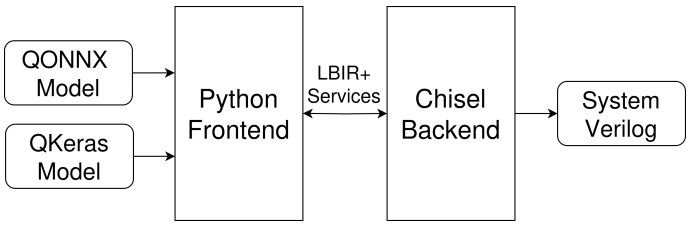


Figure 5.1: High-level software architecture of chisel4ml.

5.1 Low-Bitwidth Intermediate Representation Format

The input to the chisel4ml tool are the QONNX and QKeras model representations. Although it is possible to translate a QONNX or QKeras model directly into hardware structures, it is easier to first transform it into an intermediate form that simplifies the representation of the model. Another advantage of a unified representation is the ability to

efficiently support multiple model descriptions, as they only need to be translated into this intermediate form and the backend only needs to understand this intermediate form. In chisel4ml, this intermediate format is called Low-Bitwidth Intermediate Representation (LBIR) and provides a unified representation of a quantized neural network model. LBIR and chisel4ml can be extended with specific modules that need not be related to neural networks, e.g., an audio pre-processing pipeline [43]. LBIR is implemented with protocol buffers, a data exchange format developed by Google. LBIR uses the following attributes to represent DQNNs:

- `Datatype`
  The `Datatype` object specifies the quantization information of a particular tensor it is attached to. The `Datatype` object can represent a variety of uniform quantization specifications as well as binary quantization. It assumes that the scale factor is a power of two, as in this case the scaling can be performed as a shift by the exponent. In other words, the value is shifted by base two logarithm of the scale factor.

- `QTensor`
  The `QTensor` object is composed of the quantization information given by the `Datatype` object, the shape (tensor dimensions), the layout, information about the rounding operation, and optionally the actual values of the tensor. If the `QTensor` object does not contain any values, it is used as a placeholder for the quantization information. For example, a `QTensor` object at the input of the layer signifies the expected quantization of the input values.

- `Activation`
  The type of activation function.

- `LayerWrap`
  The `LayerWrap` object is a union of different layer objects. There are specific layer objects for the fully connected layer, the two-dimensional convolution layer, and the maximum pooling layers. Each of these layers consists of a set of `QTensor` objects that define the inputs, outputs and possibly the weight tensors. Depending on the layer type, additional parameters may also be present.

- `Model`
  The `Model` object is an ordered list of layers that define the feed-forward neural network.

Listing 5.1 gives an abridged definition of the LBIR representation with protocol buffers.

```
message Datatype {
   enum QuantizationType {
       UNIFORM = 0;
       BINARY = 1;
   }
    QuantizationType quantization = 1;
    bool signed = 2;
    uint32 bitwidth = 3;
    repeated sint32 shift = 4; // scale factor logarithm
    repeated sint32 offset = 5;
}

message QTensor {
  Datatype dtype = 1;
  repeated uint32 shape = 2;
  repeated float values = 3;
  string rounding_mode = 4; // up, down, half even...
}

message Model {
   string name = 1;
   repeated LayerWrap layers = 2;
}
```

```
24
25 enum Activation {
26     BINARY_SIGN = 0;
27     RELU = 1;
28     NO_ACTIVATION = 2;
29 }
30
31 message LayerWrap {
32     oneof sealed_value_optional {
33         DenseConfig dense = 1;
34         Conv2DConfig conv2d = 2;
35         MaxPool2DConfig maxpool2d = 3;
36     }
37 }
38
39 message DenseConfig {
40     QTensor thresh = 1;
41     QTensor kernel = 2;
42     QTensor input = 3;
43     QTensor output = 4;
44     Activation activation = 5;
45 }
46
47 message Conv2DConfig {
48     QTensor thresh = 1;
49     QTensor kernel = 2;
50     QTensor input = 3;
51     QTensor output = 4;
52     Activation activation = 5;
53     bool depthwise = 7;
54     repeated uint32 stride = 8;
55     repeated uint32 padding = 9;
56 }
57
58 message MaxPool2DConfig {
59     QTensor input = 1;
60     QTensor output = 2;
61     repeated uint32 kernel_shape = 3;
62     repeated uint32 stride = 4;
63     repeated uint32 padding = 5;
64 }
```

Listing 5.1: Protocol buffers definition of LBIR.

5.2 Python Frontend

The frontend of chisel4ml is written in Python to simplify interfacing with machine learning frameworks such as Brevitas or QKeras. Listing 5.2 illustrates how chisel4ml is used to transform a QONNX model into a System Verilog implementation.

```
1 from chisel4ml import generate, transform
2 from qonnx.core.modelwrapper import ModelWrapper
3 from my_dataset import test_data
4
5 qonnx_model = ModelWrapper("/path/to/qonnx/model.onnx")
6 lbir_model = transform.qonnx_to_lbir(qonnx_model)
7 accelerators = generate.accelerators(lbir_model)
8 circuit = generate.circuit(accelerators, lbir_model)
9 res = circuit.predict(test_data) # test_data provided as a numpy array
10 circuit.package(directory="/my/pkg/dir/")
```

Listing 5.2: Demonstration of chisel4ml frontend.

The Python frontend has the following tasks:

- **Translating DQNN representations**
  The Chisel backend works with LBIR descriptions. For this reason, the Python frontend must provide facilities to translate descriptions from DQNN to LBIR. Currently chisel4ml supports the two most common DQNN training frameworks QKeras and Brevitas by translating QONNX and QKeras models into LBIR.

- **Simulation interface**
  Once the circuit has been generated by the Chisel backend, the circuit simulation can be triggered from Python. The Python frontend allows the user to simulate a given

circuit by providing numpy arrays as input. In the process the frontend translates the numpy arrays into an appropriate `QTensor` objects and verifies that the numpy array values are appropriately quantized.

- **Circuit packaging**
  Once the designer is satisfied with the circuit, chisel4ml facilitates the packaging of the circuit by copying it to a desired location.

The Python frontend communicates with the Chisel backend using the Google Remote Procedure Call (gRPC) library. The gRPC library is integrated with protocol buffers. The definition of the communication interface is presented in Listing 5.3. The Chisel backend supports four types of services:

- `GetVersion`
  Used to check the version compatibility between the frontend and the backend.

- `GenerateCircuit`
  Generates the requested accelerators that implement the given LBIR layers. Currently, chisel4ml mainly uses only one processing unit called `ProcessingElementCombToSeq`. This processing unit represents a fully unrolled direct logic circuit implementation of the given layers. In the future, however, it will be extended and the user will be able to choose other types of processing units. The user can also specify whether a waveform should be generated in subsequent simulations, the type of waveform (vcd, fst), a timeout, and whether the Verilator simulator should be used.

- `RunSimulation`
  Sends a set of quantized tensors, i.e., `QTensor` objects, to simulate a given circuit and returns the resulting quantized tensors. It also returns the number of cycles required for the simulation. This information represents the latency of the circuit.

- `DeleteCircuit`
  Deletes a circuit from memory.

```
service Chisel4mlService {
    rpc GetVersion (GetVersionParams) returns (GetVersionReturn) {}
    rpc GenerateCircuit (GenerateCircuitParams) returns (GenerateCircuitReturn) {}
    rpc RunSimulation (RunSimulationParams) returns (RunSimulationReturn) {}
    rpc DeleteCircuit (DeleteCircuitParams) returns (DeleteCircuitReturn) {}
}

message Accelerator {
    string name = 1;
    repeated lbir.LayerWrap layers = 2;
    repeated string options = 3;
    repeated string values = 4;
}

message GenerateCircuitParams {
    repeated Accelerator accelerators = 1;
    bool use_verilator = 2;
    bool gen_waveform = 4;
    uint32 generation_timeout_sec = 5;
    string waveform_type = 6;
}

message GenerateCircuitReturn {
    message ErrorMsg {
        enum ErrorId {
            SUCCESS = 0;
            FAIL = 1;
        }
        ErrorId err_id = 1;
        string msg = 2;
    }
    uint32 circuit_id = 1;
    ErrorMsg err = 2;
}
```

```
35
36 message RunSimulationParams {
37     uint32 circuit_id = 1;
38     repeated lbir.QTensor inputs = 2;
39 }
40
41 message RunSimulationReturn {
42     uint32 consumed_cycles = 1;
43     repeated lbir.QTensor values = 2;
44 }
45
46 message DeleteCircuitParams {
47     uint32 circuit_id = 1;
48 }
49
50 message DeleteCircuitReturn {
51     bool success = 1;
52     string msg = 2;
53 }
54
55 message GetVersionParams {}
56 message GetVersionReturn {
57     string version = 1;
58 }
```

Listing 5.3: Services definition.

5.2.1 Transforming DQNN to LBIR

The DQNN representation is transformed into the LBIR representation by applying a sequence of successively applied transformation functions. Generally these transformations can be divided into the following types:

- **Canonicalization Transformations**
  The goal of canonicalization transformations is to transform the graph representation into a canonical form. This makes it easier to write subsequent transformations, as they can expect their input in standard form. Examples of such transformations are:
  - adding a dummy (zero) bias to convolution operations that have no bias,
  - topological sorting of the graph.
- **Analysis Transformations**
  These transformations analyze the graph and add additional information to the nodes and links. Example of such an analysis include:
  - inferring the shapes of the tensors in the graph,
  - inferring the data layout of the tensors in the graph.
- **Lowering Transformations**
  Lowering transformation due the bulk of the work involved in transforming individual QONNX subgraphs into equivalent LBIR subgraphs. In the first step, all QONNX `Quant` operations are transformed into equivalent LBIR `QTensor` objects. These `QTensor` objects are then also propagated through some operations, such as the tensor reshape operation. After that, all nodes in the graph are traversed to search for a subgraph that is suitable for transformation. The transformation from QONNX to LBIR is successful if all QONNX nodes are successfully transformed into equivalent LBIR nodes.
- **Cleanup Transformations**
  Transformations that remove redundant or unneeded nodes. For example, a transformation that removes identity nodes, such as multiplication by 1.

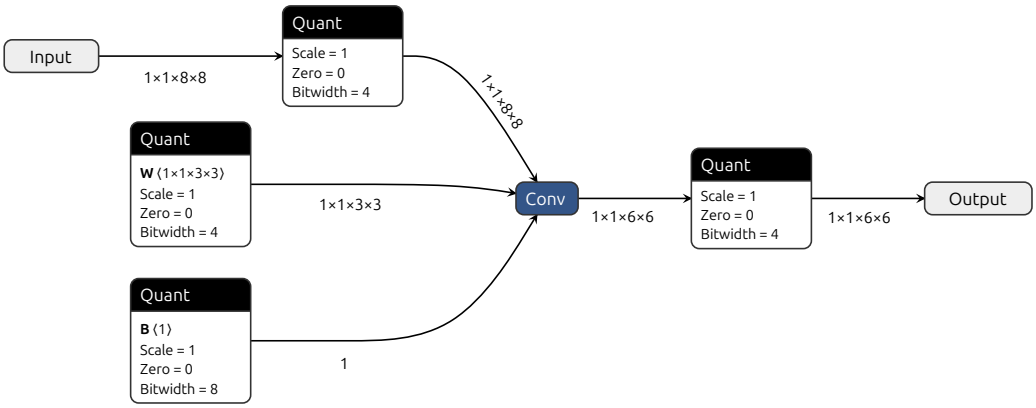


Figure 5.2: A QONNX graph of a quantized convolution layer.

The following examples illustrate how a simple QONNX graph of a quantized convolution layer is transformed step by step into LBIR. The original QONNX graph is shown in Figure 5.2.
Figure 5.3 shows an intermediate step in which the `Quant` nodes have already been transformed into `QTensor` nodes.

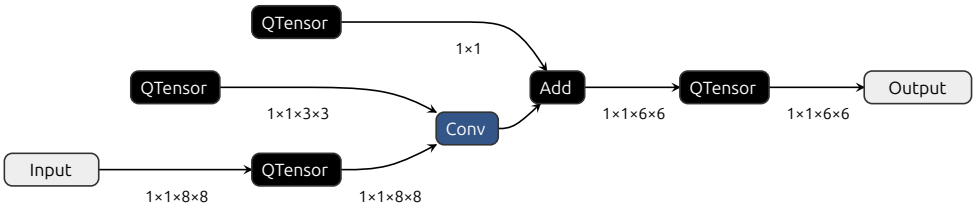


Figure 5.3: Partially transformed QONNX graph with Quant nodes transformed into QTensor nodes.

Lastly, Figure 5.4 shows the final LBIR model, which is a single quantized convolution layer.



Figure 5.4: The final LBIR model.

5.3 Chisel Backend

The Chisel backend provides hardware generation services defined using the gRPC library. Its main task is to generate processing units based on the LBIR model representation. However, other services are also required to manage the generated processing units. A generated circuit is identified by a unique identifier, which is passed to the frontend for reference after generation. This identifier is used by the frontend to drive the simulation of the generated circuit, request the circuit description, delete the circuit from memory, etc.

5.3.1 Representing Quantization Information

Scala, and consequently Chisel, is a strongly typed language. This means that all variables must have known types at compile time, regardless of whether they are specified by the programmer or inferred by the compiler. A strong type system prevents many mistakes, but also makes it more difficult to create functions and classes that can use multiple data types, i.e., generic functions. Fortunately, Scala has many powerful features that allow the programmer to write generic code. We use these to implement a generic interface to higher level functions and classes. This allows us to write these classes and functions in a quantization independent way. The main abstraction used to implement this is the `NeuronCompute` abstract class presented in Listing 5.4. It defines a set of types and functions that are needed to construct a neuron: a scalar multiplication function, a vector summation function, a shifting and rounding function, and the activation function. These functions can be used to implement the computation in Equation 2.8.

```
abstract class NeuronCompute {
  type I <: Bits // input type
  type W <: Bits // weight type
  type M <: Bits // type of multiplication products
  type A <: Bits // type of accumulator and bias
  type O <: Bits // output type

  def mul: (I, W) => M // multiplication of two scalar types
  def addVec: Vec[M] => A // vector addition
  def shiftRound: (A, Int) => A // requantization
  def actFn: (A, A) => O // activation function
}
```

Listing 5.4: The abstract NeuronCompute class and some of its children.

`NeuronCompute` itself offers neither explicit data types nor function implementations that operate on these types and cannot generate hardware structures. In order to provide specific data types and functions that can be translated into corresponding hardware structures, a specialized subclass of `NeuronCompute` must be developed. The specialization of the abstract class `NeuronCompute` is explained with three examples. The following listing shows the `NCSIntSIntUInt` class, which implements the abstract functions and types for a uniformly quantized neuron with signed inputs and weights and unsigned outputs.

```
class NCSIntSIntUInt extends NeuronCompute {
  type I = SInt
  type W = SInt
  type M = SInt
  type A = SInt
  type O = UInt

  def mul = (i: SInt, w: SInt) => i * w // scalar multiplication
  def addVec: Vec[SInt] => SInt = generateAdderTreeFunction // vector sum
  def shiftRound: (SInt, Int) => SInt = signedRequantizeFunction
  def actFn: (SInt, SInt) => UInt = reluFunction
}
```

The `NCBoolBoolBool` class refers to a binarized quantization. As discussed in Section 2.4.3, the weights and activations of binarized neurons use only two values: -1 and +1. These two values can be encoded as 0 and 1, which transforms the multiplication operation into the XNOR operation and the vector addition into a population count operation.

```
class NCBoolBoolBool extends NeuronCompute { // binarized quantization
  type I = Bool
  type W = Bool
  type M = Bool
  type A = UInt
  type O = Bool

  def mul = (i: Bool, w: Bool) => ~(i ^ w) // XNOR operation
  def addVec = (x: Vec[Bool]) => PopCount(x.asUInt) // population count operation
  def shiftRound: (UInt, Int) => UInt = unsignedRequantizeFunction
  def actFn: (UInt, UInt) => Bool = signFunction
}
```

Finally, `NCSIntBoolSInt` refers to binary quantization. In this case, only the weights are restricted to the values -1 and +1, while the inputs are signed integers. This means that instead of a multiplication, a simple multiplexer can be used to compute either the identity or the negation.

```
class NCSIntBoolUInt extends NeuronCompute { // binary quantization
  type I = SInt
  type W = Bool
  type M = SInt
  type A = SInt
  type O = UInt

  def mul = (i: SInt, w: Bool) => Mux(w, i, -i)
  def addVec = (x: Vec[SInt]) => generateAdderTreeFunction
  def shiftRound: (SInt, Int) => SInt = signedRequantizeFunction
  def actFn: (SInt, SInt) => UInt = reluFunction
}
```

5.3.2 A Combinational Neuron Implementation

Because `NeuronCompute` abstracts away the specifics of quantization, the `neuron` function can be implemented in a generic way. In other words, by providing a suitable `NeuronCompute` child object, neurons with different forms of quantization can be implemented. An abridged version of such a function is shown in Listing 5.5. The function takes five arguments:

- `NeuronCompute` - Describes the quantization information needed to implement a neuron.
- `in` - A sequence or list of inputs. These do not represent actual values, but hardware nodes (i.e., a wire instance) of a certain type (e.g., a 3 bit unsigned integer).
- `weights` - A sequence of weight literals. Unlike the `in` argument, the `weights` argument represents actual values that are known at compile time.
- `bias` - A bias literal value known at compile time.
- `shift` - A scale factor logarithm value known at compile time.

```
1 def neuron(
2   nc: NeuronCompute,
3   in: Seq[nc.I],
4   weights: Seq[nc.W],
5   bias: nc.A,
6   shift: Int
7 ): nc.O = {
8   val muls = VecInit((in.zip(weights)).map {
9     case (i, w) => nc.mul(i, w) // perform the multiplications
10  })
11  val pAct = nc.addVec(muls) + bias // add the products and bias
12  val sAct = nc.shiftRound(pAct, shift) // requantize
13  nc.actFn(sAct) // activation function
14 }
15}
```

Listing 5.5: Simplified neuron implementation in Chisel.

The operation of the `neuron` function can be summarized as follows:

- A multiplier is created and connected for each input-weight pair (lines 8-10).
- An adder tree is constructed which sums previously computed products stored in the vector `muls` and `bias` in an accumulator `pAct` (line 11).
- The value of the accumulator is requantized according to the output quantization specifications (line 12).
- Finally, the hardware to implement the activation function is constructed (line 13).

The function in Listing 5.5 is general and can implement all types of quantization. However, in some cases, e.g., binarized quantization, it is more prudent to merge the bias or a computed threshold into the activation function. In this case, chisel4ml uses a slightly different function of a neuron. This follows from the derivation in Equation 2.10.

Listing 5.6 shows an example of the use of the `neuron` function from Listing 5.5. The `ExampleSingleNeuronModule` defines a hardware module with 3 inputs, which are signed 4 bit numbers, and a single unsigned output.

```
class ExampleSingleNeuronModule extends Module {
 val io = IO(new Bundle{
  val in = Input(Vec(3, SInt(4.W))),
  val out = Output(UInt(4.W))
 })
 io.out = neuron(
  nc = new NCUIntSIntUInt(),
  in = io.in,
  weights = Seq(1.S, 2.S, -3.S),
  bias = -1.S,
  shift = -2 // scale factor == 0.25
 )
}
```

Listing 5.6: Using the `neuron` function with 4 bit quantization inside a module.

`ExampleBinarizedNeuronModule` in Listing 5.7 shows an example of the use of the `neuron` function with binarized quantization. This module has 3 binary inputs and a single binary output.

```
class ExampleBinarizedNeuronModule extends Module {
 val io = IO(new Bundle{
  val in = Input(Vec(3, Bool(4.W))),
  val out = Output(Bool())
 })
 io.out = neuron(
  nc = new NCBoolBoolBool(),
  in = io.in,
  weights = Seq(true.B, false.B, true.S),
  bias = 1.U,
  thresh = -1 // scale factor == 0.5
 )
}
```

Listing 5.7: One examples of using the neuron function.

Both examples generate a fully combinational circuit.

5.3.3 Combining Neurons into Layers

To implement the convolution and fully connected layers, many neurons must be combined and connected. In fully connected layers, each input feature is connected to each neuron. In convolution layers, only a subset of the input features is connected to a particular neuron, as described in Section 2.2. The task of connecting the input features to the corresponding neuron is performed by the `NeuronProcessingUnit` (NPU) generator. An abridged version of the NPU implementation is shown in Listing 5.8.

```
class NeuronProcessingUnit(nc: NeuronCompute, layer: lbir.LayerWrap) extends Module {
  val in = IO(Input(Vec(layer.input.numParams, layer.genI)))
  val out = IO(Output(Vec(layer.output.numParams, layer.genO)))

  for (i <- 0 until layer.output.numParams) { // loop over all neurons
    out(i) := neuron(
      nc,
      layer.inMap(in, i), // get subset of in for neuron i
      layer.weightMap(layer.weights, i), // get subset of weights for neuron i
      layer.biasMap(layer.bias, i), // get appropriate bias value for neuron i
      layer.shiftMap(layer.shift, i) // get appropriate scaling value for neuron i
    )
  }
}
```

Listing 5.8: The Neuron Processing Unit.

The arguments to the `NeuronProcessingUnit` class are a `NeuronCompute` object, which provides the quantization information, and a `lbir.LayerWrap` object, which provides the weights and structural information about the layer. The `lbir.LayerWrap` object represents either a convolution layer or a fully connected layer.

The `NeuronProcessingUnit` (NPU) module has a vector (sequence) of inputs and a vector of outputs. The length of the input vector corresponds to the number of variables in the input tensor. Similarly, the output vector has the same number of elements as the output tensor.

For each neuron or output, the NPU selects the correct subset of the inputs `in`, the appropriate weight parameters from `layer.weights`, an appropriate bias value from `layer.bias`, and the appropriate scale factor from `layer.shift`. This is done using the `inMap`, `weightMap`, `biasMap`, and `shiftMap` functions. The `biasMap` and `shiftMap` functions both return a scalar value that represents the bias value and the scale factor of a particular neuron. The `inMap` and `weightMap` functions return a sequence that represent subsets of the inputs and weights.

For a fully connected layer, `inMap` simply returns the entire set of inputs. For a convolution layer, however, the `inMap` function returns a subset of the inputs corresponding to the local receptive field. This is illustrated in Figure 5.5. For the fully connected layer in Figure 5.5a, all four inputs are mapped to both neurons. In contrast, for the convolution layer in Figure 5.5b, only a subset of the inputs is mapped to each neuron. In Figure 5.5b, only two of the four neurons are highlighted for illustrative purposes.

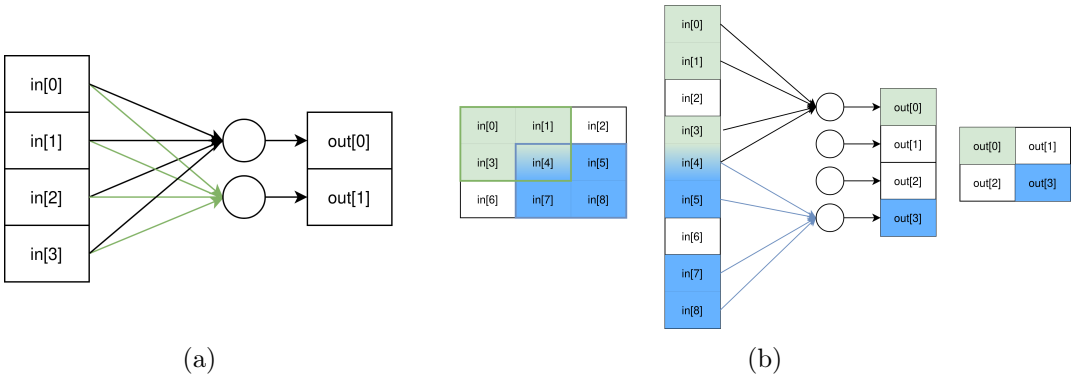


Figure 5.5: Mapping the inputs to the appropriate neurons for a fully connected layer in (a) and a two-dimensional convolution layer in (b).

Since both convolution and maximum pooling layers have similar sliding window behavior, the logic in `inMap` can be implemented by a single function. This function allows us to write the sliding window procedure only once. This is not trivial, as there are many parameters and corner cases to consider. This function takes many parameters into account:

- The **tensor layout** must be considered.
- The **input window size**, which can be non-square.
- The **kernel size**, which can be non-square.
- The **stride**, which can be different in the width and height direction.
- The **padding**, which can differ at the top, bottom, left or right.
- The **grouping** of the channels, e.g., to implement depthwise convolution.

5.3.4 Implementing Maximum Pooling Layers

The chisel4ml tool supports generation of maximum pooling layers. While the maximum pooling layer has no weights, biasses, and scale factors, the input feature window for each output are selected similarly to NPU, as mentioned earlier. In contrast to NPU, a generic maximum pooling operation is used instead of a neuron operation.

The generic operations used in the maximum pooling layer are encapsulated in an abstract `OrderCompute` class given in Listing 5.9.

```
abstract class OrderCompute {
  type T <: Bits
  def gte: (T, T) => Bool
}
```

Listing 5.9: An abstract class that defines ordering on a type.

The `OrderCompute` class is equivalent to the `NeuronCompute` class for the neuron function. It defines the required function for performing comparisons, which is necessary for implementing the maximum pooling operation. Listing 5.10 shows an example implementation of the `OrderCompute` class for signed integers.

```
class SIntOrderComputable(val bitwidth: Int) extends OrderCompute {
  type T = SInt
  def gte = (x: SInt, y: SInt) => x >= y
}
```

Listing 5.10: A concrete implementation of `OrderCompute` for signed integers.

For binarized numbers, the ordering can be defined as a binary function. This is shown in Listing 5.11.

```
object BoolOrderComputable extends OrderCompute {
  type T = Bool
  def gte = (x: Bool, _: Bool) => x
}
```

Listing 5.11: A concrete implementation of `OrderCompute` for booleans.

The abstract `OrderCompute` class can be used to define a generic maximum pooling function `maxpool`. It generates a combinational circuit that performs the maximum pooling operation on an input tensor and returns the largest value that is a scalar. This is analogous to the generic `neuron` function. An abridged version of the `maxpool` function is given in Listing 5.12.

```
def maxpool(oc: OrderCompute, in: Seq[oc.T]): oc.T {
   VecInit(in).reduceTree((x, y) => max(oc, x, y))
}
```

Listing 5.12: A generic maximum pooling function.

The maximum pooling function `maxpool` uses a helper function `max` that compares two values. This function is given in Listing 5.13. Note that there is no return statement in Scala. Instead, the last line of the function is the returned value.

```
def max(oc: OrderCompute, x: oc.T, y: oc.T): oc.T {
   val out = Wire(chiselTypeOf(x))
   when (oc.gte(x, y)) {
      out := x
   } otherwise {
      out := y
   }
   out
}
```

Listing 5.13: A generic implementation that compare two values.

The `OrderProcessingUnit` generator is similar to the `NeuronProcessingUnit` generator, with the exception that it uses a maximum pooling function `maxpool` instead of a neuron function `neuron`. The `OrderProcessingUnit` generator reuses the mapping function

used in the `NeuronProcessingUnit`. An abridged version of the `OrderProcessingUnit` generator is given in Listing 5.14.

```
class OrderProcessingUnit(oc: OrderCompute, layer: lbir.MaxPool2DConfig) extends Module {
  val in = IO(Input(Vec(layer.input.numParams, layer.genI)))
  val out = IO(Output(Vec(layer.output.numParams, layer.genO)))

  for (i <- 0 until layer.output.numParams) {
    out(i) := maxpool(oc, layer.inMap(in, i))
  }
}
```

Listing 5.14: The OrderProcessingUnit.

5.3.5 Generating the Entire Circuit

The generators described above create the hardware structure for a single layer of the neural network. To create the hardware structure for the entire neural network, the generated layers are connected together sequentially. Listing 5.15 shows an abridged code of the `ProcessingPipelineCombinational` generator, which implements a sequence of layers.

```
1  class ProcessingPipelineCombinational(layers: Seq[LayerWrap]) extends Module {
2    // Instantiate modules for separate layers
3    val peList: Seq[Module] = layers.map { l: LayerWrap match =>
4      case l: MaxPool2DConfig => Module(new OrderProcessingUnit(OrderCompute(l))(l))
5      case l: Conv2DConfig => Module(new NeuronProcessingUnit(NeuronCompute(l))(l))
6      case l: DenseConfig => Module(new NeuronProcessingUnit(NeuronCompute(l))(l))
7    }
8    val in = IO(Input(chiselTypeOf(peList.head.in)))
9    val out = IO(Output(chiselTypeOf(peList.last.out)))
10
11   // Connect the inputs and outputs of the layers
12   peList.head.in := in
13   for (i <- 1 until layers.length) {
14     peList(i).in := peList(i - 1).out
15   }
16   out := peList.last.out
17 }
```

Listing 5.15: The combinational processing pipeline.

First, the corresponding modules for the individual layers are instantiated in lines 3–7. Next, the input-output interface is created by simply copying the interface of the first and last layer. Finally, the input and output interfaces of the processing units are connected in a for loop in lines 12–16.

The `ProcessingPipelineCombinational` generator produce fully combinational circuit of a neural network. This results in very long logic paths, which lead to a very large path delay in the synthesized circuit and thus to a low achievable clock rate. To increase the maximum achievable clock rate, chisel4ml inserts pipeline registers at the input of the circuit. These pipeline registers are then distributed throughout the entire circuit by the synthesis tools using the register retiming synthesis option.

The circuits produced by the generators described above have a highly parallel interface that is not suitable for integration into larger systems. To facilitate such integration, chisel4ml wraps the generated circuit with an AXI-Stream style interface, thus the generated circuit can easily be integrated into a larger system.

Chapter 6

Results

The performance of our chisel4ml tool was compared with hls4ml and FINN on a wide set of experiments. The experiments were conducted by first creating a QONNX model. This model was then fed into the three tools to obtain a Verilog implementation. Finally, synthesis was performed. The results from the synthesis phase form the basis for the comparison in this thesis. The following experiments were carried out:

- **Individual layer experiments**
  Single layer models of the fully connected, convolution and maximum pooling layers were generated randomly. In each experiment a single attribute of a particular layer was varied, and the three tools were ran, to see how that attribute affects the results.
- **Jet tagging**
  A series of multilayer perceptron models were trained for the task of jet tagging. Two experiments were conducted in this category. In the first the quantization bitwidth was varied and in the second the global prune rate was varied.
- **Digit recognition**
  A series of convolutional neural network models was trained for the MNIST dataset, a digit recognition dataset. Two experiments were conducted in this category. In the first the quantization bitwidth was varied and in the second the prune rate.

In the experiments the performance of chisel4ml was compared to performance of FINN v0.10.1 and hls4ml v1.0.0+. The hls4ml v1.0.0+ version is not a officially released version of hls4ml. Because of some difficulties with version v1.0.0, a slightly newer but unreleased version of hls4ml was used. Both FINN and hls4ml used Vitis 2023.1 as the backend high-level synthesis tool. Logical synthesis was in all cases performed using Vivado 2023.1. The experiments were conducted on a high-performance computer with 1 TiB of DRAM memory and a 48 core Intel Xeon CPU E5-2680 v3 processor running at a 2.5 GHz frequency. An overview of the workflow of the experiments is given in Figure 6.1.

The following metrics were collected for each experimental run:

- **Circuit Resource Usage**
  Using the reports generated by Vivado the amount of look-up tables and flip-flops used by each circuit was measured. Additionally, the amount of BlockRAM, UltraRAM, and DSP blocks used by each circuit is provided.
- **Circuit Performance**
  - **Path Delay**
    The largest path delay in the generated circuit in nanoseconds. Inverse of this represents the maximum achievable frequency of the circuit.

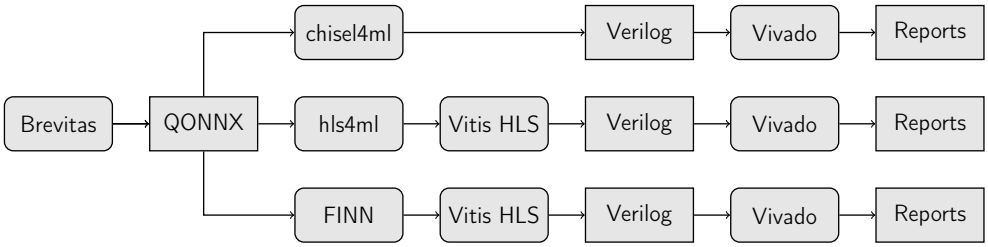


Figure 6.1: The workflow of the experiments.

  - **Cycle Latency**
    The number of cycles it takes for the circuit to compute a single input sample.
  - **Total Latency**
    The time it takes for the circuit to compute a single input sample in nanoseconds. Note that the plots for the total latency are on a logarithmic scale.
  - **Initiation Interval**
    The maximum number of clock cycles required to process a data sample at each pipeline stage. This limits the input data rate. This metric is not available for FINN since FINN configuration does not support it.
  - **Throughput**
    The throughput of the circuit is provided in number of samples per second. It is calculated based on the path delay and initiation interval using the formula: $Throughput = \frac{1}{Path\ Delay \cdot Initiation\ Interval}$. Since the FINN tool does not provide the initiation intervals, the throughput was measured using RTL simulation in this case.

- **Tool Performance**
  - **Generation time**
    The time needed to generate a synthesized Verilog netlist from a QONNX model. It is reported in hours.
  - **Peak Memory Usage**
    The peak memory consumption of the tool used to generate a given circuit is measured. It is reported in MiB.

For brevity only the results on the number of utilized look-up tables, the total latency, throughput and the generation time are provided in the main text. The full set of results is provided in the appendix. In this chapter the number of utilized flip-flops is not provided. This is because this work addresses massively parallel circuits, which primarily uses logic components (LUTs) and flip-flops are mainly used as pipeline register. The experiments targeted the FPGA platform where there is one flip-flop per each LUT and only a small fraction of available flip-flops were utilized. However, the hardware structures (circuits) generated by HLS-based tools utilize significantly more flip-flops than circuits generated by chisel4ml tool.

The chisel4ml tool generates maximally parallel, i.e., direct logic, circuits. To enable fair comparisons, both hls4ml and FINN tools were configured to generate maximally parallel circuits. This is described in Section 4 in more in more detail.

The instructions to reproduce the experiments are available at `https://github.com/jurevreca12/c4ml_test_runs` (accessed on 8 July 2025). The repository also contains all the details on versions used, configuration settings, as well as the result available in a form of spreadsheet documents.

6.1 Individual Layer Experiments

A set of individual layer experiments was performed, where the entire model consisted of a single fully connected, convolution, or maximum pooling layer. The weight parameters of these layers were generated randomly. In each experiment a single layer attribute was varied, e.g., the size of the input, while keeping the others constant. In this way the effect of that attribute on the generated circuits was studied.

6.1.1 Fully Connected Layer

Four experiments with the fully connected layer were performed in which the following layer attributes were varied: the number of **input features**, the number of **output features**, i.e., the number of neurons, the number of bits used to quantize the **input**, the number of bits used to quantize the **weights**.

In each experiment the other attributes are all held constant. The results of the experiments on the fully connected layer are given in Figures 6.2, 6.3, 6.4, and 6.5. In the four experiments a total of 26 different configurations of the fully connected layer were evaluated and thus 26 different circuits were generated by each tool.

Number of look-up tables
Based on the results presented in the figures it can be concluded that fully connected layer implementations of chisel4ml and hls4ml are roughly equivalent in terms of number of look-up tables required for implementation. FINN on average uses more look up tables, specially at higher input bitwidths.

Total latency
In terms of total latency hls4ml is better than chisel4ml in every tested configuration of the fully connected layer. These results are constituent on the cycle latency and the path delay of the circuits. The cycle latency of both chisel4ml and hls4ml circuits is similar, however, hls4ml circuit achieve a substantially better path delay. Upon inspection of the generated circuits, we noticed that some circuits generated by hls4ml do not contain any multipliers, instead the multiplications were performed using a series of adders. Thus, we believe that the lower path delay was achieved using arithmetic optimizations performed by the high-level synthesis tool.

Throughput
All three tools are able to generate circuits with a very high throughput, that can satisfy the needs of most applications. However, hls4ml designs do have consistently better throughput than chisel4ml or FINN designs. The reason for this is the aggressive pipelining in the hls4ml designs. Similar results could be achieved with chisel4ml by increasing the number of pipeline stages.

Generation time
The generation time of the circuits, however, differs significantly by tool. FINN takes up to an order of magnitude longer than chisel4ml to generate the circuits. While hls4ml hovers close to chisel4ml at smaller circuit sizes it diverges significantly at larger sizes. This is expected as high-level synthesis is intractable in general and practical implementations are based on polynomial algorithms, as explained in Section 3.2. Even though hls4ml and FINN are both based on high-level synthesis, there performance differs significantly. We had not been able to deduce the reason for this discrepancy.

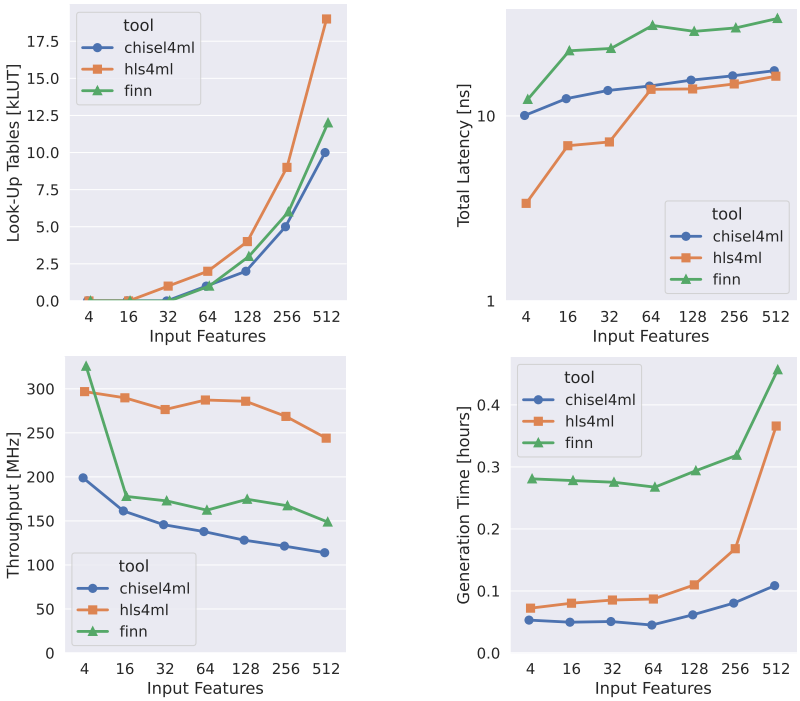


Figure 6.2: Single fully connected layer experiment in which the number of input features is varied. The number of output features equals 32. The input, output and weights are quantized to 4 bits and the bias is quantized to 8 bits.

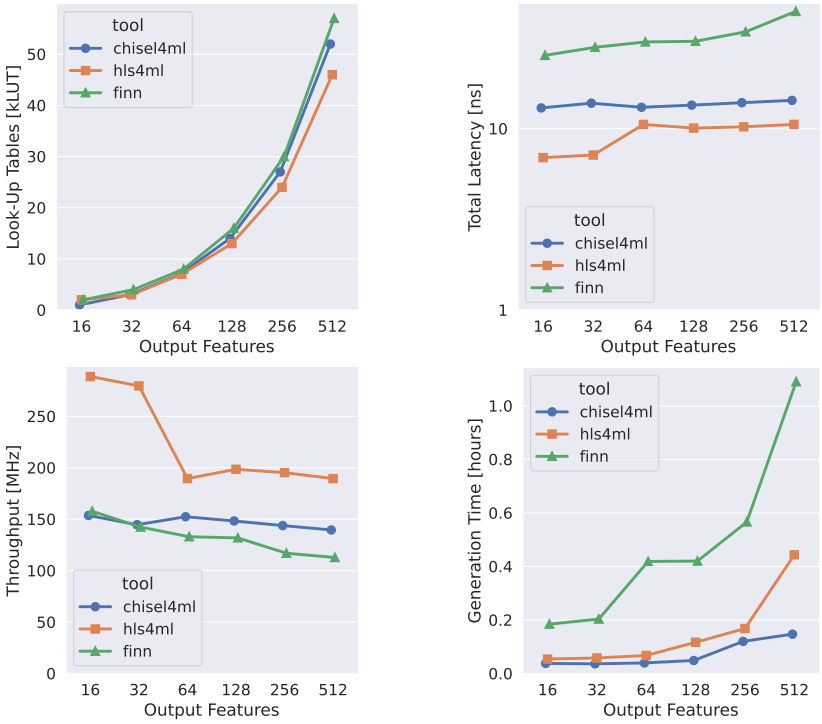


Figure 6.3: Single fully connected layer experiment in which the number of output features is varied. The number of input features equals 16. The input, output and weights are quantized to 4 bits and the bias is quantized to 8 bits.

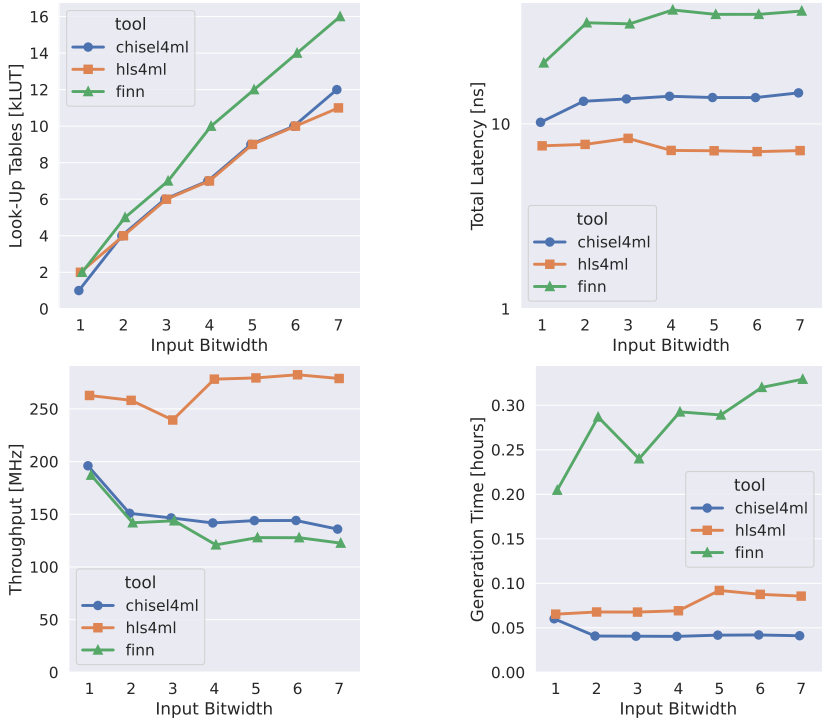


Figure 6.4: Single fully connected layer experiment in which the number of bits used to quantize the input features is varied. The number of input and output features equals 32. The weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits.

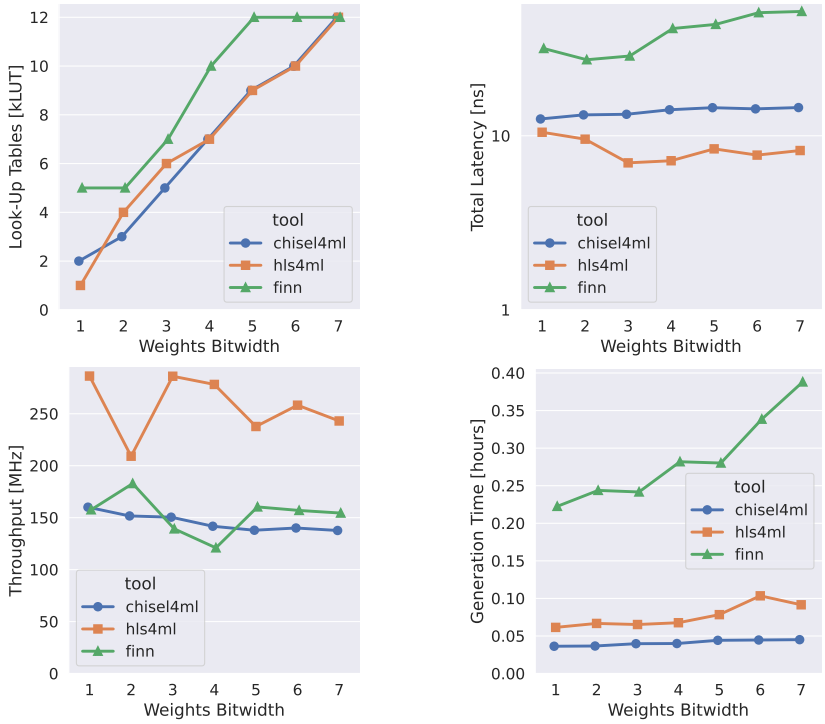


Figure 6.5: Single fully connected layer experiment in which the number of bits used to quantize the weight parameters is varied. The number of input and output features equals 32. The inputs and outputs are quantized to 4 bits and the bias is quantized to 8 bits.

6.1.2 Convolution Layer

Four experiments were performed with the convolution layer. In these experiments the following attributes of the convolution layer were varied: the number of **input channels**, the number of **output channels**, the number of bits used to quantize the **input**, the number of bits used to quantize the **weights**.

In each experiment the other attributes are held constant. Figures 6.6, 6.7, 6.8, and 6.9 show the results of the experiments on the convolution layer. As previously stated, the weight parameters of the convolution layer were generated randomly. In the four experiments a total of 24 configurations of the convolution layer were evaluated and each tool generated 24 different circuits. Unlike for the fully connected layer, the convolution layer results differ quite significantly by the tool used. Because convolution layer typically deals with higher-dimensional data, which typically results in a higher number of inputs, the circuits generated in these set of experiments are much larger.

Number of look-up tables
The chisel4ml tool generates maximally parallel, i.e., a direct logic circuits and hence require a substantial amount of look-up tables. hls4ml tool was configured to generate maximally parallel circuits as well and similar utilization of LUTs was expected. However, a hls4ml generated circuits requires notably more hardware resources than chisel4ml generated circuits. We have not been able to find the reason for this discrepancy. FINN on the other hand requires far fewer look-up tables than either chisel4ml or hls4ml. This is due to the fact that it does not fully unroll the computation, and instead computes the convolution a single neuron (pixel) at a time. This, however, has implications on the latency and throughput achieved by FINN circuits.

Total latency
The total latency results mirror the number of cycles required by the circuits to compute. As chisel4ml uses a single pipeline stage per layer, it achieves a very low total latency. Because FINN does not fully unroll the convolution layer, the circuits generated by FINN have a high cycle latency and thus also a high total latency. Somewhat surprisingly, the circuits produces by hls4ml also have a high total latency, in some cases significantly worse than even the circuits from FINN. The reason for this is again a high cycle latency. However, as the circuits generated by hls4ml are highly parallel, this should not be the case.

Throughput
Neither hls4ml nor FINN generated circuits are able to achieve a high throughput on the convolution layer models. This is the result of a high cycle latency and a similarly high initiation interval. This is expected for circuits generated with the FINN tool, as FINN does not fully unroll the convolution layer. However, the lower throughput of the hls4ml generated circuits was surprising. We were unable to explain why hls4ml circuits had such a high initiation interval, however, the high initiation interval seems to be less of a problem when the convolution layer is part of a larger neural network, as we will see Section 6.3.

Generation time
As mentioned at the beginning of this subsection, the convolution layer is typically used on higher-dimensional data with larger number of features, and thus the generated circuits are generally larger. A result of this is a very high generation time by hls4ml. FINN is able to achieve a relatively constant generation time because it does not fully parallelize the computation. Instead, the layer outputs are computed sequentially. chisel4ml on the other hand is able to generate direct logic circuit implementations in linear time. We attribute this to the fact that it is based on structural descriptions of the circuits, opposed to the high-level synthesis based approach used by hls4ml.

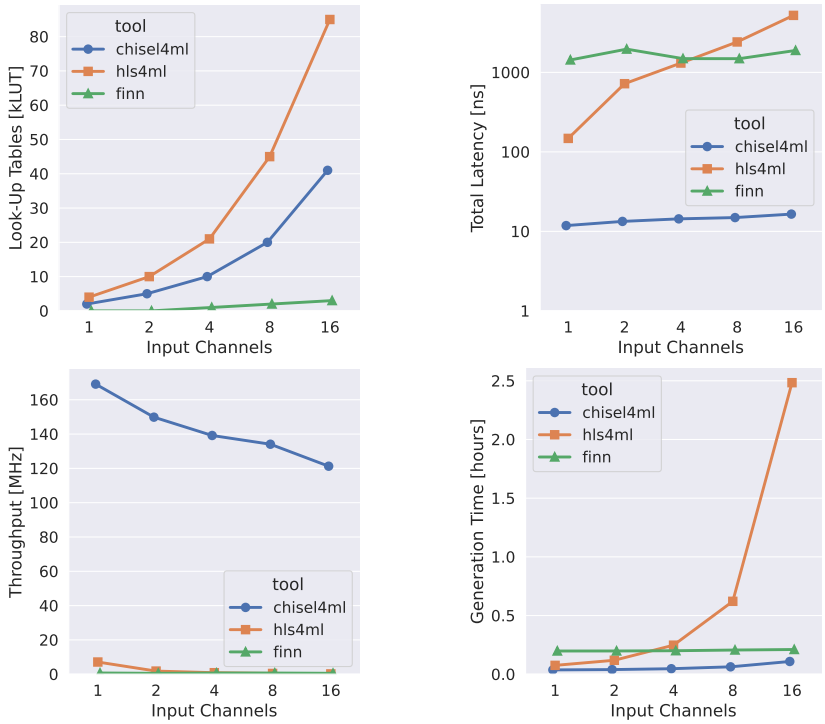


Figure 6.6: Single convolution layer experiment in which we vary the number of input channels of the convolution layer. The input size is set to an 8 by 8 square and the kernel size is set to 3 by 3. The number of output channels is set to 1. The inputs, weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits.

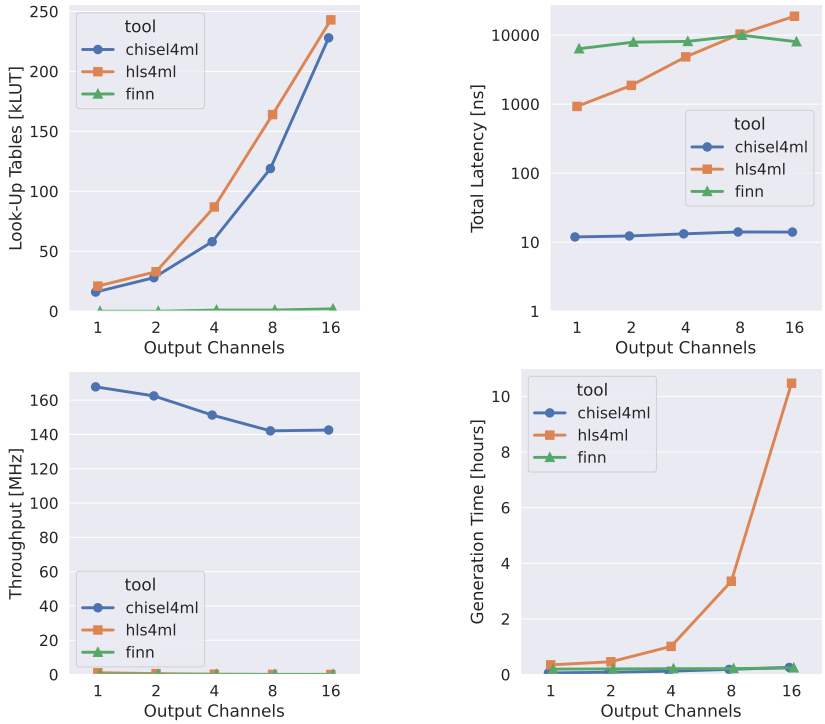


Figure 6.7: Single convolution layer experiment in which the number of output channels of the convolution layer is varied. The input size is set to an 16 by 16 square and the kernel size is set to 3 by 3. The number of input channels is set to 1. The inputs, weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits.

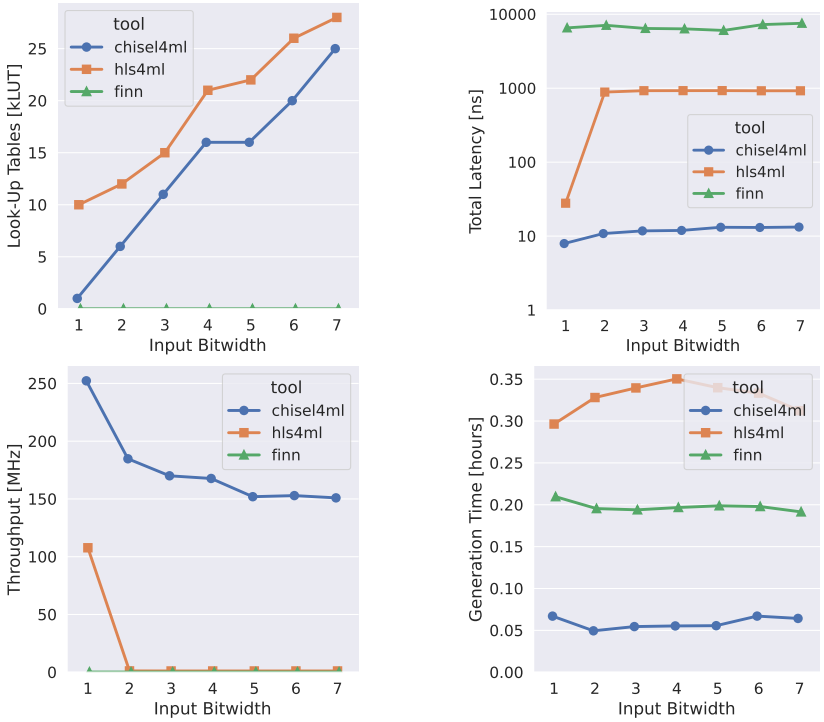


Figure 6.8: Single convolution layer experiment in which the number of bits used to quantize the input is varied. The input size is set to an 16 by 16 square and the kernel size is set to 3 by 3. The number of input and output channels is set to 1. The weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits.

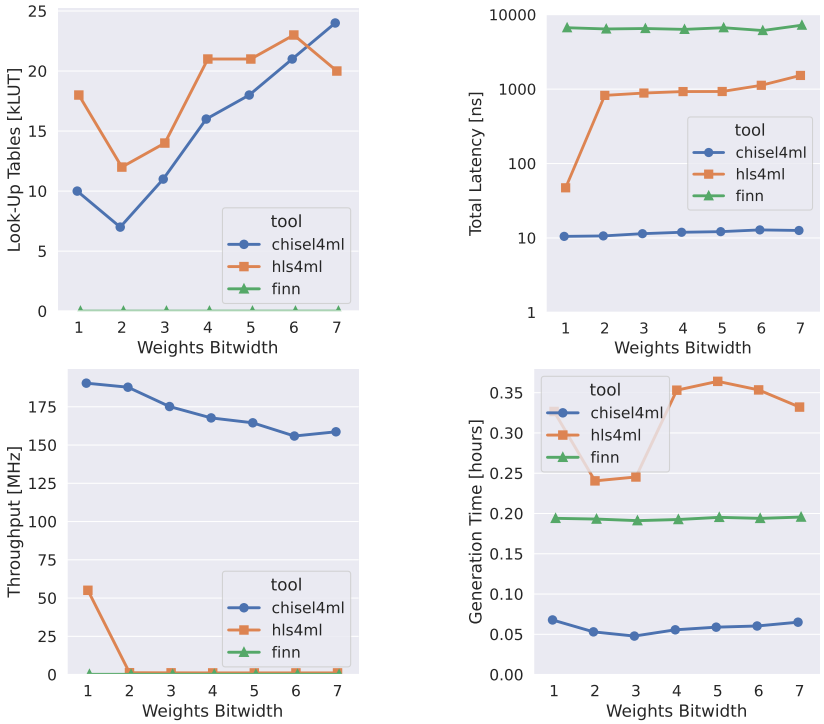


Figure 6.9: Single convolution layer experiment in which the number of bits used to quantize the weight parameters is varied. The input size is set to an 16 by 16 square and the kernel size is set to 3 by 3. The number of input and output channels is set to 1. The inputs and outputs are quantized to 4 bits, and the bias is quantized to 8 bits.

6.1.3 Maximum Pooling Layer

Four experiments with the maximum pooling layer were performed. In these experiments the following attributes of the maximum pooling layer were varied: the size of the square **input window**, the number of **input channels**, the size of the square **kernel**, and the bitwidth of the **input** data.

In each experiment the other attributes are held constant. The results of the maximum pooling layer experiments are given by Figures 6.11, 6.12, 6.13, and 6.14. In the four experiments a total of 22 configurations of the maximum pooling layer was evaluated and each tool generated 22 different circuits.

Number of look-up tables
Circuits implementing the maximum pooling layer generated by hls4ml utilize significantly more look-up tables than those generated by either chisel4ml or FINN. The stark difference between hls4ml and FINN is expected as FINN does not fully parallelize the computation in a maximum pooling layer. However, the difference between the look-up tables used by chisel4ml and hls4ml is surprising as both should generate circuits that are maximally parallel. High parallelization is in fact achieved, as is evident from the results on total latency and throughput. A possible reason for the large look-up table count by hls4ml circuits is the fact that they seem to use linear style reductions, as opposed to tree style reductions that are used in chisel4ml. This results in a higher growth of the bitwidths of the intermediate results. This differences are illustrated in Figure 6.10

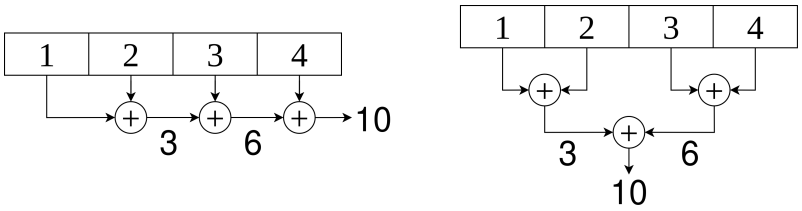


Figure 6.10: Linear versus tree style reduction.

Total latency and throughput
Both chisel4ml and hls4ml generate highly parallel implementations of the maximum pooling layer. Thus they both have a very low total latency and high throughput. In general hls4ml is able to achieve a higher throughput, but this is again constituent on the lower path delay of the hls4ml circuits. The same throughput could likely be achieved by with chisel4ml by fine tuning the number of pipeline stages. FINN computes pooling layers in a similar fashion to convolution layers. That is it does not fully parallelize them. Thus the results for FINN are similar to the results for the convolution layer, as FINN generated circuits are not able to achieve a high throughput or low total latency.

Generation time
The generation time of chisel4ml and FINN circuits is proportional to the input size. For FINN this is because of not parallelizing the circuit with the size of inputs, and for chisel4ml it is because it is based on structural descriptions of the circuits. On the other hand, the hls4ml generation time increases significantly with input size.

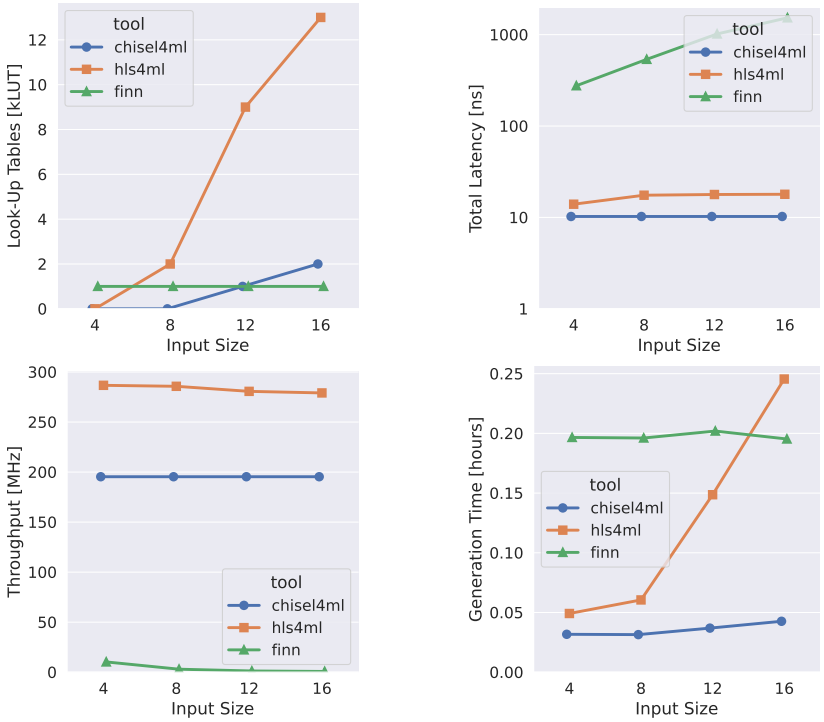


Figure 6.11: Single maximum pooling layer experiment in which the input size is varied whilst keeping it square. The size of the kernel is 3 by 3 and the number of channels is set to 3. The inputs to the layer are quantized to 4 bits.

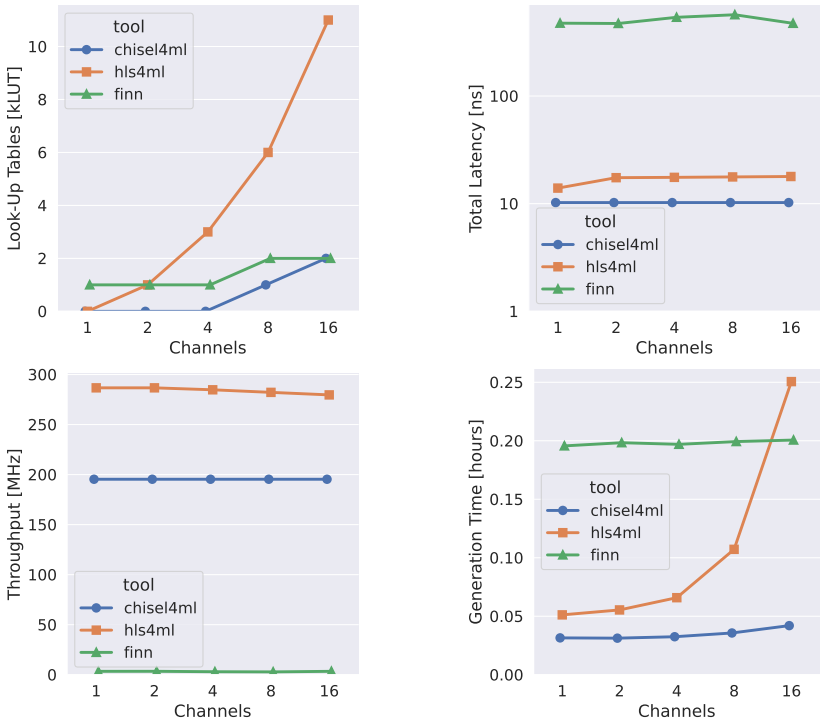


Figure 6.12: Single maximum pooling layer experiment in which the number of channels of the input tensor is varied. The input size is set to an 8 by 8 square and the kernel size is set to 3 by 3. The inputs to the layer are quantized to 4 bits.

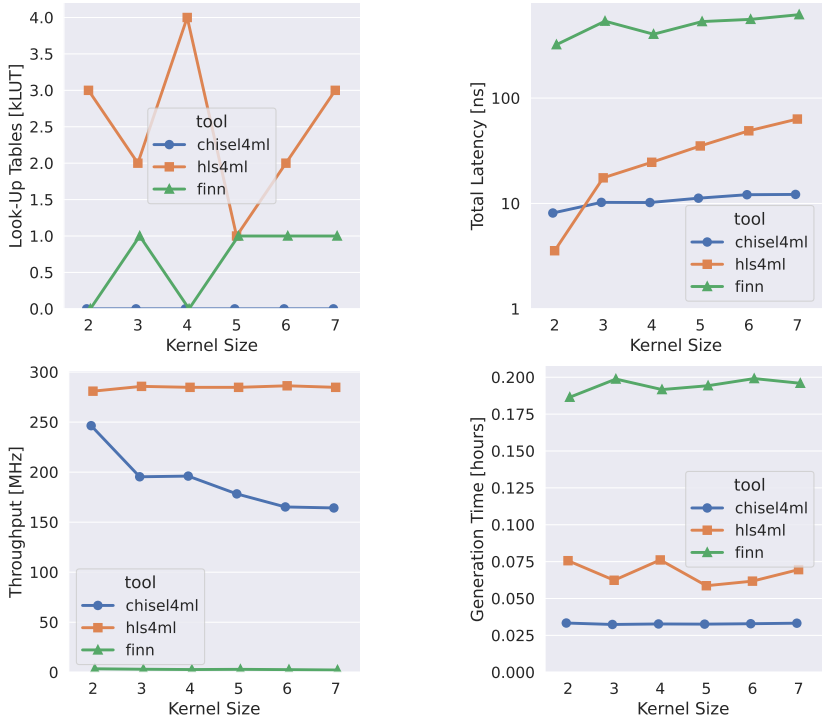


Figure 6.13: Single maximum pooling layer experiment in which the size of the square kernel is varied. The number of channels is set to 3 and input size to an 8 by 8 square. The inputs to the layer are quantized to 4 bits.

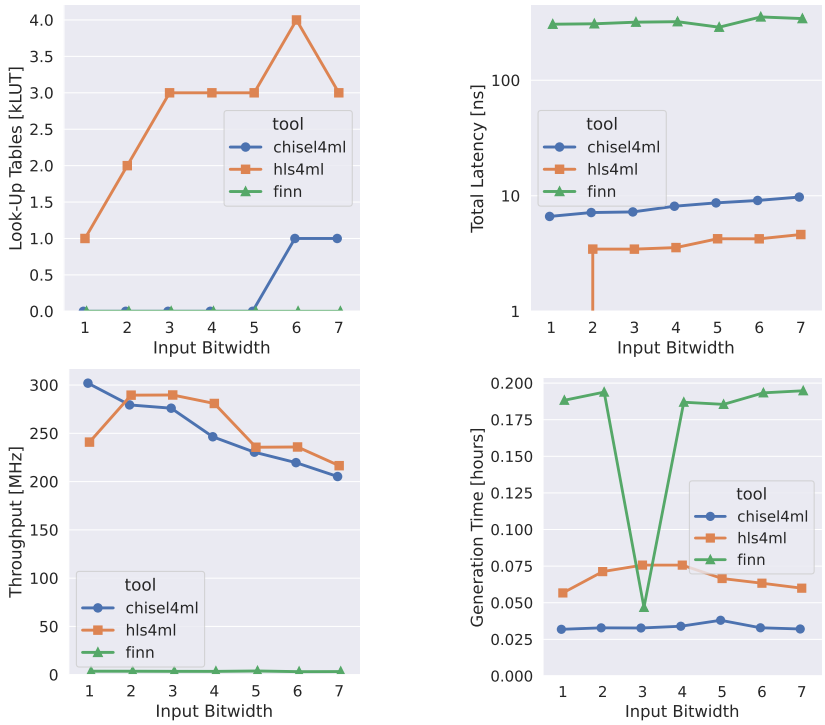


Figure 6.14: Single maximum pooling layer experiment in which the number of bits used to quantize the input is varied. The number of channels is set to 3 and the input size to an 8 by 8 square. The kernel size is set to 2 by 2.

6.2 Jet Tagging Multilayer Perceptron

The experimental physics community is a frequent user of neural network inference systems with ultra-low latency requirements. Jet tagging is a classification task that identify jets arising from particles of high interest, e.g., quark, gluon, the W or Z boson, or the top quark [2]. A set of MLP models were trained on the *hls4ml_lhc_jets_hlf* dataset. Each of the 830.000 instances in the dataset has 16 features of the jet. The dataset was split randomly into a training dataset representing 70 % of the data and the test dataset representing the other 30 % of the data.

Two experiments were performed in this set of experiments. In the first experiment a constant prune rate of 0.5 was set and the bitwidth of the weight parameters was varied in conjunction with the bitwidth of the activations. In the second experiment the weight parameters and activations were set to a constant bitwidth of 4 and the prune rate was varied. In the two experiments a total of 12 MLP models were trained for the task of jet tagging and each tool generated 12 different circuits.

The topology of the neural networks are shown in Table 6.1. After every fully connected layer a batch normalization layer is also inserted. The input to the first layer is always quantized to 8 bits. The bias values are all quantized to 4 bits. Weights and activations on the other hand are all quantized according to to the bitwidth specified by the experiment. All tensors in the models have a single scalar scale factor that is restricted to be a power of two. In the experiment in which the bitwidth was varied, the bitwidth of 1 results in a binarized model. In this case the ReLU activations are replaced with sign functions. A global unstructured pruning was applied to all models, with the level of pruning being determined by the parameters of the experiment.

Table 6.1: Topology of multilayer perceptron models after folding the batch normalization layers.

Layer	Input Size	Output Size	Activation
Fully Connected	16	64	ReLU
Fully Connected	64	32	ReLU
Fully Connected	32	32	ReLU
Fully Connected	32	5	-

All models were first trained for a total of 10 epochs. After 5 epochs the batch normalization layers were folded into the preceding layer, and another 5 epochs of training were performed. The optimization procedure was performed using the Adam optimizer [12], and a learning rate of 0.001. For the first five epochs of training the pruning was not applied in any case. In final five epochs the prune rate was gradually increased towards the desired final prune rate using the formula:

$$effective_prune_rate = max(0, desired_prune_rate \cdot \frac{current_epoch + 1 - 5}{5}).$$

Thus for a desired prune rate of 0.5 we first have 5 epochs of 0.0 effective pruning rate, and then it gradually increases by 0.1 reaching 0.5 in epoch 10. The accuracies obtained by the models on the test dataset are plotted in Figure 6.15. A floating-point model without pruning was also trained using the same procedure and obtained an accuracy of 0.76.

Number of look-up tables

The results of the experiments on the MLP models are shown in Figures 6.16 and 6.17. The results largely mirror the results obtained for single fully connected layer models. The number of look-up tables used increases with bitwidth and decreases with the increasing prune rate in all tools as expected. chisel4ml and hls4ml achieve similar numbers of look-up

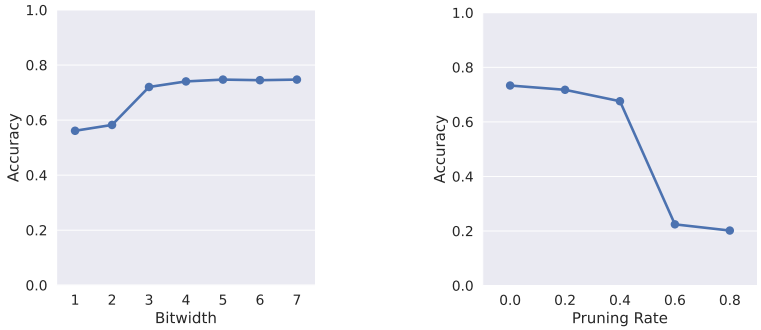


Figure 6.15: Accuracies of the MLP models.

tables, while FINN designs use more look-up tables at higher bitwidths and lower prune rates. The reason FINN uses approximately equal numbers of look-up tables to chisel4ml and FINN at lower bitwidths, but proportionally more look-up tables at higher bitwidths is unknown. However, it the same behavior as in the case of a single fully connected layer, as shown in Figures 6.4 and 6.5. These figures show the experiments on a single fully connected layer on the number of bits used to quantized inputs and weights.

Total latency

The best total latencies are achieved by hls4ml circuits. chisel4ml circuits have a higher total latency than hls4ml, except for the binarized case. This is due to the path delay of chisel4ml circuits being greater than the path delay of hls4ml circuits. As mentioned perviously this is likely the result of arithmetic optimization performed by the HLS tool. FINN circuits have the worst total latency of all three tools. The total latencies of FINNs circuits increase significantly with bitwidth. We did not identify any reason why FINN generated circuits perform worse in this regard. This discrepancy could likely be mitigated in FINN by setting more aggressive timing targets. The total latency of chisel4ml and hls4ml circuits at the prune rate of 0.8 is so low that it does not represent achievable total latency. This is caused by very low reported path delay after logic synthesis. We removed such data points from the total latency plot.

Throughput

The throughput of hls4ml designs is far greater than either chisel4ml or FINN. In large part this is because hls4ml circuits introduce a lot of pipeline stages and achieve achieve a lower path delay. chisel4ml designs could achieve similar results, if the number of pipeline stages was properly tuned. Similarly, FINN could also be configured to produce circuits with a lower path delay and thus a higher throughput. In general the circuits of MLP models generated by all three tools are maximally parallel and are able to achieve a very high throughput of over 100 MHz, which should be satisfactory for a wide number of applications. The throughput graph of Figure 6.17 shows a sharp increase in throughput for the chisel4ml circuit at a prune rate of 0.8. This is an anomaly. The highly pruned circuit does achieve a very low path delay after logic synthesis. However, such throughput and low path delay would definitely not be achievable in any real FPGA.

Generation time

The generation time of hls4ml and chisel4ml is roughly equivalent, whilst FINN generation time is substantially greater, especially for higher bitwidths and lower prune rates. The reason chisel4ml and hls4ml have approximately equal generation time is because the tested MLP models in these experiments are relatively small and they had not yet reached the threshold at which high-level synthesis requires a substantially longer time, as shown in the single fully connected layer experiment in Figure 6.2. FINN has a significantly larger generation time than either hls4ml or chisel4ml. This is in line with the results of the

testing performed on single fully connected layer models.

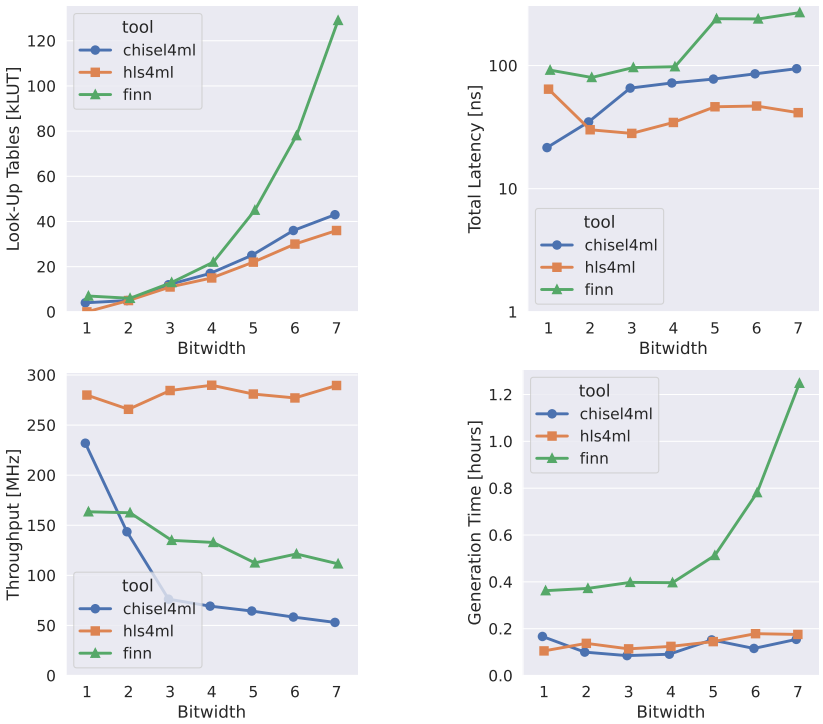


Figure 6.16: The multilayer perceptron experiment in which the number of bits used to quantize the weights and activations is varied. The inputs to the MLP are quantized to 8 bits. The global prune rate is set to 0.5.

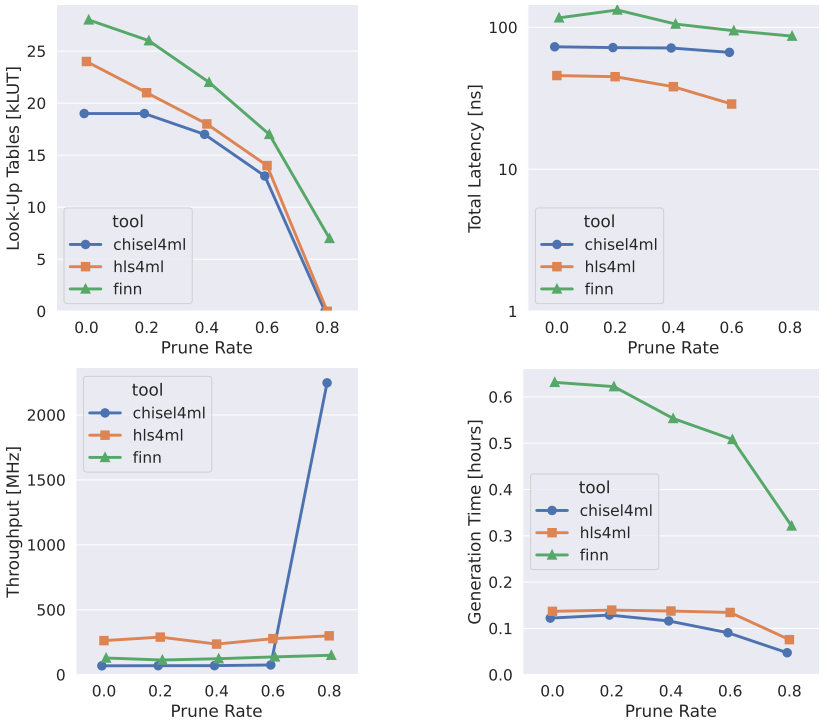


Figure 6.17: The multilayer perceptron experiment in which the global prune rate is varied. The weights and activations are quantized to 4 bits. The inputs to the MLP are quantized to constant 8 bits.

6.3 MNIST Digit Recognition

A series of convolutional neural networks was trained on the MNIST digit recognition dataset [44]. It contains 70.000 28 by 28 grayscale images of hand written digits. The task is to classify the images as one of the decimal digits. The dataset is split into 60.000 training images and 10.000 test images. The relatively high dimensionality of the data will illustrate how the tools handle larger problems.

Two experiments were performed in this set of experiments. In the first experiment a constant prune rate of 0.7 was set and the bitwidth of the weight parameters was varied in conjunction with the bitwidth of the activations. In the second experiment the weight parameters and activations were set to a constant bitwidth of 4 and the prune rate was varied. In total 12 CNN models were trained for the task of hand written digit classification. Some tools were not able to generate circuits for all 12 CNN models. The binarized models were troublesome for both hls4ml and FINN because internal transformations do not yet fully support convolutional binarized models. FINN also failed to generate some 7 bit models. This is due to the fact that FINN merges the input and output vectors on the interfaces of various sub-components into a single wire. Vitis HLS has a hard limit on the maximum bitwidth of single wires of 8191. This resulted in an error in the high-level synthesis flow.

The topology of the neural networks is shown in Table 6.2. After every convolution and fully connected layer a batch normalization layer is inserted. The input to the first layer is quantized to 8 bits. All bias values in the CNN are also quantized to 8 bits. The weights and activations, however, are quantized according to the bitwidth specified by the experiment. Each tensor of the models has a single scalar scale factor that is restricted to be a power of two, i.e., per-tensor granularity. In the experiment in which the bitwidth is varied, the bitwidth of 1 results in a binarized model. In this case the ReLU activation is replaced with the sign function. Global unstructured pruning was applied to all models, with the level of pruning being determined by the parameters of the experiment. The pruning procedure is the same as described in previous section.

Table 6.2: Topology of the convolutional neural networks after folding the batch normalization layers.

Layer	Input Size	Kernel Size	Output Size	Activation
Convolution	28 x 28 x 1	3 x 3 x 8	26 x 26 x 8	ReLU
Maximum Pooling	26 x 26 x 8	2 x 2	13 x 13 x 8	-
Depthwise Convolution	13 x 13 x 8	3 x 3	11 x 11 x 8	ReLU
Maximum Pooling	11 x 11 x 8	2 x 2	5 x 5 x 8	-
Fully Connected	200 = 5 x 5 x 8	-	256	ReLU
Fully Connected	256	-	10	-

The training procedure was equivalent to the procedure used for training models for jet tagging. The accuracies obtained by the models on the MNIST test dataset are shown in Figure 6.18. A floating-point model without pruning was also trained using the same procedure and obtained an accuracy of 0.994. As can be seen from Figure 6.18 the neural networks have proven to be very robust to the high levels of quantization and pruning. With a 3 bit model with 70 % of the weights pruned achieving a 0.985 accuracy score. In other words compared to a floating-point model without pruning the storage requirements for the weights of the model are reduced to $\frac{3}{32} \cdot 0.7 \approx 0.066 = 6.6\%$ of the original size of the weights, while keeping practically the same level of accuracy.

The results of the experiments on the CNN models are shown in Figures 6.19 and 6.20.

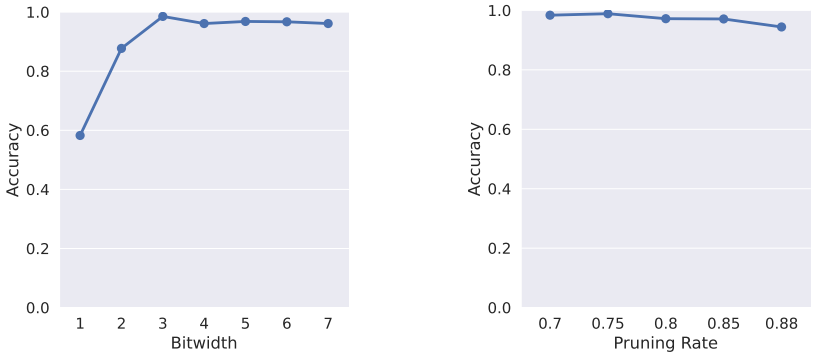


Figure 6.18: Accuracies of the CNN models on the MNIST test dataset.

Number of look-up tables

The circuits generated by hls4ml require the greatest amount of look-up tables, while the circuits generated by FINN require the least amount of look-up tables. The circuits generated by chisel4ml are somewhere in between. However, as mentioned in the individual layer section, FINN is not able to generate circuits that fully parallelize convolution and maximum pooling layer, which is the reason for the low number of look-up tables used. For all tools the increasing rate of pruning results in a decreasing size of the circuits. The increase of the bitwidth of parameters also results in a increasing size of the circuits. The exception to this rule is the binarized neural network implementation by chisel4ml. This peculiarity is likely caused by unlucky weight parameter values. These are a result of the randomness in the training process, and can result in variations between the trained CNN models for each bitwidth.

Total latency

FINNs inability to parallelize convolution and maximum pooling layers results also in a very high total latency. This in turn manifests in a low throughput. Despite the fact that hls4ml generated circuits did not achieve low latency on single convolution layer experiments, they are able to achieve a low initiation interval and low total latency in these CNN experiments. From this it can be concluded that the models and/or configuration in our single convolution layer experiments prevents hls4ml from generating efficient designs. However, those single convolution layer experiments are not representative of performance of convolution layers achieved in larger models like those in this experiments.

Throughput

The throughput achieved by the chisel4ml and hls4ml circuits are high in the range of 60 MHz. However, for both tools this could be tweaked by configuring more aggressive timing targets to the high-level synthesis flow in the case of hls4ml or by increasing the number of pipeline stages in the case of chisel4ml. The circuits generated by FINN, however, are not capable of such high throughput because of the inability in parallelizing convolution and maximum pooling layers.

Generation time

The CNN models in this section are the largest models we tested in our experiments. This has a strong impact on the generation time, and shows a stark difference between high-level synthesis tools like hls4ml and FINN versus the structural description based chisel4ml. It takes over three days to finish the highly parallel designs with hls4ml. Another major disadvantage of the hls4ml tool is its very high consumption of memory, with the peak memory consumption reaching 600 GiB of memory. These factors limit hls4ml for such highly parallel circuits to be used only in HPC platforms and limits the amount of

simultaneous instances the user can run.

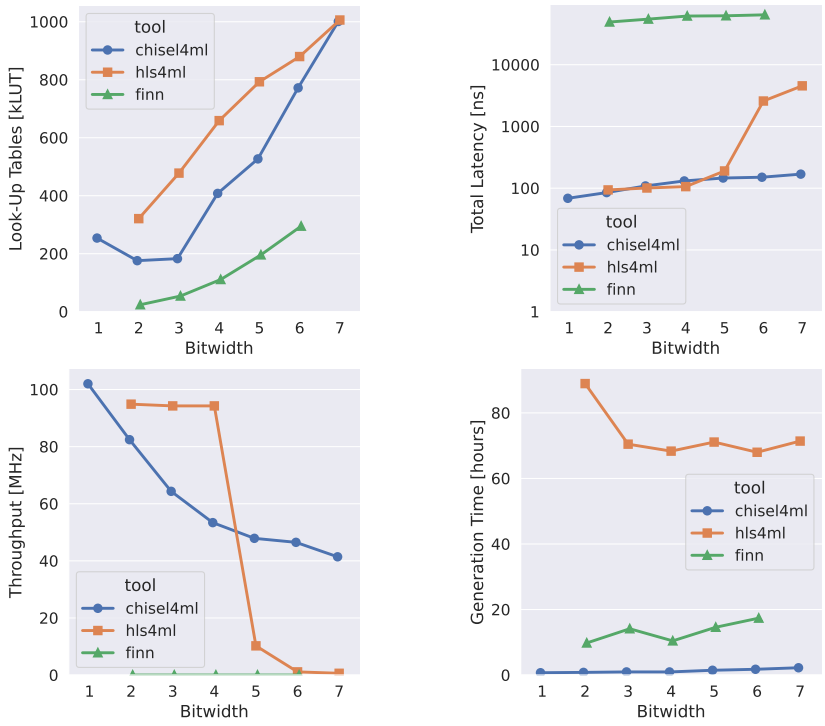


Figure 6.19: The convolutional neural network experiment in which the number of bits used to quantize the weights and activations is varied. The inputs to the CNN are quantized to constant 8 bits. The global prune rate is set to 0.5.

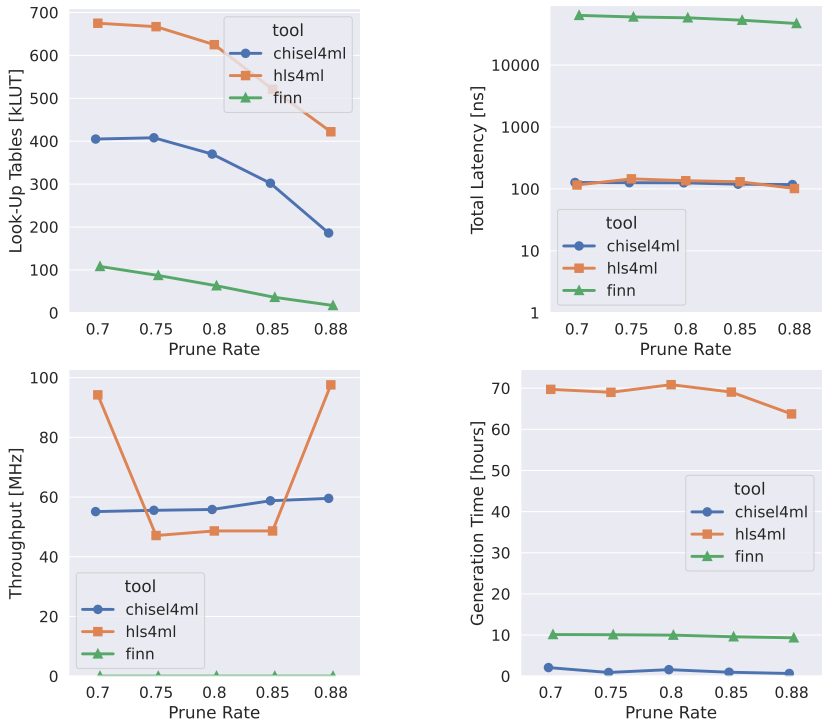


Figure 6.20: The convolutional neural network experiment in which the global prune rate is varied. The weights and activations are quantized to 3 bits. The inputs to the CNN are quantized to constant 8 bits.

Chapter 7

Conclusions

This thesis focuses on exploring generation methods for ultra-low latency hardware implementations of deeply quantized neural networks. The ultra-low latency is achieved by storing the weights entirely on-chip and by heavily parallelizing the neural network processing. Such a technique produces ultra-low latency circuits, but is limited in the size of neural networks it supports. It is crucial that neural networks are highly quantized and pruned for this technique to be viable. Using larger bitwidths would make such a technique impractical, as it would result in circuits that would not fit on most FPGAs. Fortunately, neural networks are very tolerant of quantization in most cases.

We have researched the current state of the art methods, which are based on high-level synthesis technology. These methods have some drawbacks, including the large size of the generated circuits and the very long generation time even for moderately-sized neural networks. To overcome these problems, we have proposed a method based on hardware generators that rely on structural descriptions of the hardware. The result of this research is the chisel4ml tool, which is able to generate direct logic circuit implementations of quantized neural networks in orders of magnitude less time and produce smaller circuits. Using generators based on structural descriptions of the hardware in Chisel, we were able to succinctly describe the fully connected layer, the two-dimensional convolution layer, and the two-dimensional maximum pooling layer. Additionally, the descriptions are general and support a variety of different configurations of the layers, e.g., different kernel sizes, strides, padding, etc. To efficiently support different quantized neural network description formats, we developed an intermediate representation called LBIR. This allowed us to create several frontends that translated into LBIR and create a single backend that uses LBIR.

7.1 Hypothesis

Our initial hypothesis for this work were:

H1: Deeply quantized neural networks can be efficiently and automatically converted to highly parallel hardware implementations using generators based on structural descriptions of the hardware.

H2: The automatic conversion of deeply quantized neural networks using generators based on structural descriptions of the hardware can perform the conversion in significantly less time, then conversions based on algorithmic descriptions.

H3: The automatic conversion of deeply quantized neural networks using generators based on structural descriptions of the hardware can generate pipelined hardware circuits that can achieve very high throughput and ultra-low latency, when implemented on field-programmable gate array circuits.

7.2 Synopsis

The programming features of the Scala language and thus also of the Chisel language are crucial for the efficient expression of the hardware we develop. The core of this abstraction are the `NeuronCompute` and `OrderCompute` abstract classes, which defines abstract types and abstract functions that operate on them. The first of these provides the functions required to compute the neuron operation and the second to compute the maximum pooling operation. By abstracting the details of quantization, we are able to express the implementations in a generic way. In this way, the same logic can be reused to implement neurons and maximum pooling operators quantized to different specifications. Also, we can very easily extend chisel4ml to enable new types of quantization, as this can be achieved simply by constructing new subclasses of `NeuronCompute` and `OrderCompute`.

The difference between convolution and fully connected layers is merely in the way the input is connected to the neurons of the layer. This observation allowed us to generalize the process of instantiating hardware for these layers. When instantiating a neuron, a suitable subset of the input, weights, bias, and scale factor is selected. The selection of the subset is based on simple mapping functions that take as input the specification of the layer and return a corresponding sequence or a value thereof, e.g., they return the sequence of inputs that are input to the first neuron of a convolution layer. By abstracting the connections of the neurons, we can also more easily extend support for other types of neural network layers based on neurons. For example, with this technique we could easily enable support for one-dimensional convolution layers in the chisel4ml backend, as we would only need to provide appropriate mapping functions.

The abstractions in our hardware generators have enabled us to express our hardware efficiently and concisely. From this we conclude that the hypothesis **H1** is confirmed.

To evaluate our methodology, we conducted a series of experiments with a wide range of different quantized neural network models. The first set of experiments involved single layer models, where we attempted to isolate the performance of the tools on different attributes of the fully connected, convolution and maximum pooling layers. This was followed up by training and testing a series of multilayer perceptron models and then a series of convolutional neural network models. The goal of these experiments was to determine how well the tools can handle larger neural network models.

The experiments with fully connected single layer models showed that all three tools can generate direct logic circuits that have high throughput and low latency for these types of models. The size of the circuits generated by chisel4ml and hls4ml is similar, but the circuits generated by FINN consume more hardware resources, especially at higher bitwidths. The maximum path delay and in turn total latency of the chisel4ml circuits was uneven throughout the experiments because chisel4ml does not automatically adjust the pipeline depth to achieve a desired maximum path delay. However, the user can easily adjust the maximum path delay by changing the number of pipeline stages. These trends in the experiments with fully connected single layer models were reaffirmed and reinforced by the experiments with MLPs consisting only of fully connected layers.

For the single layer models with convolution and maximum pooling, the difference between the performance of the tools is much larger. The FINN tool is not able to generate direct logic circuits and therefore can not fulfill the requirements for ultra-low latency. Both chisel4ml and hls4ml can generate highly parallel implementations of the convolution layer and the maximum pooling layer, but the circuits generated by chisel4ml use fewer look-up tables than those generated by hls4ml in most of these experiments, especially for the maximum pooling layer.

Based on the results discussed in the previous two paragraphs, we have concluded that our method is capable of achieving the same or better performance than the state of the

art on a variety of different neural network topologies. Therefore, we conclude that the hypothesis **H3** is confirmed.

In all experiments, the chisel4ml tool was able to generate circuits in significantly less time than hls4ml or FINN. This difference becomes increasingly clear as the size of the neural network and the circuits increase. For neural network models that work with high-dimensional data with many elements, such as convolution layers, the differences become ever greater. In the CNN experiments, which are the largest neural network models we tested, the difference in generation speed is more than 100x. This confirms the hypothesis **H2**.

7.3 Scientific Contributions

The main scientific contributions of this thesis are:

SC1: We have developed a method for implementing highly parallel hardware implementations of deeply quantized neural networks based on hardware generators that use structural descriptions.

SC2: We have developed an intermediate representation capable of representing a wide range of uniformly quantized neural networks. The representation supports different granularities of quantization and is extensible.

SC3: We have developed an open source tool that is a practical implementation of our methods and is open for others to use.

SC4: We conducted an in-depth analysis of our chisel4ml tool, hls4ml and FINN for the task of generating highly parallel implementations of deeply quantized neural networks. The analysis investigated both the per-layer properties and how the tools can handle larger neural networks.

7.4 Future Work

The biggest disadvantage of our approach is that it is limited to the pre-designed hardware structures. Both hls4ml and FINN have parameters that can be used to control the degree of parallelism of a circuit, so that the user is able to choose a compromise between the complexity of the circuit and the desired performance. Such a facility in chisel4ml would allow a wider range of applications with different performance requirements to be addressed. To make this possible in the future, we intend to write a series of parameterizable generators for sequential circuits. By providing an intelligent method for selecting processing units, the user would be able to select the desired metrics and ask this procedure to select the appropriate processing units to achieve them. Such a method could be formulated as a constrained optimization problem.

Another avenue for future work would be to investigate the effects of approximate multipliers on the performance of the neural networks and the size of the generated circuits. In general, approximate multipliers below 8 bits do not provide a significant advantage over exact multipliers, so they are only useful in the layers which are not heavily quantized. In deeply quantized neural networks, this is usually the input to the first layer. The input layer often can not be quantized, as this would mean throwing away valuable information. Such layers consume a disproportionate amount of logic resources in the final implementations. In our opinion, it might be possible to significantly reduce the amount of logic resources used in the input layer by using approximate multipliers in the first layer.

Appendix A

Plots of All Experiment Results

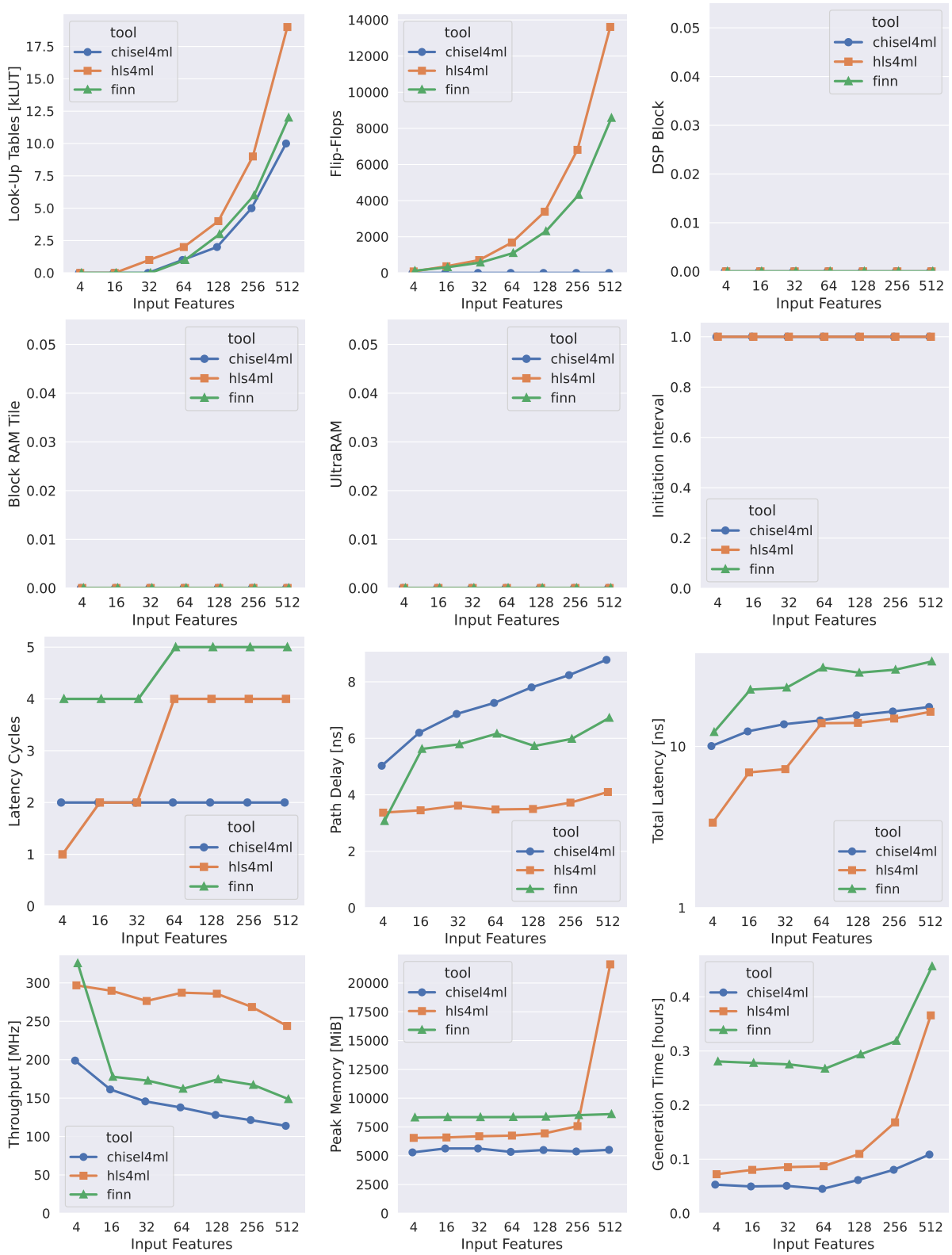
In this appendix we provide the entire set of results in form of graphical plots. For each experiment we provide a wide set of results. Table A.1 provides a description of the measured quantities.

Table A.1: Quantity description table.

Quantity	Description
Generation Time	The generation time in hours.
Look-Up Tables	The number of look-up tables used by the design.
Flip-Flops	The number of flip-flops used by the design.
Block RAM	The number of Block RAM tiles used by the design. This includes both 36 and 16 kbit Block RAM blocks.
UltraRAM	The number of UltraRAM blocks used by the design.
DSP Block	The number of DSP blocks used by the design.
Path Delay	The maximum path delay of the design in nanoseconds.
Peak Memory	The peak memory consumption of the generation process in MiB.
Initiation Interval	Initiation interval of the design. That is the number of cycles that it takes the design to except a new input.
Throughput	Throughput of the design in number of samples per second.
Latency Cycles	The latency of the design in cycles. That is how many cycles it takes to compute the output for one particular input.
Total Lateny	The total latency of the design in nanoseconds. That is how many nanoseconds it takes to compute the output for one particular input.

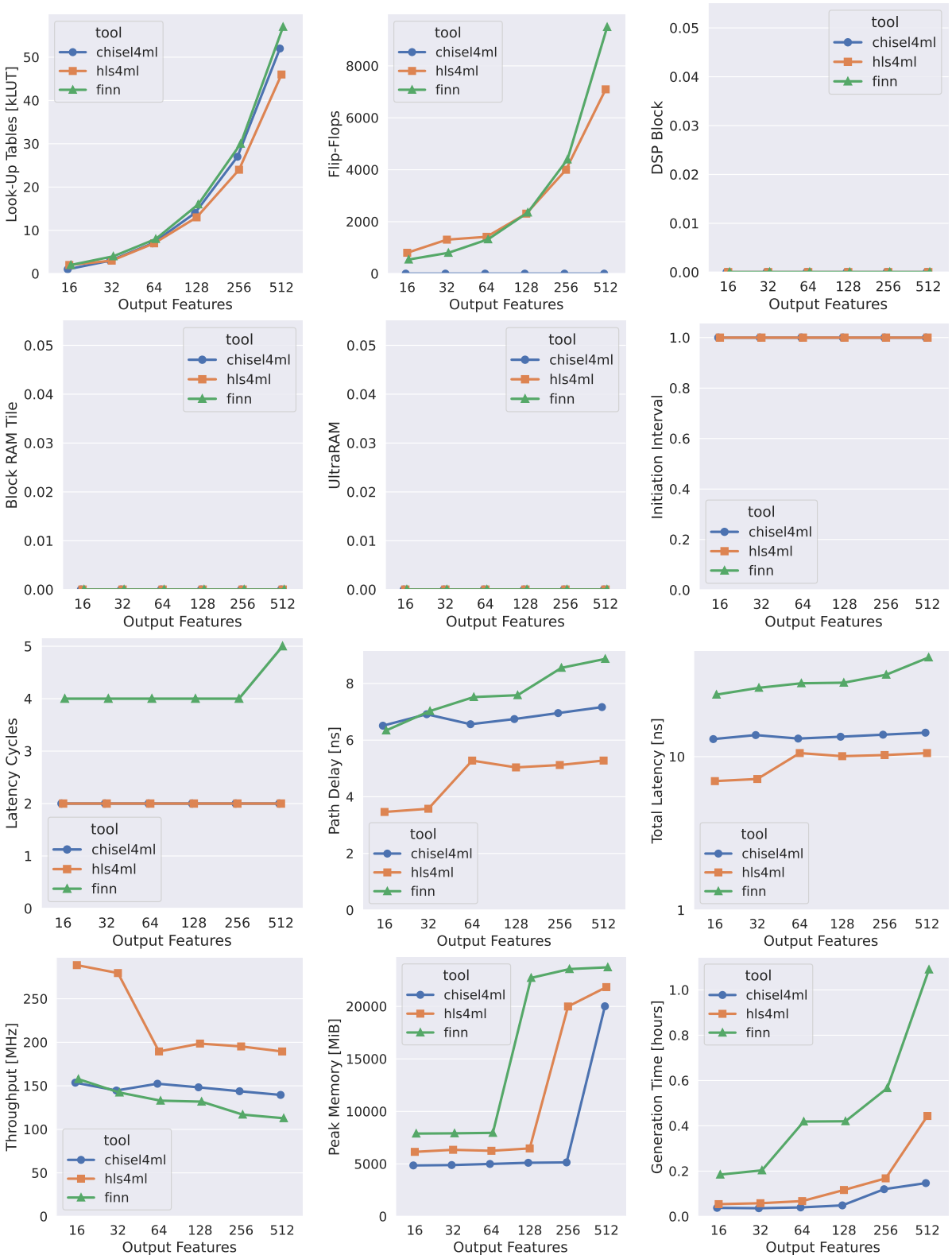
A.1 Fully Connected Layer - Input Features

Single fully connected layer experiment in which the number of input features is varied. The number of output features equals 32. The input, output and weights are quantized to 4 bits and the bias is quantized to 8 bits.



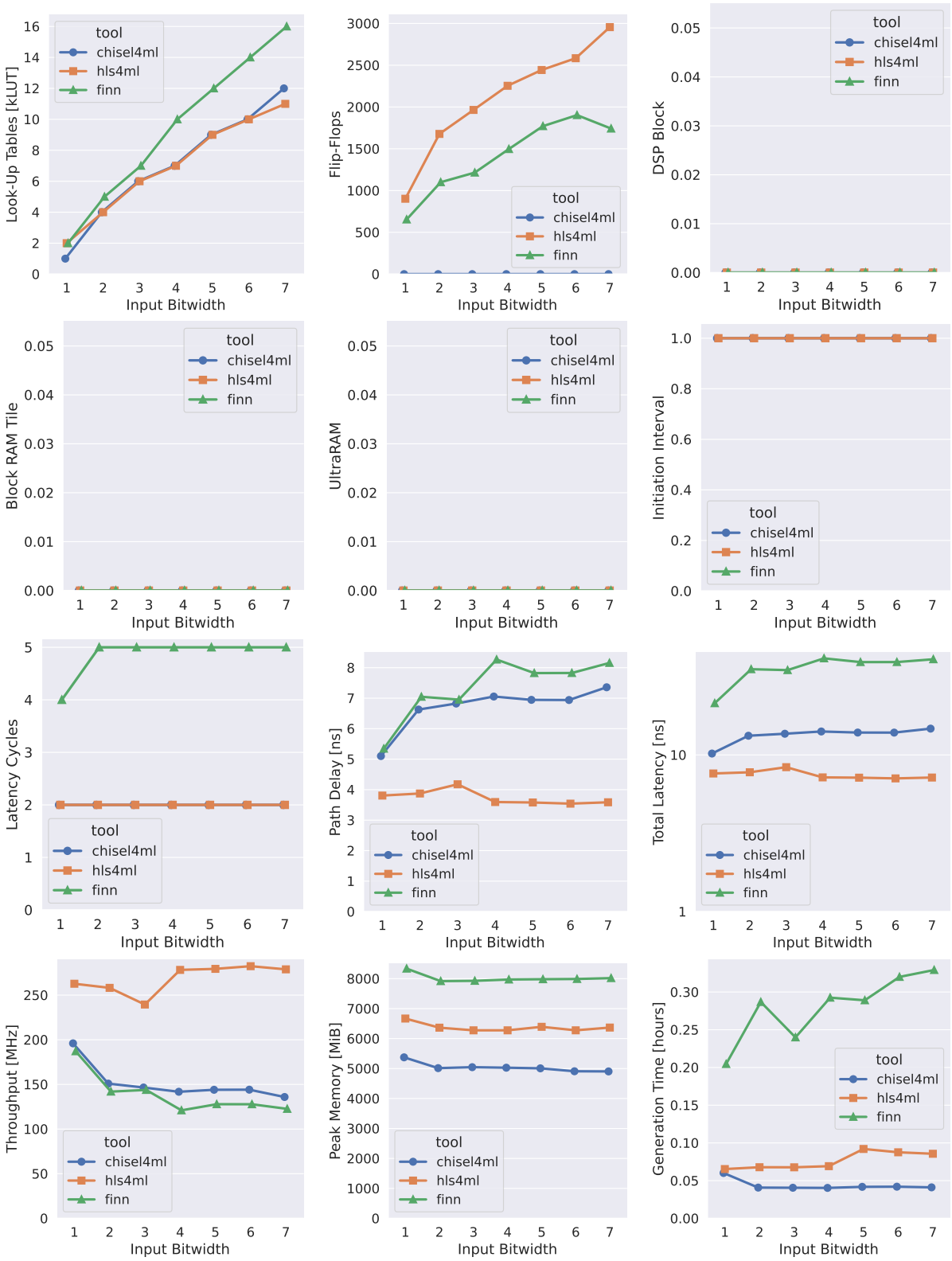
A.2 Fully Connected Layer - Output Features

Single fully connected layer experiment in which the number of output features is varied. The number of input features equals 16. The input, output and weights are quantized to 4 bits and the bias is quantized to 8 bits.



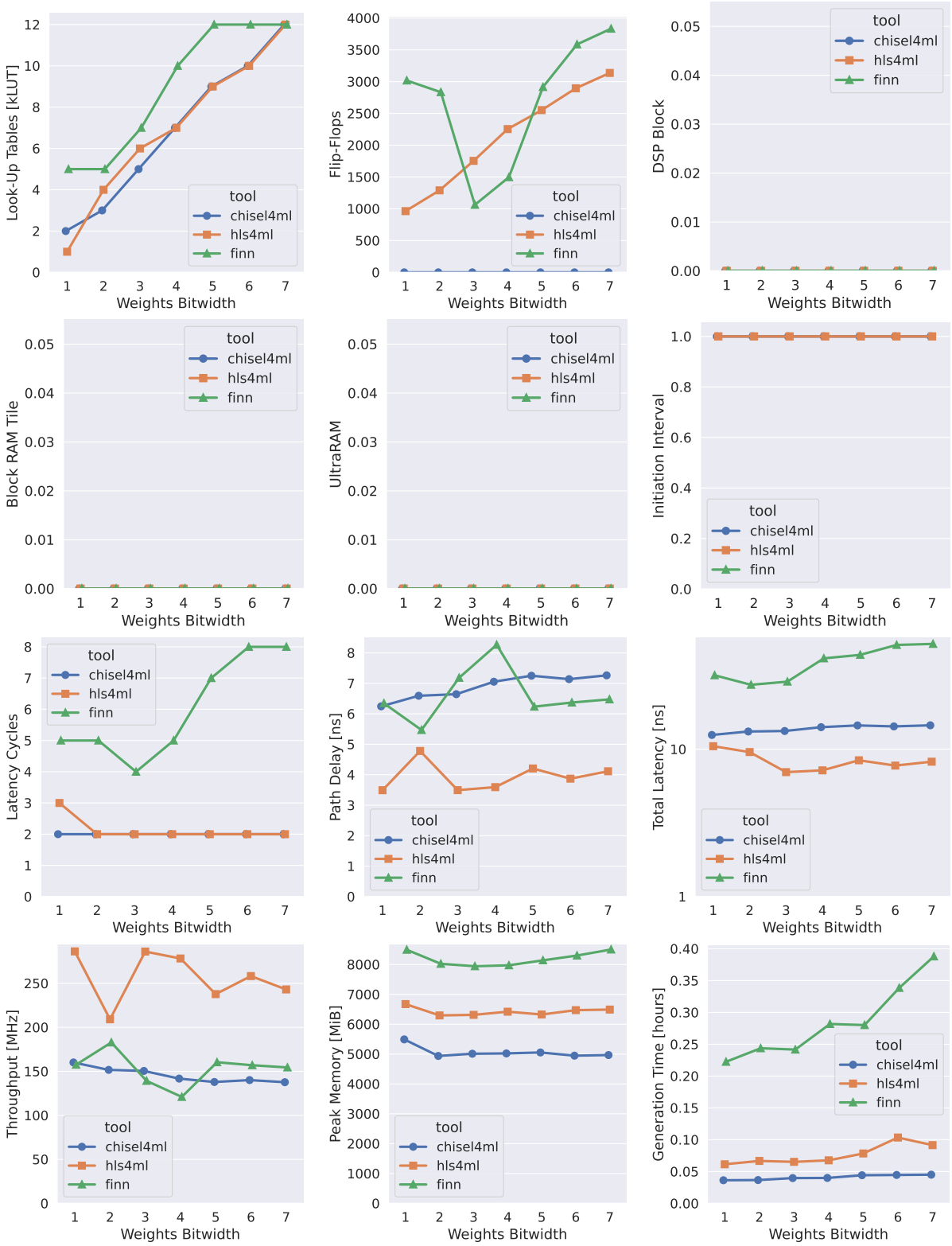
A.3 Fully Connected Layer - Input Bitwidth

Single fully connected layer experiment in which the number of bits used to quantize the input features is varied. The number of input and output features equals 32. The weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits.



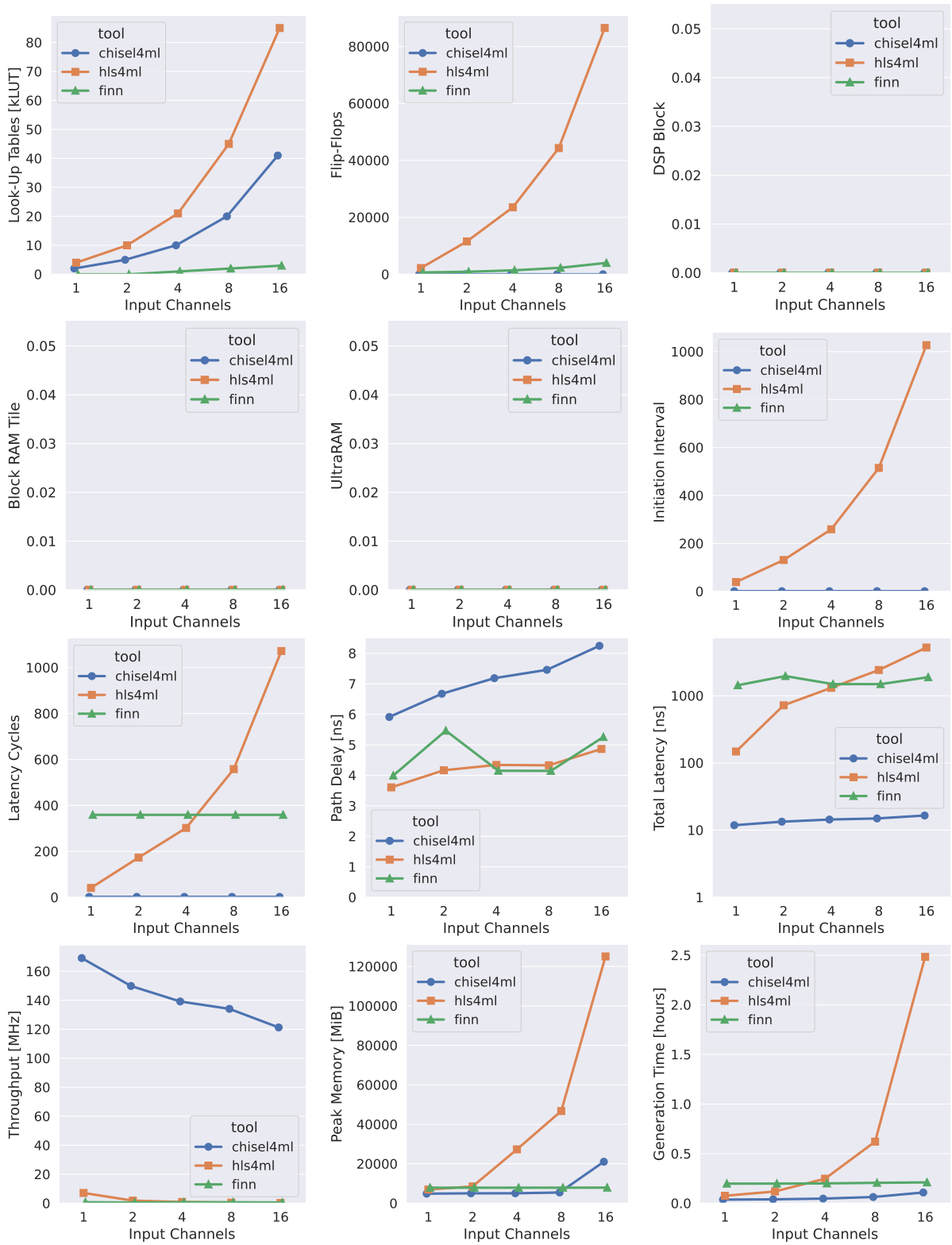
A.4 Fully Connected Layer - Weight Bitwidth

Single fully connected layer experiment in which the number of bits used to quantize the weight parameters is varied. The number of input and output features equals 32. The inputs and outputs are quantized to 4 bits and the bias is quantized to 8 bits.



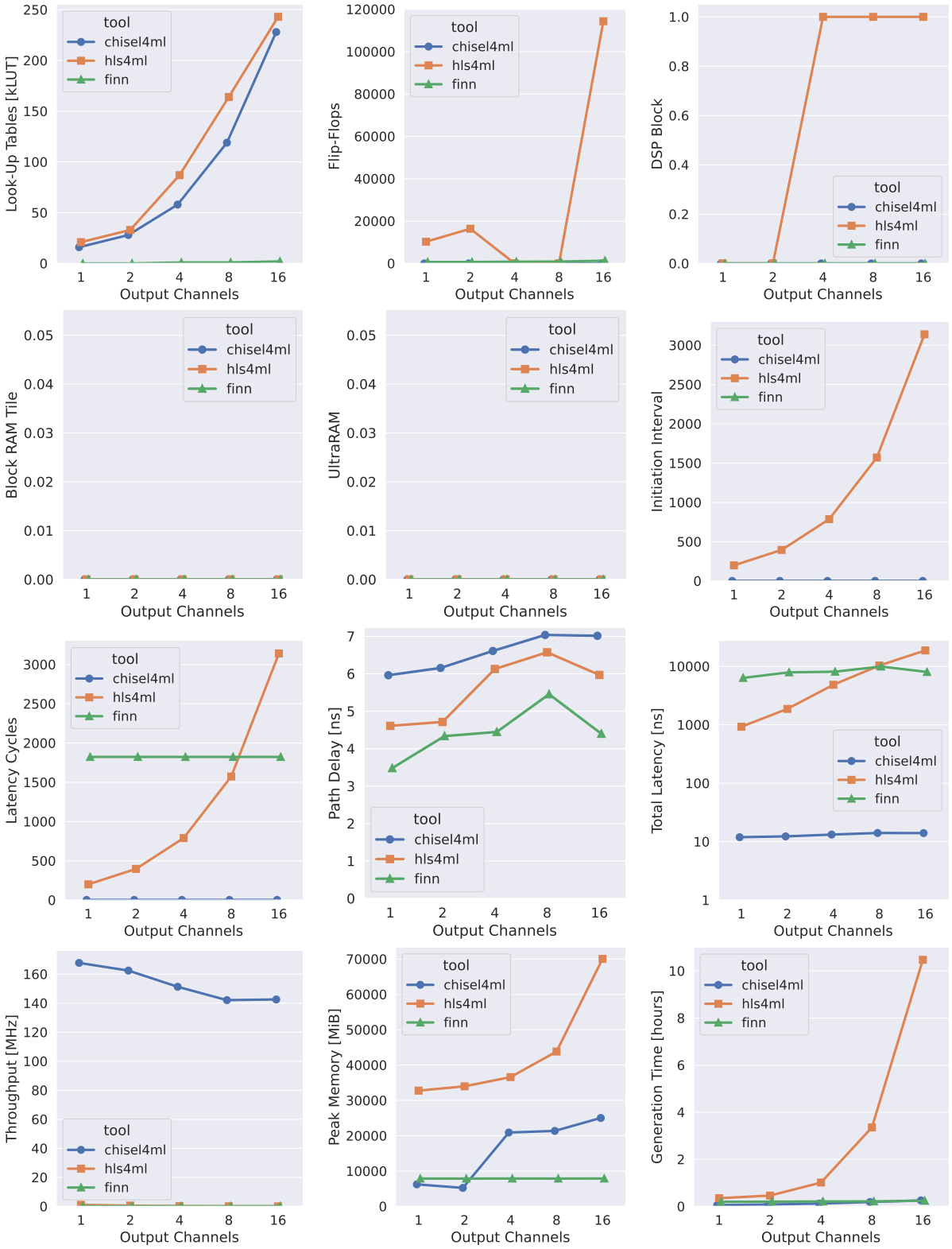
A.5 Convolution Layer - Input Channels

Single convolution layer experiment in which the number of input channels of the convolution layer is varied. The input size is set to an 8 by 8 square and the kernel size is set to 3 by 3. The number of output channels is set to 1. The inputs, weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits.



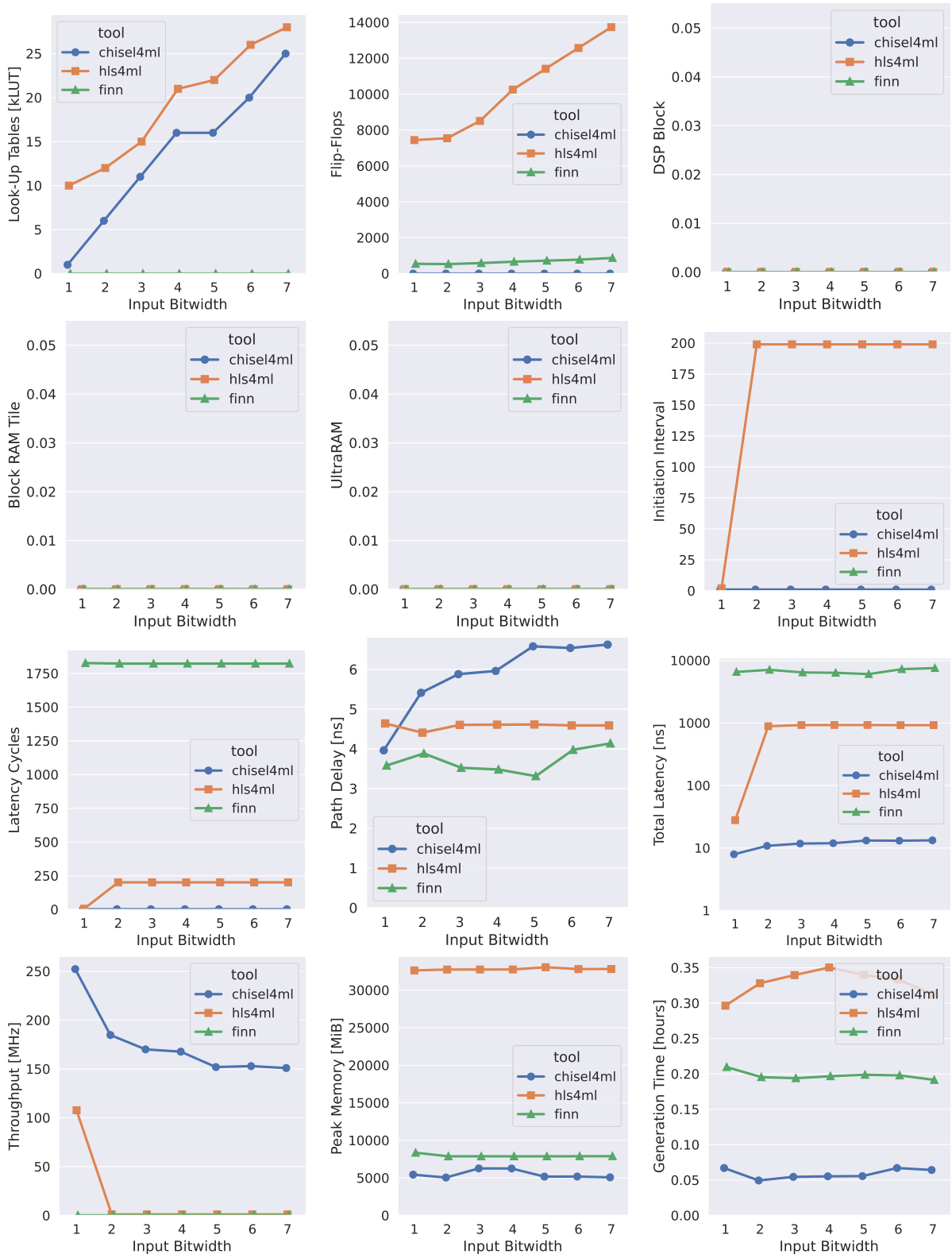
A.6 Convolution Layer - Output Channels

Single convolution layer experiment in which the number of output channels of the convolution layer is varied. The input size is set to an 16 by 16 square and the kernel size is set to 3 by 3. The number of input channels is set to 1. The inputs, weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits.



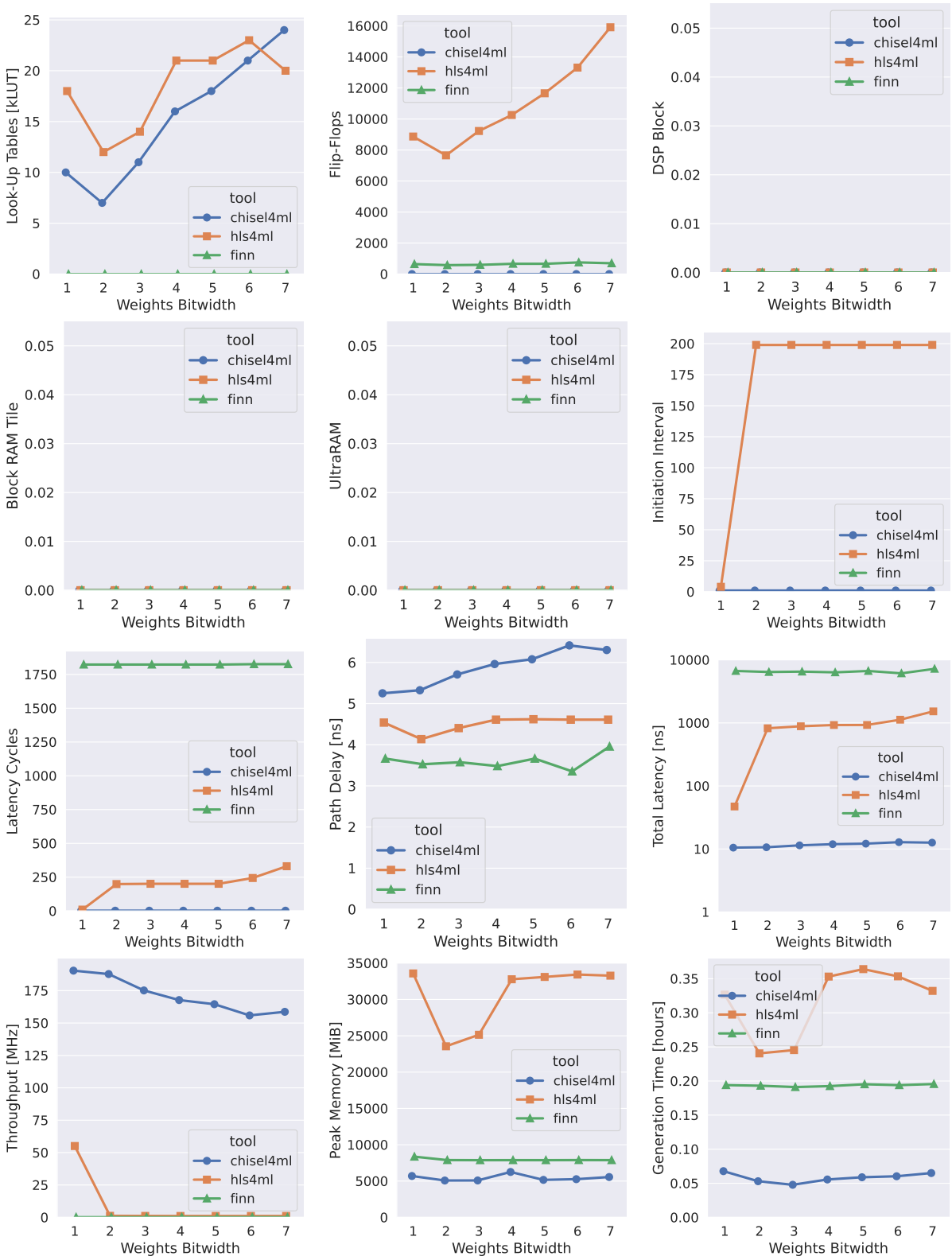
A.7 Convolution Layer - Input Bitwidth

Single convolution layer experiment in which the number of bits used to quantize the input is varied. The input size is set to an 16 by 16 square and the kernel size is set to 3 by 3. The number of input and output channels is set to 1. The weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits.



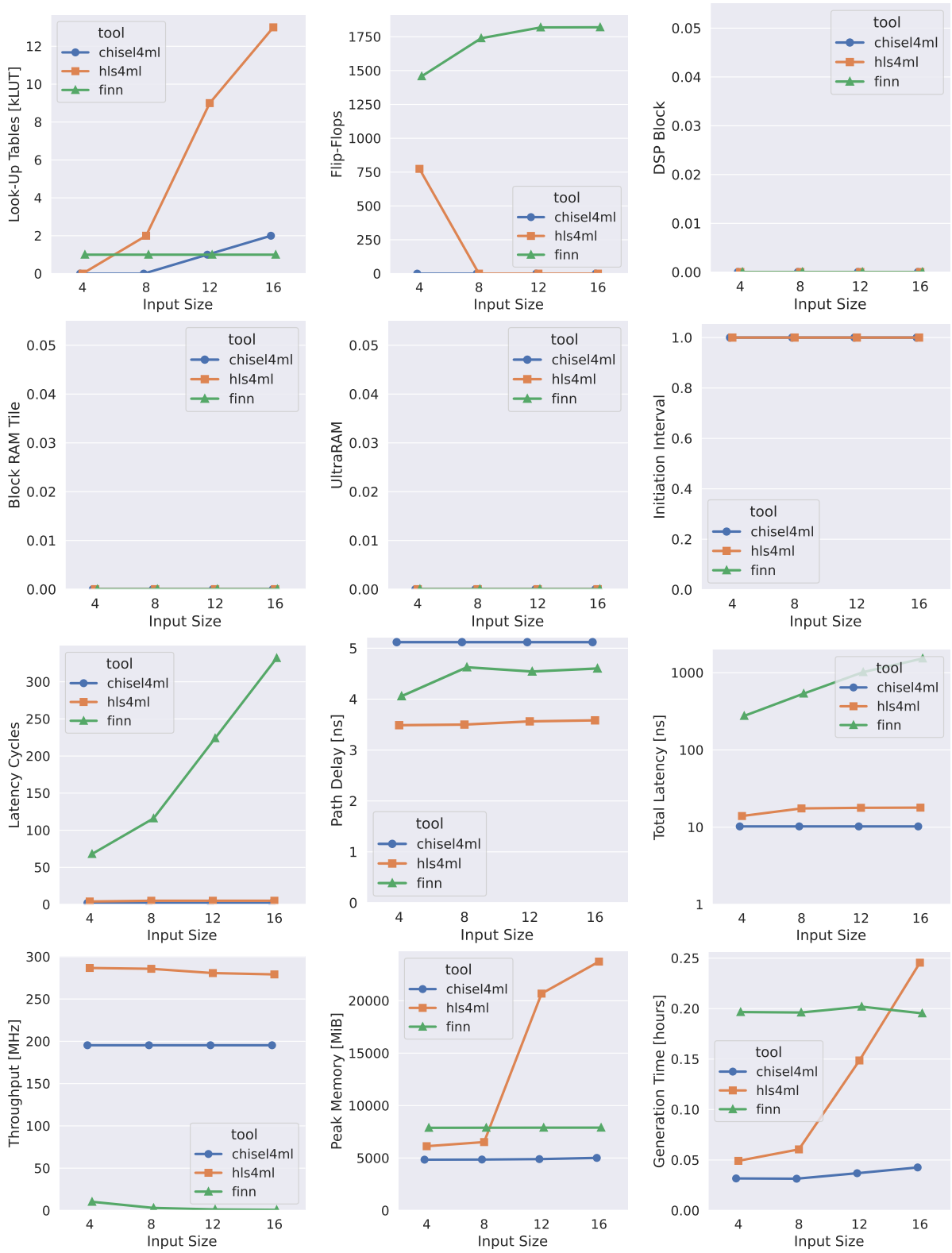
A.8 Convolution Layer - Weight Bitwidth

Single convolution layer experiment in which the number of bits used to quantize the weight parameters is varied. The input size is set to an 16 by 16 square and the kernel size is set to 3 by 3. The number of input and output channels is set to 1. The inputs and outputs are quantized to 4 bits, and the bias is quantized to 8 bits.



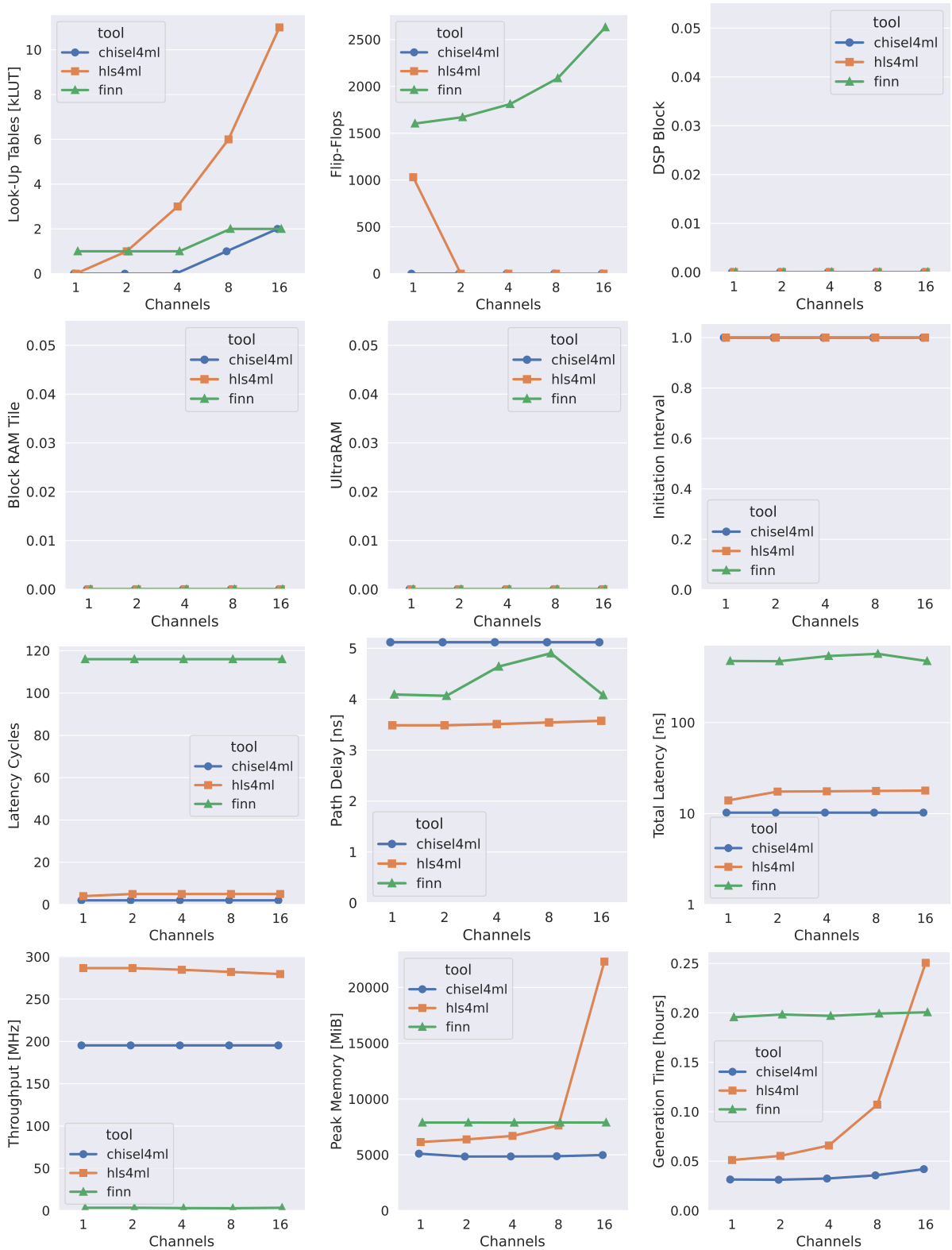
A.9 Maximum Pooling Layer - Input Size

Single maximum pooling layer experiment in which the input size is varied whilst keeping it square. The size of the kernel is 3 by 3 and the number of channels is set to 3. The inputs to the layer are quantized to 4 bits.



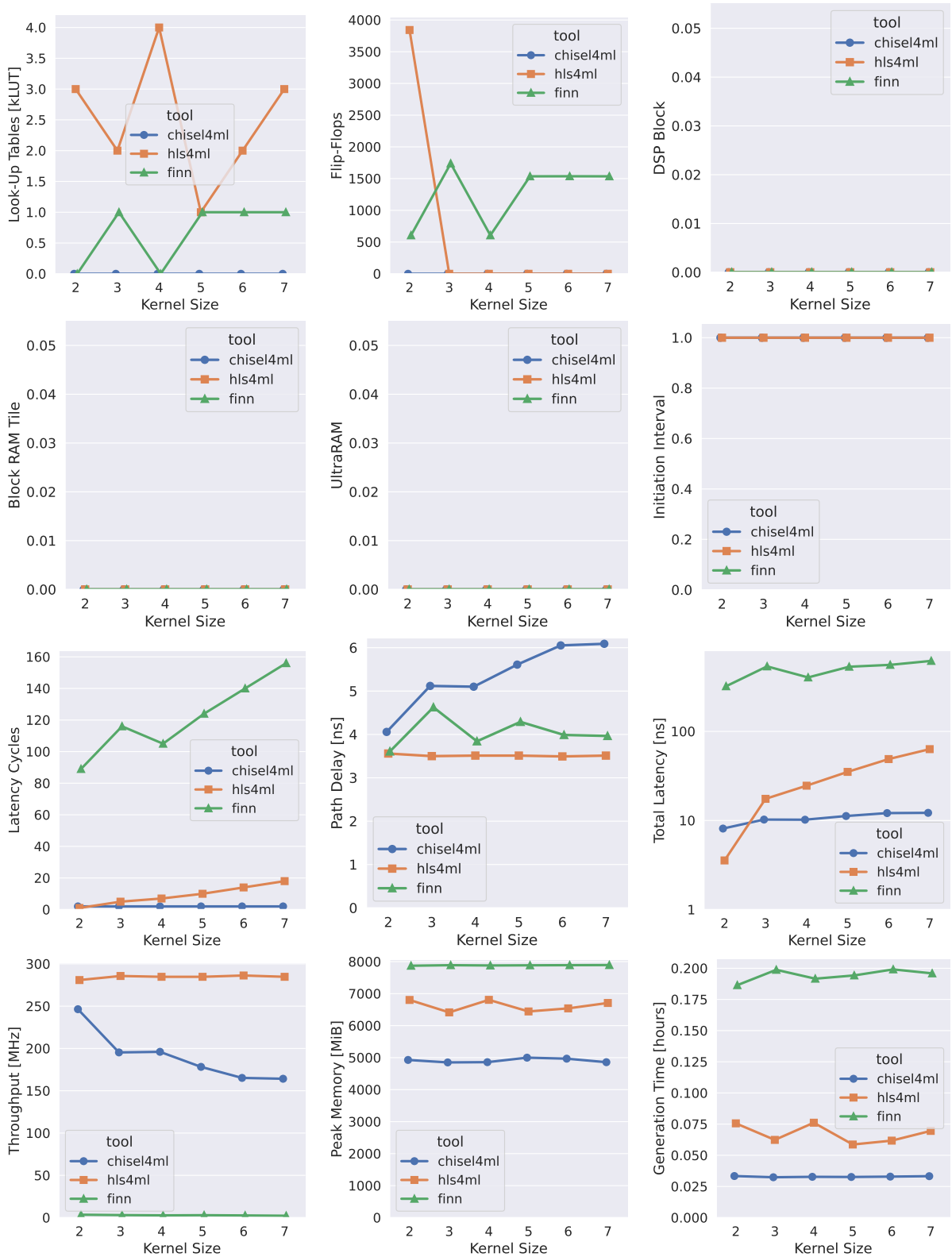
A.10 Maximum Pooling Layer - Channels

Single maximum pooling layer experiment in which the number of channels of the input tensor is varied. The input size is set to an 8 by 8 square and the kernel size is set to 3 by 3. The inputs to the layer are quantized to 4 bits.



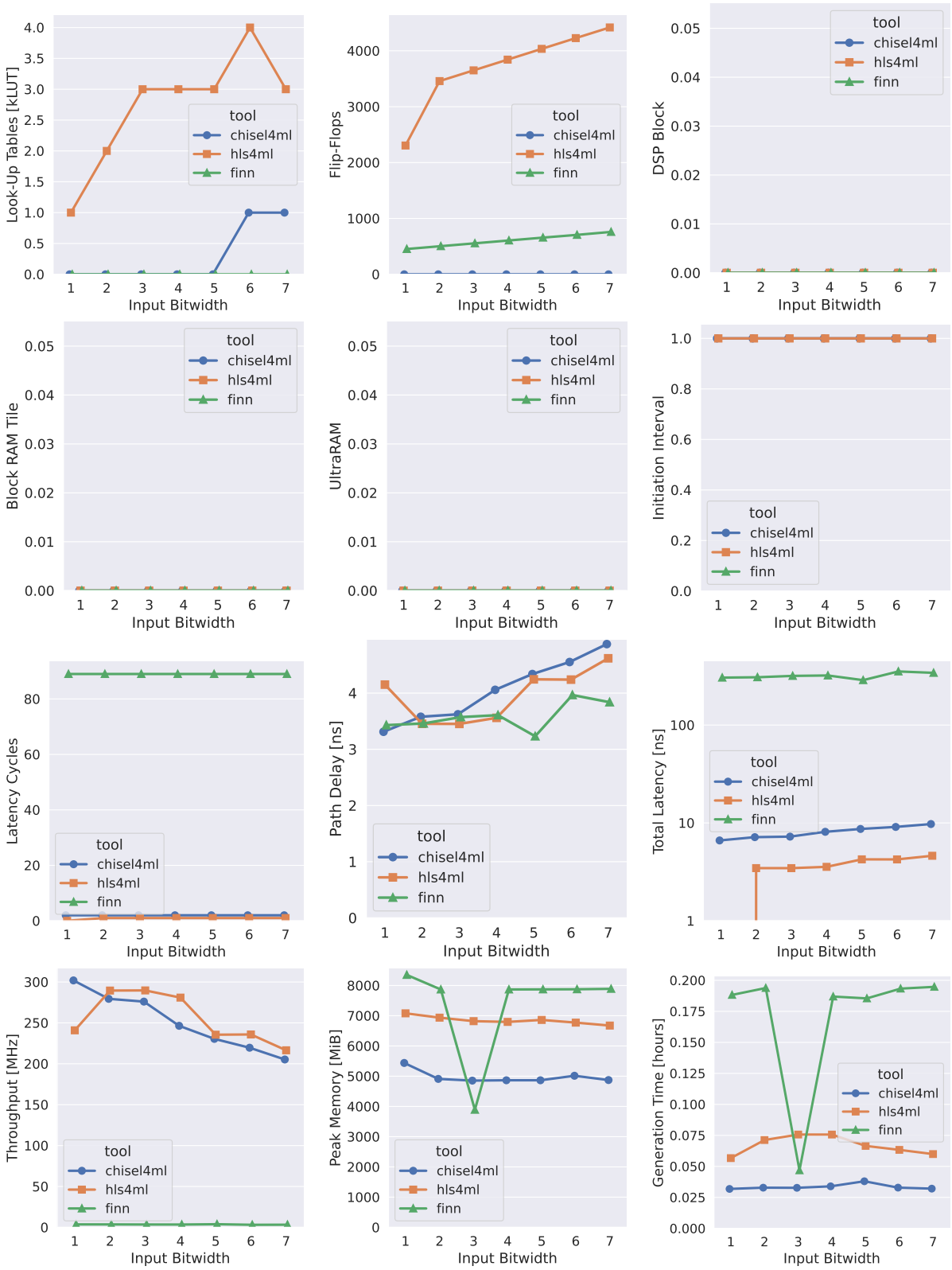
A.11 Maximum Pooling Layer - Kernel Size

Single maximum pooling layer experiment in which the size of the square kernel is varied. The number of channels is set to 3 and input size to an 8 by 8 square. The inputs to the layer are quantized to 4 bits.



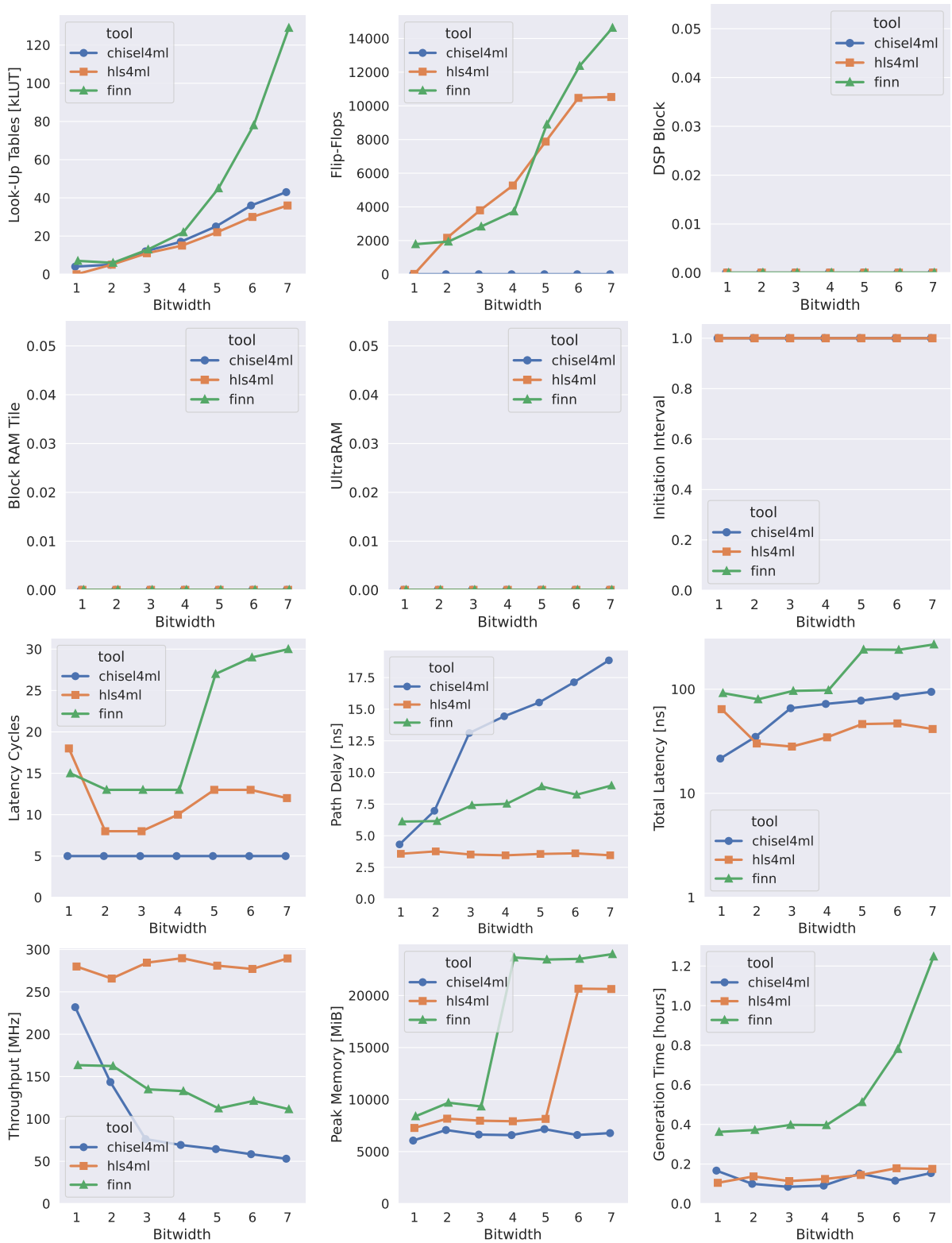
A.12 Maximum Pooling Layer - Input Bitwidth

Single maximum pooling layer experiment in which the number of bits used to quantize the input is varied. The number of channels is set to 3 and the input size to an 8 by 8 square. The kernel size is set to 2 by 2.



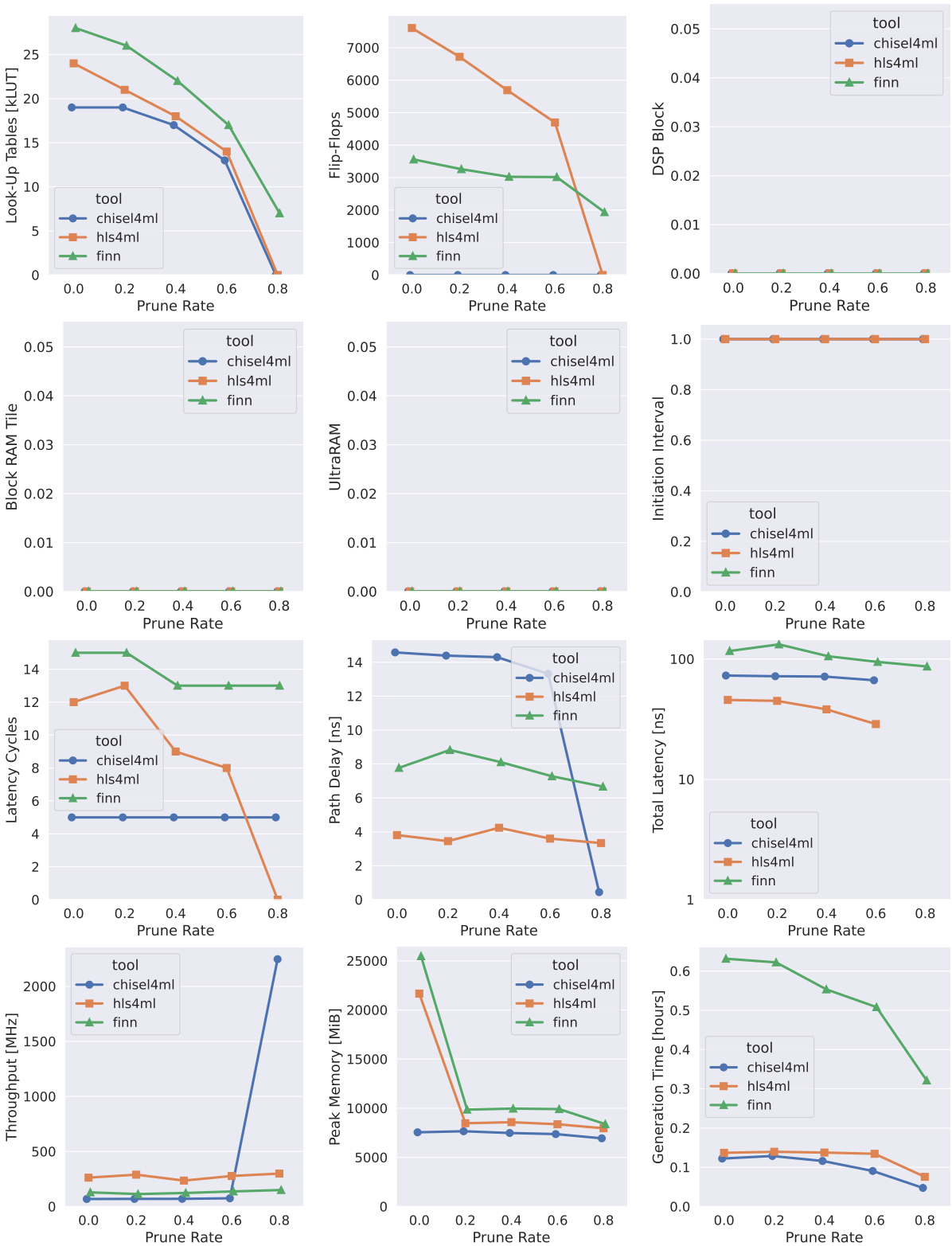
A.13 Jet Tagging MLP - Bitwidth

The multilayer perceptron experiment in which the number of bits used to quantize the weights and activations is varied. The inputs to the MLP are quantized to 8 bits. The global prune rate is set to 0.5.



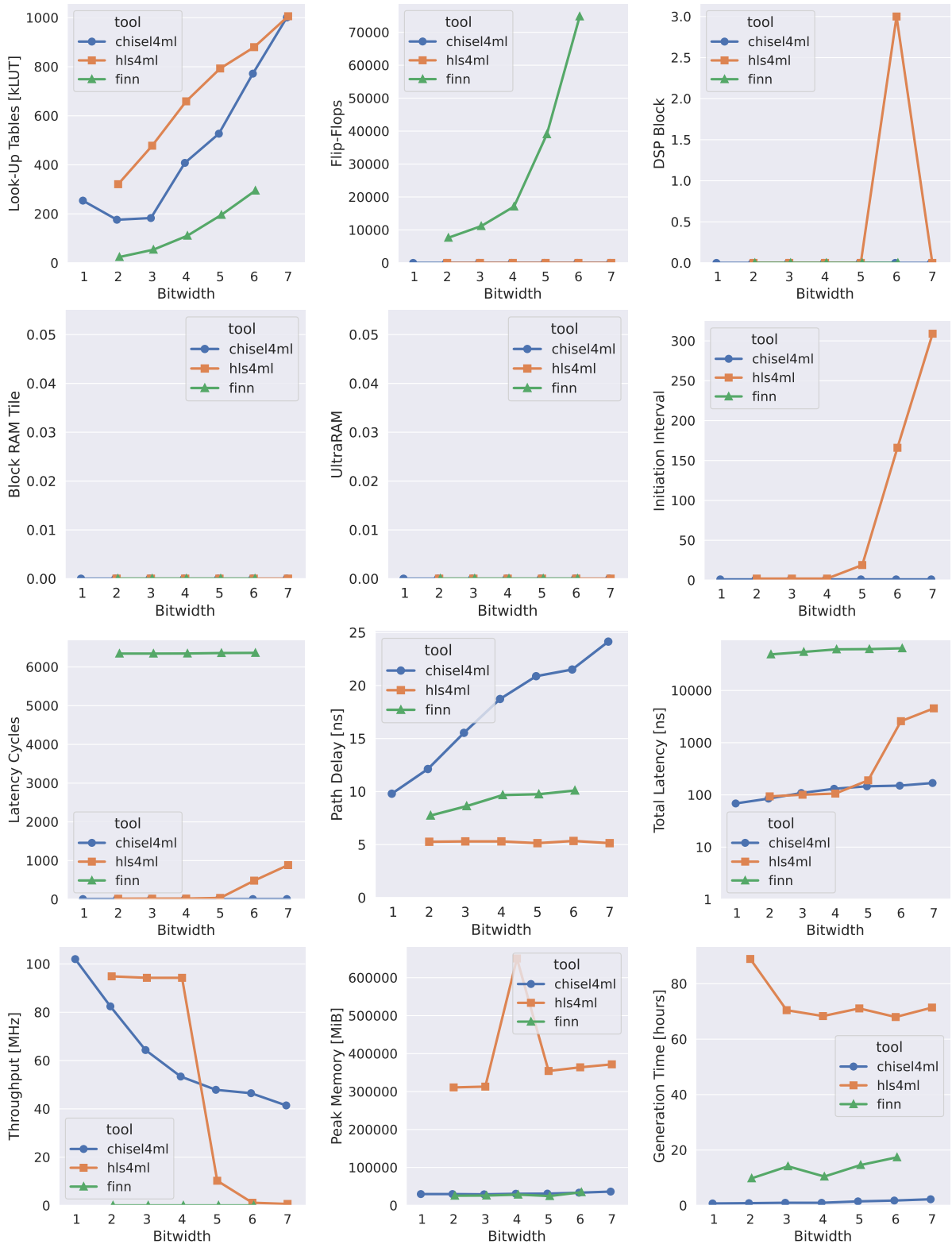
A.14 Jet Tagging MLP - Prune Rate

The multilayer perceptron experiment in which the global prune rate is varied. The weights and activations are quantized to 4 bits. The inputs to the MLP are quantized to constant 8 bits.



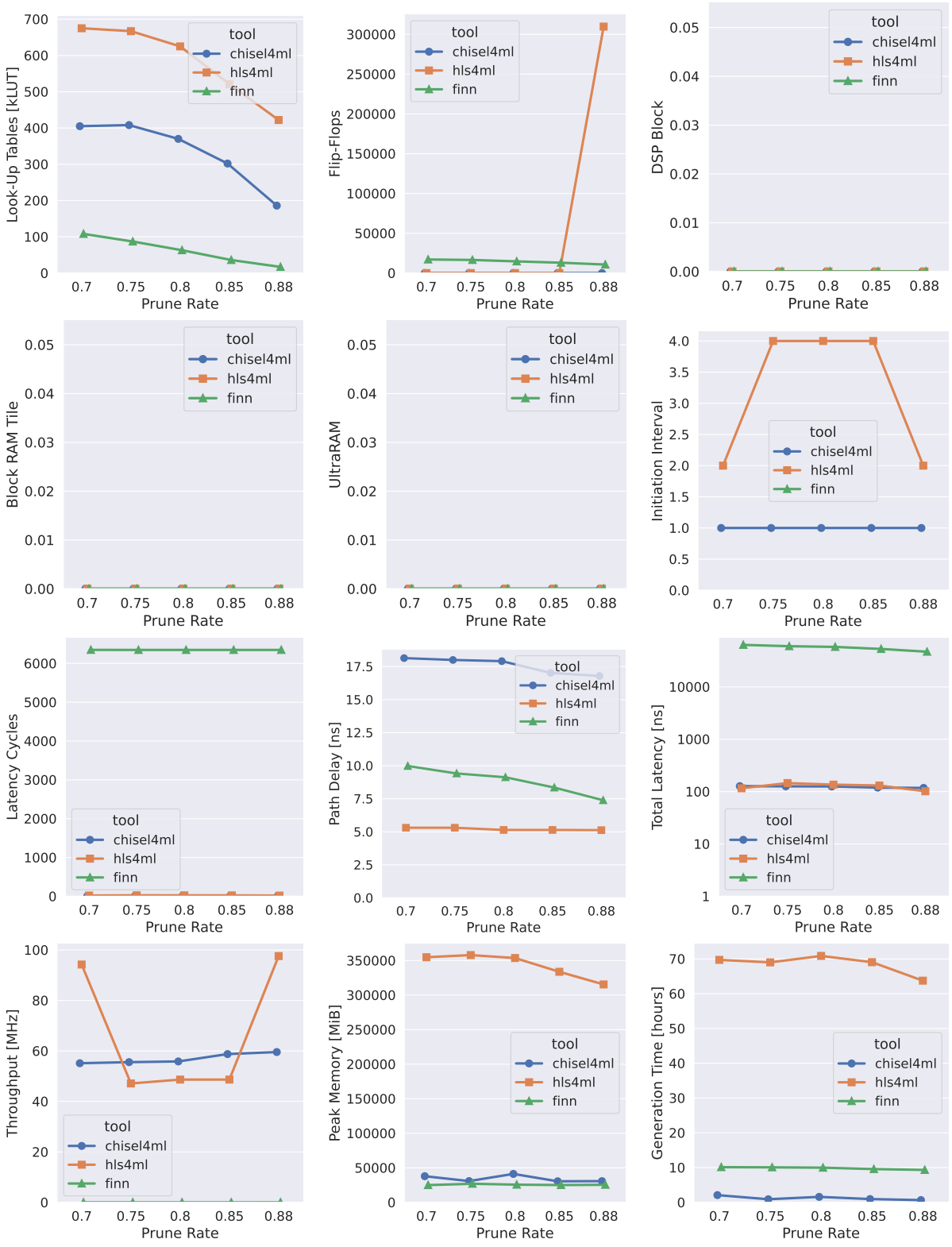
A.15 MNIST CNN - Bitwidth

The convolutional neural network experiment in which the number of bits used to quantize the weights and activations is varied. The inputs to the CNN are quantized to constant 8 bits. The global prune rate is set to 0.5.



A.16 MNIST CNN - Prune Rate

The convolutional neural network experiment in which the global prune rate is varied. The weights and activations are quantized to 3 bits. The inputs to the CNN are quantized to constant 8 bits.



Appendix B

Table of All Experiment Results

In this appendix we provide the entire set of results in a form of a tables. For each experiment we provide a separate table and each table contains a short description of the experiment in the caption. Table B.1 details the shorthands of the columns we provide.

Table B.1: Description of shorthands.

Quantity Shorthand	Description
Time	The generation time in hours.
LUT	The number of look-up tables used by the design.
FF	The number of flip-flops used by the design.
BRAM	The number of Block RAM tiles used by the design. This includes both 36 and 16 kbit Block RAM blocks.
BRAM36	The number of 36 kbit Block RAM blocks used by the design.
BRAM16	The number of 16 kbit Block RAM blocks used by the design.
URAM	The number of UltraRAM blocks used by the design.
DSP	The number of DSP blocks used by the design.
PD	The maximum path delay of the design in nanoseconds.
PM	The peak memory consumption of the generation process in MiB.
II	Initiation interval of the design. That is the number of cycles that it takes the design to except a new input.
TP	Throughput of the design in number of samples per second.
LC	The latency of the design in cycles. That is how many cycles it takes to compute the output for one particular input.
TL	The total latency of the design in nanoseconds. That is how many nanoseconds it takes to compute the output for one particular input.

B.1 Fully Connected Layer - Input Features

Table B.2: Single fully connected layer experiment in which the number of input features is varied. The number of output features equals 32. The input, output and weights are quantized to 4 bits and the bias is quantized to 8 bits.

Input Features	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
16	chisel4ml	0.04	3908	0	0	0	0	0	0	6.91	5018.02	1	144654997	2	13.83
32	chisel4ml	0.04	7587	0	0	0	0	0	0	7.05	4986.06	1	141763538	2	14.11
64	chisel4ml	0.05	15052	0	0	0	0	0	0	7.63	5184.85	1	131061598	2	15.26
128	chisel4ml	0.16	29989	0	0	0	0	0	0	8.08	5785.41	1	123808344	2	16.15
256	chisel4ml	0.23	61228	0	0	0	0	0	0	8.50	22073.74	1	117702448	2	16.99
512	chisel4ml	0.22	119541	0	0	0	0	0	0	9.07	21604.55	1	110253583	2	18.14
16	hls4ml	0.06	3954	1308	0	0	0	0	0	3.58	6181.11	1	279563880	2	7.15
32	hls4ml	0.07	7780	2254	0	0	0	0	0	3.59	6278.07	1	278241513	2	7.19
64	hls4ml	0.13	15751	4016	0	0	0	0	0	4.31	6620.29	1	231803430	2	8.63
128	hls4ml	0.17	31471	8667	0	0	0	0	0	3.53	19757.96	1	283607487	4	14.10
256	hls4ml	0.36	62910	16691	0	0	0	0	0	3.54	21118.25	1	282326369	4	14.17
512	hls4ml	1.06	126033	33056	0	0	0	0	0	3.69	25137.32	1	271002710	4	14.76
16	finn	0.21	4949	801	0	0	0	0	0	2.97	7906.79	/	142328494	4	11.90
32	finn	0.29	10899	1497	0	0	0	0	0	1.73	7965.45	/	120962864	5	8.67
64	finn	0.36	20345	2008	0	0	0	0	0	1.84	22948.95	/	122609122	5	9.22
128	finn	0.55	36460	3111	0	0	0	0	0	2.06	23985.16	/	125912868	5	10.29
256	finn	0.89	68786	5541	0	0	0	0	0	1.49	24617.09	/	117439812	5	7.43
512	finn	2.07	130823	10359	0	0	0	0	0	0.23	25731.77	/	102396067	5	1.17

B.2 Fully Connected Layer - Output Features

Table B.3: Single fully connected layer experiment in which the number of output features is varied. The number of input features equals 16. The input, output and weights are quantized to 4 bits and the bias is quantized to 8 bits.

Output Features	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
16	chisel4ml	0.04	1998	0	0	0	0	0	0	6.51	4855.37	1	153633430	2	13.02
32	chisel4ml	0.04	3908	0	0	0	0	0	0	6.91	4890.91	1	144654997	2	13.83
64	chisel4ml	0.04	7392	0	0	0	0	0	0	6.56	4995.01	1	152415790	2	13.12
128	chisel4ml	0.05	14096	0	0	0	0	0	0	6.74	5107.24	1	148301942	2	13.49
256	chisel4ml	0.12	27492	0	0	0	0	0	0	6.95	5143.35	1	143802128	2	13.91
512	chisel4ml	0.15	52616	0	0	0	0	0	0	7.17	20013.16	1	139567341	2	14.33
16	hls4ml	0.05	2035	803	0	0	0	0	0	3.46	6146.96	1	288683602	2	6.93
32	hls4ml	0.06	3954	1308	0	0	0	0	0	3.58	6343.28	1	279563880	2	7.15
64	hls4ml	0.07	7188	1418	0	0	0	0	0	5.28	6248.56	1	189501610	2	10.55
128	hls4ml	0.12	13323	2310	0	0	0	0	0	5.04	6475.29	1	198530871	2	10.07
256	hls4ml	0.17	24828	3999	0	0	0	0	0	5.12	19991.09	1	195274360	2	10.24
512	hls4ml	0.44	46678	7097	0	0	0	0	0	5.28	21835.80	1	189465706	2	10.56
16	finn	0.18	2627	542	0	0	0	0	0	3.66	7889.17	/	157703832	4	14.64
32	finn	0.20	4949	801	0	0	0	0	0	2.97	7912.18	/	142328494	4	11.90
64	finn	0.42	8717	1321	0	0	0	0	0	2.48	7956.50	/	132978723	4	9.92
128	finn	0.42	16320	2352	0	0	0	0	0	2.42	22715.79	/	131839156	4	9.66
256	finn	0.57	30602	4397	0	0	0	0	0	1.45	23554.76	/	116945386	4	5.80
512	finn	1.09	57984	9496	0	0	0	0	0	1.13	23722.59	/	112790435	5	5.67

B.3 Fully Connected Layer - Input Bitwidth

Table B.4: Single fully connected layer experiment in which the number of bits used to quantize the input features is varied. The number of input and output features equals 32. The weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits.

Input Bitwidth	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
1	chisel4ml	0.06	1899	0	0	0	0	0	0	5.10	5374.30	1	196001568	2	10.20
2	chisel4ml	0.04	4230	0	0	0	0	0	0	6.63	5013.75	1	150897842	2	13.25
3	chisel4ml	0.04	6000	0	0	0	0	0	0	6.82	5045.54	1	146541617	2	13.65
4	chisel4ml	0.04	7587	0	0	0	0	0	0	7.05	5028.15	1	141763538	2	14.11
5	chisel4ml	0.04	9139	0	0	0	0	0	0	6.95	5007.11	1	143988480	2	13.89
6	chisel4ml	0.04	10538	0	0	0	0	0	0	6.94	4909.21	1	144092219	2	13.88
7	chisel4ml	0.04	12302	0	0	0	0	0	0	7.36	4904.11	1	135906496	2	14.72
1	hls4ml	0.07	2128	902	0	0	0	0	0	3.81	6668.94	1	262743037	2	7.61
2	hls4ml	0.07	4208	1678	0	0	0	0	0	3.87	6364.29	1	258131130	2	7.75
3	hls4ml	0.07	6015	1965	0	0	0	0	0	4.18	6275.71	1	239463601	2	8.35
4	hls4ml	0.07	7780	2254	0	0	0	0	0	3.59	6275.77	1	278241513	2	7.19
5	hls4ml	0.09	9427	2443	0	0	0	0	0	3.58	6392.90	1	279407655	2	7.16
6	hls4ml	0.09	10437	2584	0	0	0	0	0	3.54	6276.13	1	282406099	2	7.08
7	hls4ml	0.09	11330	2956	0	0	0	0	0	3.59	6367.86	1	278862242	2	7.17
1	finn	0.20	2551	654	0	0	0	0	0	4.66	8334.82	/	187125748	4	18.62
2	finn	0.29	5141	1100	0	0	0	0	0	2.95	7917.99	/	141924496	5	14.77
3	finn	0.24	7089	1215	0	0	0	0	0	3.05	7927.76	/	143822810	5	15.23
4	finn	0.29	10899	1497	0	0	0	0	0	1.73	7971.62	/	120962864	5	8.67
5	finn	0.29	12969	1771	0	0	0	0	0	2.18	7981.22	/	127828198	5	10.88
6	finn	0.32	14460	1903	0	0	0	0	0	2.17	7989.09	/	127779197	5	10.87
7	finn	0.33	16038	1742	0	0	0	0	0	1.84	8018.29	/	122609122	5	9.22

B.4 Fully Connected Layer - Weights Bitwidth

Table B.5: Single fully connected layer experiment in which the number of bits used to quantize the weight parameters is varied. The number of input and output features equals 32. The inputs and outputs are quantized to 4 bits and the bias is quantized to 8 bits.

Weights Bitwidth	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
1	chisel4ml	0.04	2430	0	0	0	0	0	0	6.25	5492.52	1	160102465	2	12.49
2	chisel4ml	0.04	3481	0	0	0	0	0	0	6.59	4938.13	1	151676020	2	13.19
3	chisel4ml	0.04	5855	0	0	0	0	0	0	6.64	5012.46	1	150489089	2	13.29
4	chisel4ml	0.04	7587	0	0	0	0	0	0	7.05	5022.91	1	141763538	2	14.11
5	chisel4ml	0.04	9362	0	0	0	0	0	0	7.25	5054.30	1	137912012	2	14.50
6	chisel4ml	0.04	10834	0	0	0	0	0	0	7.14	4947.73	1	140075640	2	14.28
7	chisel4ml	0.05	12723	0	0	0	0	0	0	7.26	4968.21	1	137684152	2	14.53
1	hls4ml	0.06	1663	963	0	0	0	0	0	3.49	6675.05	1	286286859	3	10.48
2	hls4ml	0.07	4452	1290	0	0	0	0	0	4.78	6296.07	1	209161263	2	9.56
3	hls4ml	0.07	6253	1754	0	0	0	0	0	3.50	6314.57	1	286123032	2	6.99
4	hls4ml	0.07	7780	2254	0	0	0	0	0	3.59	6420.68	1	278241513	2	7.19
5	hls4ml	0.08	9154	2552	0	0	0	0	0	4.21	6326.47	1	237755587	2	8.41
6	hls4ml	0.10	10262	2895	0	0	0	0	0	3.87	6472.08	1	258264462	2	7.74
7	hls4ml	0.09	12023	3137	0	0	0	0	0	4.11	6491.27	1	243072435	2	8.23
1	finn	0.22	5861	3018	0	0	0	0	0	3.65	8494.17	/	157579577	5	18.27
2	finn	0.24	5616	2834	0	0	0	0	0	4.53	8024.34	/	182848784	5	22.66
3	finn	0.24	7854	1062	0	0	0	0	0	2.82	7943.75	/	139217597	4	11.27
4	finn	0.28	10899	1497	0	0	0	0	0	1.73	7974.62	/	120962864	5	8.67
5	finn	0.28	12118	2915	0	0	0	0	0	3.76	8140.85	/	160359204	7	26.35
6	finn	0.34	12094	3586	0	0	0	0	0	3.63	8301.42	/	156961230	8	29.03
7	finn	0.39	12240	3833	0	0	0	0	0	3.52	8503.14	/	154440154	8	28.20

B.5 Convolution Layer - Input Channels

Table B.6: Single convolution layer experiment in which the number of input channels of the convolution layer is varied. The input size is set to an 8 by 8 square and the kernel size is set to 3 by 3. The number of output channels is set to 1. The inputs, weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits.

Input Channels	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
1	chisel4ml	0.04	2825	0	0	0	0	0	0	5.91	4869.48	1	169118890	2	11.83
2	chisel4ml	0.04	5049	0	0	0	0	0	0	6.67	5027.53	1	149857635	2	13.35
4	chisel4ml	0.05	10803	0	0	0	0	0	0	7.18	5053.69	1	139178844	2	14.37
8	chisel4ml	0.06	20931	0	0	0	0	0	0	7.46	5445.70	1	134084204	2	14.92
16	chisel4ml	0.11	41850	0	0	0	0	0	0	8.25	21043.68	1	121270919	2	16.49
1	hls4ml	0.08	4475	2189	0	0	0	0	0	3.60	6950.16	39	7112628	41	147.81
2	hls4ml	0.12	10536	11543	0	0	0	0	0	4.17	8549.89	131	1832354	173	720.72
4	hls4ml	0.25	21194	23527	0	0	0	0	0	4.34	27303.00	259	889632	302	1310.68
8	hls4ml	0.62	45043	44329	0	0	0	0	0	4.33	46731.58	515	448647	558	2415.02
16	hls4ml	2.48	85650	86532	0	0	0	0	0	4.86	125085.83	1027	200187	1072	5214.21
1	finn	0.20	721	653	0	0	0	0	0	6.01	7884.26	/	719650	359	2156.51
2	finn	0.20	920	890	0	0	0	0	0	4.54	7879.50	/	525908	359	1628.42
4	finn	0.20	1240	1405	0	0	0	0	0	5.85	7884.56	/	692424	359	2100.15
8	finn	0.21	2007	2281	0	0	0	0	0	5.86	7889.65	/	693260	359	2101.95
16	finn	0.21	3332	3999	0	0	0	0	0	4.74	7915.57	/	546616	359	1702.74

B.6 Convolution Layer - Output Channels

Table B.7: Single convolution layer experiment in which the number of output channels of the convolution layer is varied. The input size is set to an 16 by 16 square and the kernel size is set to 3 by 3. The number of input channels is set to 1. The inputs, weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits.

Output Channels	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
1	chisel4ml	0.06	16279	0	0	0	0	0	0	5.96	6242.02	1	167728950	2	11.92
2	chisel4ml	0.08	28287	0	0	0	0	0	0	6.16	5237.15	1	162443144	2	12.31
4	chisel4ml	0.12	58981	0	0	0	0	0	0	6.61	20912.45	1	151285930	2	13.22
8	chisel4ml	0.19	119582	0	0	0	0	0	0	7.04	21368.27	1	142085819	2	14.08
16	chisel4ml	0.26	228488	0	0	0	0	0	0	7.01	25033.39	1	142551674	2	14.03
1	hls4ml	0.35	21115	10257	0	0	0	0	0	4.61	32767.98	199	1090048	201	926.61
2	hls4ml	0.46	33238	16381	0	0	0	0	0	4.71	33985.73	395	537048	397	1871.46
4	hls4ml	1.01	87529	0	0	0	0	0	1	6.13	36597.00	787	207351	790	4841.12
8	hls4ml	3.35	164981	0	0	0	0	0	1	6.57	43828.95	1571	96826	1574	10347.48
16	hls4ml	10.47	243864	114357	0	0	0	0	1	5.97	70061.71	3139	53362	3141	18751.77
1	finn	0.20	750	663	0	0	0	0	0	6.52	7881.13	/	158539	1823	11884.14
2	finn	0.20	869	688	0	0	0	0	0	5.66	7875.20	/	127277	1823	10325.47
4	finn	0.21	1072	824	0	0	0	0	0	5.55	7889.87	/	124100	1823	10123.12
8	finn	0.21	1388	882	0	0	0	0	0	4.54	7878.56	/	101168	1823	8285.53
16	finn	0.24	2100	1311	0	0	0	0	0	5.60	7894.26	/	125454	1823	10210.62

B.7 Convolution Layer - Input Bitwidth

Table B.8: Single convolution layer experiment in which the number of bits used to quantize the input is varied. The input size is set to an 16 by 16 square and the kernel size is set to 3 by 3. The number of input and output channels is set to 1. The weights and outputs are quantized to 4 bits and the bias is quantized to 8 bits.

Input Bitwidth	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
1	chisel4ml	0.07	1928	0	0	0	0	0	0	3.96	5436.11	1	252270433	2	7.93
2	chisel4ml	0.05	6710	0	0	0	0	0	0	5.41	5049.30	1	184774575	2	10.82
3	chisel4ml	0.05	11958	0	0	0	0	0	0	5.88	6264.07	1	170039108	2	11.76
4	chisel4ml	0.06	16279	0	0	0	0	0	0	5.96	6256.97	1	167728950	2	11.92
5	chisel4ml	0.06	16669	0	0	0	0	0	0	6.58	5174.25	1	151952590	2	13.16
6	chisel4ml	0.07	20123	0	0	0	0	0	0	6.54	5182.20	1	152905198	2	13.08
7	chisel4ml	0.06	25487	0	0	0	0	0	0	6.62	5078.06	1	150943396	2	13.25
1	hls4ml	0.30	10628	7442	0	0	0	0	0	4.64	32633.82	2	107735401	6	27.85
2	hls4ml	0.33	12777	7545	0	0	0	0	0	4.41	32751.16	199	1140001	201	886.01
3	hls4ml	0.34	15832	8509	0	0	0	0	0	4.60	32756.73	199	1091469	201	925.40
4	hls4ml	0.35	21115	10257	0	0	0	0	0	4.61	32759.67	199	1090048	201	926.61
5	hls4ml	0.34	22827	11417	0	0	0	0	0	4.61	33058.30	199	1089103	201	927.41
6	hls4ml	0.33	26291	12577	0	0	0	0	0	4.59	32812.04	199	1094560	201	922.79
7	hls4ml	0.31	28840	13737	0	0	0	0	0	4.59	32824.75	199	1094560	201	922.79
1	finn	0.21	674	542	0	0	0	0	0	6.42	8355.75	/	154112	1827	11727.51
2	finn	0.20	675	529	0	0	0	0	0	6.12	7876.45	/	142053	1823	11147.65
3	finn	0.19	726	586	0	0	0	0	0	6.47	7880.46	/	156560	1823	11803.92
4	finn	0.20	750	663	0	0	0	0	0	6.52	7876.09	/	158539	1823	11884.14
5	finn	0.20	802	720	0	0	0	0	0	6.68	7875.18	/	166478	1823	12186.75
6	finn	0.20	830	777	0	0	0	0	0	6.02	7885.89	/	138767	1823	10979.93
7	finn	0.19	961	872	0	0	0	0	0	5.86	7895.25	/	133335	1823	10684.60

B.8 Convolution Layer - Weights Bitwidth

Table B.9: Single convolution layer experiment in which the number of bits used to quantize the weight parameters is varied. The input size is set to an 16 by 16 square and the kernel size is set to 3 by 3. The number of input and output channels is set to 1. The inputs and outputs are quantized to 4 bits, and the bias is quantized to 8 bits.

Weights Bitwidth	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
1	chisel4ml	0.07	10569	0	0	0	0	0	0	5.25	5698.58	1	190439916	2	10.50
2	chisel4ml	0.05	7025	0	0	0	0	0	0	5.33	5078.58	1	187793427	2	10.65
3	chisel4ml	0.05	11259	0	0	0	0	0	0	5.71	5085.97	1	175162024	2	11.42
4	chisel4ml	0.06	16279	0	0	0	0	0	0	5.96	6254.52	1	167728950	2	11.92
5	chisel4ml	0.06	18864	0	0	0	0	0	0	6.08	5165.32	1	164527805	2	12.16
6	chisel4ml	0.06	21824	0	0	0	0	0	0	6.42	5260.39	1	155884645	2	12.83
7	chisel4ml	0.07	24513	0	0	0	0	0	0	6.30	5555.53	1	158654608	2	12.61
1	hls4ml	0.33	18421	8868	0	0	0	0	0	4.54	33582.09	4	55078211	10	47.21
2	hls4ml	0.24	12848	7654	0	0	0	0	0	4.14	23553.75	199	1214678	199	823.26
3	hls4ml	0.25	14781	9226	0	0	0	0	0	4.41	25140.44	199	1140777	201	885.41
4	hls4ml	0.35	21115	10257	0	0	0	0	0	4.61	32781.93	199	1090048	201	926.61
5	hls4ml	0.36	21667	11657	0	0	0	0	0	4.62	33102.11	199	1087689	201	928.62
6	hls4ml	0.35	23908	13314	0	0	0	0	0	4.61	33430.50	199	1090048	244	1124.84
7	hls4ml	0.33	20619	15916	0	0	0	0	0	4.61	33277.78	199	1090048	331	1525.91
1	finn	0.19	740	642	0	0	0	0	0	6.34	8360.01	/	150662	1823	11552.35
2	finn	0.19	688	575	0	0	0	0	0	6.47	7886.77	/	156516	1823	11802.10
3	finn	0.19	711	594	0	0	0	0	0	6.42	7874.22	/	154371	1823	11712.77
4	finn	0.19	750	663	0	0	0	0	0	6.52	7880.11	/	158539	1823	11884.14
5	finn	0.20	794	663	0	0	0	0	0	6.34	7876.51	/	150662	1823	11552.35
6	finn	0.19	772	751	0	0	0	0	0	6.65	7888.33	/	164591	1826	12137.42
7	finn	0.20	826	696	0	0	0	0	0	6.04	7880.39	/	139503	1826	11036.34

B.9 Maximum Pooling Layer - Input Size

Table B.10: Single maximum pooling layer experiment in which the input size whilst keeping it square is varied. The size of the kernel is 3 by 3 and the number of channels is set to 3. The inputs to the layer are quantized to 4 bits.

Input Size	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
4	chisel4ml	0.03	102	0	0	0	0	0	0	5.12	4844.59	1	195350654	2	10.24
8	chisel4ml	0.03	408	0	0	0	0	0	0	5.12	4857.20	1	195350654	2	10.24
12	chisel4ml	0.04	1632	0	0	0	0	0	0	5.12	4895.68	1	195350654	2	10.24
16	chisel4ml	0.04	2550	0	0	0	0	0	0	5.12	5011.13	1	195350654	2	10.24
4	hls4ml	0.05	544	774	0	0	0	0	0	3.49	6126.43	1	286697247	4	13.95
8	hls4ml	0.06	2255	0	0	0	0	0	0	3.50	6521.07	1	285714285	5	17.50
12	hls4ml	0.15	9437	0	0	0	0	0	0	3.56	20677.07	1	280662363	5	17.82
16	hls4ml	0.25	13055	0	0	0	0	0	0	3.58	23729.15	1	279095729	5	17.91
4	finn	0.20	1721	1458	0	0	0	0	0	5.94	7879.14	/	10267783	68	404.06
8	finn	0.20	1812	1739	0	0	0	0	0	5.37	7886.82	/	3002353	116	623.38
12	finn	0.20	1865	1819	0	0	0	0	0	5.46	7892.77	/	1223151	224	1222.59
16	finn	0.20	1873	1820	0	0	0	0	0	5.40	7896.15	/	754502	332	1792.14

B.10 Maximum Pooling Layer - Channels

Table B.11: Single maximum pooling layer experiment in which the number of channels of the input tensor is varied. The input size is set to an 8 by 8 square and the kernel size is set to 3 by 3. The inputs to the layer are quantized to 4 bits.

Channels	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
1	chisel4ml	0.03	136	0	0	0	0	0	0	5.12	5104.78	1	195350654	2	10.24
2	chisel4ml	0.03	272	0	0	0	0	0	0	5.12	4848.34	1	195350654	2	10.24
4	chisel4ml	0.03	544	0	0	0	0	0	0	5.12	4856.03	1	195350654	2	10.24
8	chisel4ml	0.04	1088	0	0	0	0	0	0	5.12	4874.72	1	195350654	2	10.24
16	chisel4ml	0.04	2176	0	0	0	0	0	0	5.12	4978.88	1	195350654	2	10.24
1	hls4ml	0.05	723	1030	0	0	0	0	0	3.49	6147.63	1	286697247	4	13.95
2	hls4ml	0.06	1505	0	0	0	0	0	0	3.49	6381.68	1	286697247	5	17.44
4	hls4ml	0.07	3005	0	0	0	0	0	0	3.51	6695.36	1	284738041	5	17.56
8	hls4ml	0.11	6293	0	0	0	0	0	0	3.54	7627.02	1	282167042	5	17.72
16	hls4ml	0.25	11141	0	0	0	0	0	0	3.58	22317.70	1	279642058	5	17.88
1	finn	0.20	1725	1603	0	0	0	0	0	5.91	7891.29	/	3393327	116	685.21
2	finn	0.20	1774	1671	0	0	0	0	0	5.93	7893.61	/	3415020	116	688.23
4	finn	0.20	1863	1811	0	0	0	0	0	5.36	7886.50	/	2992005	116	621.53
8	finn	0.20	2037	2087	0	0	0	0	0	5.10	7896.95	/	2832732	116	591.25
16	finn	0.20	2359	2631	0	0	0	0	0	5.92	7895.21	/	3404139	116	686.72

B.11 Maximum Pooling Layer - Kernel Size

Table B.12: Single maximum pooling layer experiment in which the size of the square kernel is varied. The number of channels is set to 3 and input size to an 8 by 8 square. The inputs to the layer are quantized to 4 bits.

Kernel Size	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
2	chisel4ml	0.03	576	0	0	0	0	0	0	4.06	4928.21	1	246426811	2	8.12
3	chisel4ml	0.03	408	0	0	0	0	0	0	5.12	4850.75	1	195350654	2	10.24
4	chisel4ml	0.03	912	0	0	0	0	0	0	5.10	4858.77	1	196039992	2	10.20
5	chisel4ml	0.03	342	0	0	0	0	0	0	5.61	4999.40	1	178253119	2	11.22
6	chisel4ml	0.03	444	0	0	0	0	0	0	6.05	4968.07	1	165207335	2	12.11
7	chisel4ml	0.03	678	0	0	0	0	0	0	6.09	4857.93	1	164203612	2	12.18
2	hls4ml	0.08	3653	3842	0	0	0	0	0	3.56	6803.02	1	280898876	1	3.56
3	hls4ml	0.06	2255	0	0	0	0	0	0	3.50	6414.88	1	285714285	5	17.50
4	hls4ml	0.08	4338	0	0	0	0	0	0	3.51	6806.30	1	284738041	7	24.58
5	hls4ml	0.06	1743	0	0	0	0	0	0	3.51	6443.64	1	284738041	10	35.12
6	hls4ml	0.06	2528	0	0	0	0	0	0	3.49	6539.90	1	286286859	14	48.90
7	hls4ml	0.07	3485	0	0	0	0	0	0	3.51	6708.77	1	284738041	18	63.22
2	finn	0.19	701	606	0	0	0	0	0	6.39	7872.10	/	3464523	89	568.89
3	finn	0.20	1812	1739	0	0	0	0	0	5.37	7889.29	/	3002353	116	623.38
4	finn	0.19	707	606	0	0	0	0	0	6.16	7880.32	/	2711261	105	646.59
5	finn	0.19	1765	1536	0	0	0	0	0	5.71	7885.45	/	2913752	124	708.04
6	finn	0.20	1756	1537	0	0	0	0	0	6.01	7890.11	/	2612002	140	841.68
7	finn	0.20	1727	1536	0	0	0	0	0	6.04	7893.58	/	2251846	156	941.46

B.12 Maximum Pooling Layer - Input Bitwidth

Table B.13: Single maximum pooling layer experiment in which the number of bits used to quantize the input is varied. The number of channels is set to 3 and the input size to an 8 by 8 square. The kernel size is set to 2 by 2.

Input Bitwidth	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
1	chisel4ml	0.03	48	0	0	0	0	0	0	3.31	5438.35	1	301932367	2	6.62
2	chisel4ml	0.03	144	0	0	0	0	0	0	3.58	4910.89	1	279407655	2	7.16
3	chisel4ml	0.03	288	0	0	0	0	0	0	3.62	4854.67	1	276014352	2	7.25
4	chisel4ml	0.03	576	0	0	0	0	0	0	4.06	4865.77	1	246426811	2	8.12
5	chisel4ml	0.04	864	0	0	0	0	0	0	4.34	4865.87	1	230467849	2	8.68
6	chisel4ml	0.03	1056	0	0	0	0	0	0	4.55	5015.18	1	219635405	2	9.11
7	chisel4ml	0.03	1440	0	0	0	0	0	0	4.87	4873.32	1	205254515	2	9.74
1	hls4ml	0.06	1396	2305	0	0	0	0	0	4.15	7079.86	1	240905805	0	0.00
2	hls4ml	0.07	2838	3458	0	0	0	0	0	3.45	6936.70	1	289519397	1	3.45
3	hls4ml	0.08	3125	3650	0	0	0	0	0	3.45	6820.96	1	289687137	1	3.45
4	hls4ml	0.08	3653	3842	0	0	0	0	0	3.56	6796.21	1	280898876	1	3.56
5	hls4ml	0.07	3941	4035	0	0	0	0	0	4.25	6861.33	1	235571260	1	4.25
6	hls4ml	0.06	4277	4227	0	0	0	0	0	4.24	6772.33	1	235849056	1	4.24
7	hls4ml	0.06	3989	4418	0	0	0	0	0	4.62	6675.05	1	216497077	1	4.62
1	finn	0.19	623	453	0	0	0	0	0	6.57	8350.76	/	3644314	89	584.73
2	finn	0.19	650	504	0	0	0	0	0	6.54	7868.97	/	3612716	89	582.06
3	finn	0.05	692	555	0	0	0	0	0	6.43	3889.89	/	3498460	89	572.00
4	finn	0.19	701	606	0	0	0	0	0	6.39	7869.41	/	3464523	89	568.89
5	finn	0.19	754	657	0	0	0	0	0	6.77	7872.25	/	3867574	89	602.35
6	finn	0.19	788	706	0	0	0	0	0	6.03	7875.84	/	3151790	89	537.03
7	finn	0.19	807	759	0	0	0	0	0	6.16	7889.51	/	3256904	89	548.42

B.13 Jet Tagging MLP - Bitwidth

Table B.14: The multilayer perceptron experiment in which the number of bits used to quantize the weights and activations is varied. The inputs to the MLP are quantized to 8 bits. The global prune rate is set to 0.5.

BW	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
1	chisel4ml	0.17	4145	0	0	0	0	0	0	4.31	6060.64	1	231910946	5	21.56
2	chisel4ml	0.10	5397	0	0	0	0	0	0	6.97	7072.87	1	143513203	5	34.84
3	chisel4ml	0.09	12623	0	0	0	0	0	0	13.12	6634.65	1	76219512	5	65.60
4	chisel4ml	0.09	17770	0	0	0	0	0	0	14.44	6585.63	1	69252077	5	72.20
5	chisel4ml	0.15	25597	0	0	0	0	0	0	15.53	7159.55	1	64391500	5	77.65
6	chisel4ml	0.12	36497	0	0	0	0	0	0	17.13	6595.67	1	58377116	5	85.65
7	chisel4ml	0.15	43857	0	0	0	0	0	0	18.86	6777.89	1	53022269	5	94.30
1	hls4ml	0.11	8	19	0	0	0	0	0	3.57	7272.27	1	279955207	18	64.30
2	hls4ml	0.14	5963	2164	0	0	0	0	0	3.76	8165.12	1	265745415	8	30.10
3	hls4ml	0.11	11376	3795	0	0	0	0	0	3.52	7969.41	1	284495021	8	28.12
4	hls4ml	0.12	15542	5269	0	0	0	0	0	3.45	7916.29	1	289771080	10	34.51
5	hls4ml	0.14	22309	7877	0	0	0	0	0	3.56	8143.49	1	280977802	13	46.27
6	hls4ml	0.18	30606	10472	0	0	0	0	0	3.61	20657.02	1	277085065	13	46.92
7	hls4ml	0.18	36052	10526	0	0	0	0	0	3.45	20625.16	1	289519397	12	41.45
1	finn	0.36	7769	1786	0	0	0	0	0	3.88	8394.37	/	163452108	15	58.23
2	finn	0.37	6216	1939	0	0	0	0	0	3.85	9697.91	/	162469536	13	49.98
3	finn	0.40	13742	2835	0	0	0	0	0	2.59	9337.46	/	134934556	13	33.66
4	finn	0.40	22726	3732	0	0	0	0	0	2.47	23655.87	/	132855055	13	32.15
5	finn	0.51	45443	8897	0	0	0	0	0	1.11	23456.62	/	112447992	27	29.89
6	finn	0.78	78958	12373	0	0	0	0	0	1.76	23524.20	/	121344497	29	51.01
7	finn	1.25	129322	14635	0	0	0	0	0	1.03	23979.91	/	111482720	30	30.90

B.14 Jet Tagging MLP - Prune Rate

Table B.15: The multilayer perceptron experiment in which the global prune rate is varied. The weights and activations are quantized to 4 bits. The inputs to the MLP are quantized to constant 8 bits.

Prune Rate	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
0.00	chisel4ml	0.12	19683	0	0	0	0	0	0	14.58	7552.09	1	68587105	5	72.90
0.20	chisel4ml	0.13	19898	0	0	0	0	0	0	14.39	7659.95	1	69492703	5	71.95
0.40	chisel4ml	0.12	17991	0	0	0	0	0	0	14.30	7481.46	1	69930069	5	71.50
0.60	chisel4ml	0.09	13642	0	0	0	0	0	0	13.32	7371.49	1	75075075	5	66.60
0.80	chisel4ml	0.05	6	0	0	0	0	0	0	0.45	6940.96	1	2247191011	5	2.23
0.00	hls4ml	0.14	24320	7608	0	0	0	0	0	3.81	21665.20	1	262467191	12	45.72
0.20	hls4ml	0.14	21849	6720	0	0	0	0	0	3.45	8473.38	1	289603243	13	44.89
0.40	hls4ml	0.14	18911	5692	0	0	0	0	0	4.24	8582.08	1	235793444	9	38.17
0.60	hls4ml	0.13	14014	4698	0	0	0	0	0	3.60	8369.87	1	277623542	8	28.82
0.80	hls4ml	0.08	1	0	0	0	0	0	0	3.34	7963.53	1	299401197	0	0.00
0.00	finn	0.63	28215	3559	0	0	0	0	0	2.23	25474.13	/	128650456	15	33.41
0.20	finn	0.62	26424	3260	0	0	0	0	0	1.17	9860.65	/	113288773	15	17.59
0.40	finn	0.55	22434	3024	0	0	0	0	0	1.89	9963.73	/	123350191	13	24.61
0.60	finn	0.51	17380	3017	0	0	0	0	0	2.72	9914.61	/	137400384	13	35.39
0.80	finn	0.32	7455	1936	0	0	0	0	0	3.33	8385.02	/	149970005	13	43.32

B.15 MNIST CNN - Bitwidth

Table B.16: The convolutional neural network experiment in which the number of bits used to quantize the weights and activations is varied. The inputs to the CNN are quantized to constant 8 bits. The global prune rate is set to 0.5.

BW	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
1	chisel4ml	0.72	254776	0	0	0	0	0	0	9.80	29947.97	1	102019995	7	68.61
2	chisel4ml	0.82	176186	0	0	0	0	0	0	12.13	29941.73	1	82440230	7	84.91
3	chisel4ml	0.97	183505	0	0	0	0	0	0	15.54	29294.13	1	64350064	7	108.78
4	chisel4ml	0.96	408325	0	0	0	0	0	0	18.73	30752.34	1	53390282	7	131.11
5	chisel4ml	1.47	527912	0	0	0	0	0	0	20.88	31151.16	1	47892720	7	146.16
6	chisel4ml	1.77	772409	0	0	0	0	0	0	21.51	33566.62	1	46490004	7	150.57
7	chisel4ml	2.24	1001624	0	0	0	0	0	0	24.14	36497.83	1	41425020	7	168.98
1	hls4ml	/	/	/	/	/	/	/	/	/	/	/	/	/	/
2	hls4ml	88.95	321077	0	0	0	0	0	0	5.27	310908.00	2	94876660	17	93.81
3	hls4ml	70.46	478886	0	0	0	0	0	0	5.30	313045.41	2	94268476	19	100.78
4	hls4ml	68.34	659972	0	0	0	0	0	0	5.30	650586.27	2	94268476	20	106.08
5	hls4ml	71.10	793225	0	0	0	0	0	0	5.14	354242.05	19	10241599	37	190.14
6	hls4ml	67.99	880217	0	0	0	0	0	3	5.35	363909.95	166	1126841	483	2582.12
7	hls4ml	71.40	1006987	0	0	0	0	0	0	5.14	371721.00	309	629742	885	4550.07
1	finn	/	/	/	/	/	/	/	/	/	/	/	/	/	/
2	finn	9.79	24807	7637	0	0	0	0	0	2.26	25680.19	/	20935	6346	14316.58
3	finn	14.14	54031	11171	0	0	0	0	0	1.37	26028.34	/	18784	6346	8687.67
4	finn	10.43	111581	17107	0	0	0	0	0	0.33	28499.03	/	16759	6348	2069.45
5	finn	14.59	196545	39043	0	0	0	0	0	0.25	24715.59	/	16625	6360	1577.28
6	finn	17.38	295565	74812	0	0	0	0	0	-0.10	34948.57	/	16049	6364	-649.13
7	finn	/	/	/	/	/	/	/	/	/	/	/	/	/	/

B.16 MNIST CNN - Prune Rate

Table B.17: The convolutional neural network experiment in which the global prune rate is varied. The weights and activations are quantized to 3 bits. The inputs to the CNN are quantized to constant 8 bits.

Prune Rate	Tool	Time	LUT	FF	BRAM	BRAM36	BRAM16	URAM	DSP	PD	PM	II	TP	LC	TL
0.70	chisel4ml	2.09	405915	0	0	0	0	0	0	18.14	37894.35	1	55126791	7	126.98
0.75	chisel4ml	0.90	408979	0	0	0	0	0	0	18.00	30806.71	1	55555555	7	126.00
0.80	chisel4ml	1.59	370822	0	0	0	0	0	0	17.91	41195.62	1	55834729	7	125.37
0.85	chisel4ml	0.96	302702	0	0	0	0	0	0	17.02	30504.37	1	58754406	7	119.14
0.88	chisel4ml	0.66	186171	0	0	0	0	0	0	16.79	30734.64	1	59559261	7	117.53
0.70	hls4ml	69.72	675012	0	0	0	0	0	0	5.31	354802.45	2	94232943	21	115.67
0.75	hls4ml	69.00	667311	0	0	0	0	0	0	5.31	357890.93	4	47116471	27	145.38
0.80	hls4ml	70.86	625354	0	0	0	0	0	0	5.14	353640.93	4	48647596	26	135.67
0.85	hls4ml	69.06	521908	0	0	0	0	0	0	5.14	333696.76	4	48647596	25	130.53
0.88	hls4ml	63.74	422315	309757	0	0	0	0	0	5.12	315405.59	2	97580015	19	101.46
0.70	finn	10.12	108772	17057	0	0	0	0	0	0.02	25007.39	/	16248	6348	139.66
0.75	finn	10.08	87901	16477	0	0	0	0	0	0.59	26934.58	/	17225	6347	3732.04
0.80	finn	9.97	63845	14670	0	0	0	0	0	0.88	25623.14	/	17769	6347	5559.97
0.85	finn	9.58	36007	13001	0	0	0	0	0	1.66	25074.52	/	19428	6347	10504.28
0.88	finn	9.33	17003	10702	0	0	0	0	0	2.62	25448.85	/	21953	6347	16597.41

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[2] J. Duarte, S. Han, P. Harris, *et al.*, "Fast inference of deep neural networks for real-time particle physics applications," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, Seaside, CA, USA: Association for Computing Machinery, 2019, p. 305, ISBN: 9781450361378. DOI: 10.1145/3289602.3293986. [Online]. Available: https://doi.org/10.1145/3289602.3293986.

[3] J. Vreča, I. Ivanov, G. Papa, and A. Biasizzo, "Detecting network intrusion using binarized neural networks," in *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, 2021, pp. 622–627. DOI: 10.1109/WF-IoT51360.2021.9595961.

[4] F. Jentzsch, Y. Umuroglu, A. Pappalardo, M. Blott, and M. Platzner, "Radioml meets finn: Enabling future rf applications with fpga streaming architectures," *IEEE Micro*, vol. 42, no. 6, pp. 125–133, 2022. DOI: 10.1109/MM.2022.3202091.

[5] Y. Wei, R. F. Forelli, C. Hansen, *et al.*, "Low latency optical-based mode tracking with machine learning deployed on fpgas on a tokamak," *Review of Scientific Instruments*, vol. 95, no. 7, p. 073 509, Jul. 2024, ISSN: 0034-6748. DOI: 10.1063/5.0190354. eprint: https://pubs.aip.org/aip/rsi/article-pdf/doi/10.1063/5.0190354/20033792/073509_1_5.0190354.pdf. [Online]. Available: https://doi.org/10.1063/5.0190354.

[6] A. M. Deiana, N. Tran, J. Agar, *et al.*, "Applications and techniques for fast machine learning in science," en, *Front Big Data*, vol. 5, p. 787 421, Apr. 2022.

[7] S. Eyerman, W. Heirman, and I. Hur, "Dram bandwidth and latency stacks: Visualizing dram bottlenecks," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022, pp. 322–331. DOI: 10.1109/ISPASS55109.2022.00045.

[8] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989, ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[9] Y. Le Cun, B. Boser, J. S. Denker, *et al.*, "Handwritten digit recognition with a back-propagation network," in *Proceedings of the 3rd International Conference on Neural Information Processing Systems*, ser. NIPS'89, Cambridge, MA, USA: MIT Press, 1989, pp. 396–404.

[10] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, K. Chaudhuri and R. Salakhutdinov, Eds., ser. Proceedings of Machine Learning Research, vol. 97, PMLR, Jun. 2019, pp. 6105–6114. [Online]. Available: https://proceedings.mlr.press/v97/tan19a.html.

[11] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15, Lille, France: JMLR.org, 2015, pp. 448–456.

[12] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980.

[13] B. Jacob, S. Kligys, B. Chen, *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2018.

[14] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019. DOI: 10.1109/IEEESTD.2019.8766229.

[15] D. A. Gudovskiy and L. Rigazio, "ShiftCNN: Generalized Low-Precision Architecture for Inference of Convolutional Neural Networks," 2017. arXiv: 1706.02393. [Online]. Available: http://arxiv.org/abs/1706.02393.

[16] M. Nagel, M. v. Baalen, T. Blankevoort, and M. Welling, "Data-free quantization through weight equalization and bias correction," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, Oct. 2019.

[17] I. Hubara, Y. Nahshan, Y. Hanani, R. Banner, and D. Soudry, "Improving post training neural quantization: Layer-wise calibration and integer programming," *CoRR*, vol. abs/2006.10518, 2020. arXiv: 2006.10518. [Online]. Available: https://arxiv.org/abs/2006.10518.

[18] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," *Advances in Neural Information Processing Systems*, no. Nips, pp. 4114–4122, 2016, ISSN: 10495258. arXiv: 1602.02505.

[19] Y. Bhalgat, J. Lee, M. Nagel, T. Blankevoort, and N. Kwak, "Lsq+: Improving low-bit quantization through learnable offsets and better initialization," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2020, pp. 2978–2985. DOI: 10.1109/CVPRW50498.2020.00356.

[20] Z. Yao, Z. Dong, Z. Zheng, *et al.*, "Hawq-v3: Dyadic neural network quantization," in *Proceedings of the 38th International Conference on Machine Learning*, M. Meila and T. Zhang, Eds., ser. Proceedings of Machine Learning Research, vol. 139, PMLR, Jul. 2021, pp. 11 875–11 886. [Online]. Available: https://proceedings.mlr.press/v139/yao21a.html.

[21] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:18268744.

[22] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=rJl-b3RcF7.

[23] A. K. Mishra, J. A. Latorre, J. Pool, *et al.*, "Accelerating sparse deep neural networks," *CoRR*, vol. abs/2104.08378, 2021. arXiv: 2104.08378. [Online]. Available: https://arxiv.org/abs/2104.08378.

[24] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[25] Martín Abadi, Ashish Agarwal, Paul Barham, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[26] A. Pappalardo, *Xilinx/brevitas*, 2023. DOI: 10.5281/zenodo.3333552. [Online]. Available: https://doi.org/10.5281/zenodo.3333552.

[27] C. N. Coelho, A. Kuusela, S. Li, *et al.*, "Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors," *Nature Machine Intelligence*, vol. 3, no. 8, pp. 675–686, Jun. 2021, ISSN: 2522-5839. DOI: 10.1038/s42256-021-00356-5. [Online]. Available: http://dx.doi.org/10.1038/s42256-021-00356-5.

[28] A. Pappalardo, Y. Umuroglu, M. Blott, *et al.*, "QONNX: Representing Arbitrary-Precision Quantized Neural Networks," in *4th Workshop on Accelerated Machine Learning (AccML) at HiPEAC 2022 Conference*, Jun. 2022. arXiv: 2206.07527 [cs.LG]. [Online]. Available: https://accml.dcs.gla.ac.uk/papers/2022/4thAccML_paper_1(12).pdf.

[29] Y. Umuroglu, H. Borras, V. Loncar, S. Summers, and J. Duarte, *Fastmachinelearning/qonnx*, Jun. 2022. DOI: 10.5281/zenodo.7622236. [Online]. Available: https://github.com/fastmachinelearning/qonnx.

[30] J. Bai, F. Lu, K. Zhang, *et al.*, *Onnx: Open neural network exchange*, https://github.com/onnx/onnx, 2019.

[31] J. Bachrach, H. Vo, B. Richards, *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12, San Francisco, California: Association for Computing Machinery, 2012, pp. 1216–1225, ISBN: 9781450311991. DOI: 10.1145/2228360.2228584. [Online]. Available: https://doi.org/10.1145/2228360.2228584.

[32] M. C. McFarland, A. C. Parker, and R. Camposano, "Tutorial on high-level synthesis," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, ser. DAC '88, Atlantic City, New Jersey, USA: IEEE Computer Society Press, 1988, pp. 330–336, ISBN: 0818688645.

[33] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on sdc formulation," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 433–438. DOI: 10.1145/1146909.1147025.

[34] J. Koening. "Chisel data hierarchy." (2025), [Online]. Available: https://github.com/chipsalliance/chisel.

[35] M. Wachs. "Width inference." (2025), [Online]. Available: https://www.chisel-lang.org/docs/explanations/width-inference (visited on 02/28/2025).

[36] S. Bailey, A. Izraelevitz, R. Lin, C. Markley, P. Rigge, and E. Wang, *Chisel bootcamp*, https://github.com/freechipsproject/chisel-bootcamp, Accessed: 2024-11-13.

[37] FastML Team, *Fastmachinelearning/hls4ml*, version v1.0.0, 2024. DOI: 10.5281/zenodo.1201549. [Online]. Available: https://github.com/fastmachinelearning/hls4ml.

[38] M. Blott, T. B. Preußer, N. J. Fraser, *et al.*, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, Dec. 2018, ISSN: 1936-7406. DOI: 10.1145/3242897. [Online]. Available: https://doi.org/10.1145/3242897.

[39] Y. Umuroglu, N. J. Fraser, G. Gambardella, *et al.*, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, Monterey, California, USA: Association for Computing Machinery, 2017, pp. 65–74, ISBN: 9781450343541. DOI: 10.1145/3020078.3021744. [Online]. Available: https://doi.org/10.1145/3020078.3021744.

[40] S. A. Alam, D. Gregg, G. Gambardella, T. Preusser, and M. Blott, "On the rtl implementation of finn matrix vector unit," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 6, Nov. 2023, ISSN: 1539-9087. DOI: 10.1145/3547141. [Online]. Available: https://doi.org/10.1145/3547141.

[41] J. Vreča and A. Biasizzo, "Towards deploying highly quantized neural networks on fpga using chisel," in *2023 26th Euromicro Conference on Digital System Design (DSD)*, 2023, pp. 161–167. DOI: 10.1109/DSD60849.2023.00032.

[42] J. Vreča and A. Biasizzo, "Generating direct logic circuit implementations of deeply quantized neural networks using chisel4ml," *Electronics*, vol. 14, no. 5, 2025, ISSN: 2079-9292. DOI: 10.3390/electronics14050849. [Online]. Available: https://www.mdpi.com/2079-9292/14/5/849.

[43] J. Vreča, R. Pilipović, and A. Biasizzo, "Hardware–software co-design of an audio feature extraction pipeline for machine learning applications," *Electronics*, vol. 13, no. 5, 2024, ISSN: 2079-9292. DOI: 10.3390/electronics13050875. [Online]. Available: https://www.mdpi.com/2079-9292/13/5/875.

[44] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012. DOI: 10.1109/MSP.2012.2211477.

Bibliography

Publications Related to the Thesis

Journal Articles

[41] J. Vreča and A. Biasizzo, "Generating direct logic circuit implementations of deeply quantized neural networks using chisel4ml," *Electronics*, vol. 14, no. 5, 2025, ISSN: 2079-9292. DOI: 10.3390/electronics14050849. [Online]. Available: https://www.mdpi.com/2079-9292/14/5/849.

[49] J. Vreča, R. Pilipović, and A. Biasizzo, "Hardware–software co-design of an audio feature extraction pipeline for machine learning applications," *Electronics*, vol. 13, no. 5, 2024, ISSN: 2079-9292. DOI: 10.3390/electronics13050875. [Online]. Available: https://www.mdpi.com/2079-9292/13/5/875.

Conference Paper

[40] J. Vreča and A. Biasizzo, "Towards deploying highly quantized neural networks on fpga using chisel," in *2023 26th Euromicro Conference on Digital System Design (DSD)*, 2023, pp. 161–167. DOI: 10.1109/DSD60849.2023.00032.

[43] J. Vreča, I. Ivanov, G. Papa, and A. Biasizzo, "Detecting network intrusion using binarized neural networks," in *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, 2021, pp. 622–627. DOI: 10.1109/WF-IoT51360.2021.9595961.

[44] J. Vreča, "Chisel4ml: Generating fast implementations of deeply quantized neural networks using chisel generators," in *2nd FPGA Developers' Forum (FDF) meeting*, May 2025.

[45] J. Vreča and A. Biasizzo, "Chisel4ml: A tool for generating fast direct logic implementations of deeply quantized neural networks," in *13th Prague Embedded Systems Workshop*, Jun. 2025.

[46] J. Vreča, "Chisel4ml: Using chisel for direct circuit implementation of deeply quantized neural networks," in *Fast Machine Learning for Science Conference 2025*, Aug. 2025.

Other Publications

Conference Paper

[50] I. Ivanov, J. Vreča, and A. Biasizzo, "Hardware acceleration of the yolo convolutional neural network on a field-programmable gate array," in *Proceedings of 57th International Conference on Microelectronics, Devices and Materials (MIDEM2022)*, Maribor, Slovenia, 2022. DOI: 10.1109/WF-IoT51360.2021.9595961.

Biography

Jure Vreča received in 2017 a Bachelor of Science degree in Electrical Engineering from the Faculty of Electrical Engineering, Computer Science and Informatics. In 2020 he received a Masters of Science degree from the same faculty. In both degrees he pursued a specialization in electronics.

In 2020, he began his PhD in Informatics and Communication Technologies at the Jožef Stefan International Postgraduate School in Ljubljana, Slovenia, under the supervision of Prof. Dr. Anton Biasizzo. Simultaneously he started a position as a research assistant at the Computer Science department of the Jožef Stefan Institute, where he contributed mainly to the european Distributed Artificial Intelligence Systems (DAIS) project.

His main research areas of interest include computer architecture, digital design and compiler design. Combining these approaches to the field of accelerating quantized neural network processing is a particular interest. He has presented his work at several international conferences and published papers in scientific journals with an impact factor.