Tampere University

Väinö-Waltteri Granat

Zero to DLA: Building Software Stack For Accelerating Deep Neural Networks On Custom RISC-V SoC

Faculty of Information Technology and Communication Sciences (ITC)
Master's thesis
December 2024

Use of artificial intelligence in this work

Artificial intelligence (AI) has been used in generating this work:

I hereby declare, that the AI-based applications used in generating this work are as follows:

Application	Version
OpenAi, ChatGPT-4 Turbo	April - November 2024
Microsoft Copilot.	April - November 2024

Purpose of the use of AI

Large language models were use as an aid for generating code for some of the Tizk diagrams presented in this thesis, with the purpose of improving the quality of the given diagrams.

Explain here *in detail*, for which purpose and how AI was utilized in writing this thesis.

Parts of this work, where AI was used

List here all chapters, sections, subsections, tables, figures and so forth, that were generated by an AI, or that an AI had a hand in generating.

The following figures were created with a help of AI: 2.1, 2.6, 2.8, 2.9

Acknowledgement of risks

I hereby acknowledge, that as the author of this work, I am fully responsible for the contents presented in this thesis. This includes the parts that were generated by an AI, in part or in their entirety. I therefore also acknowledge my responsibility in the case, where use of AI has resulted in ethical guidelines being breached.

Abstract

Väinö-Waltteri Granat: Zero to DLA: Building Software Stack For Accelerating Deep Neural Networks On Custom RISC-V SoC
Master's thesis
Tampere University
Master's Degree Programme in Signal Processing and Machine Learning
December 2024

Lorem ipsum

Keywords: DLA, Deep-Learning, SoC, Virtual Prototype.

The originality of this thesis has been checked using the Turnitin Originality Check service.

List of Abbreviations

DLA Deep Learning Accelerator

DLA-VP Headsail Deep Learning Accelerator Virtual Prototype

FFI Foreign Function Interface

SoC System on a chip

ISA Instruction set architecture

ML Machine Learning

AI Artifical Intelligence

DL Deep Learning

MPL Multilayer Perceptron

RGB Red Green Blue

DNN Deep Neural Network

CNN Convolutional Neural Network

CHW Channel Height Width

HWC Height Width Channel

AOT Ahead-of-Time runtime

BYOC Bring your own codegen

AUC Area under the curve

IC Image Classification

VWW Visual Wake Words

KWS Keyword Spotting

AD Anomaly Detection

LSB Least Significant Bit

MSB Most Significant Bit

RTL Register Level Transfer

List of Figures

List of Tables

Contents

1 Introduction

In recent years neural network based application have become more and more prominent in our everyday-life. The large driver for this has been the adoption of efficient accelerators in mobile device, that have enabled running neural network applications of mobile devices, such as smartphones.

This interest in neural networks has coincided with the industry's move to heterogeneous System-on-chip solutions being used in consumer and professional devices, to improve computational performance. More often these companies integrate their accelerators into SoCs, which include CPUs, GPUs, memory and other accelerators and peripherals in one package. Apple and Qualcomm have proved with their SoCs that they can attain desktop like performance in a smaller package than was previously possible. The industry moving towards SoCs has generated new interest in developing open-source SoCs.

The goal of this project was to build software support for the Deep-Learning Accelerator in the upcoming Headsail SoC from SocHub using a Renode based virtual prototype as the development platform. The goal was to use this concurrent development approach to have software support ready before the chip had been manufactured.

2 Background

This section covers the topics in Deep Learning that relate to the implementation of the project, with focus on performing inference on deep neural networks. In addition to this we discuss hardware approaches for building tightly integrated single chip systems with SoCs and for accelerating inference workloads with dedicated deep learning accelerators.

2.1 Machine Learning

Machine Learning is a field of computer science that researches algorithms to categorize data into distinct concepts so that yet unseen data can be categorized similarly. The central component of machine learning is the model, which is a function that distinguishes a concept from data.

To build a model we first need training data. Training data is a set of data, often with known associated categories, that corresponds with the unseen data we want to categorize with the model. For example if training a model to count the number of people in image, we would have a training data consisting of images with different amount of people, with the wanted categorization (number of people present in the image) associated with each. Model is then trained by choosing a particular function and changing its parameters so that it can categorize training data in a wanted manner. After training the performance can then be validated by feeding the model unseen data and seeing how well it can categorize it.

A particular models domain and codomain are defined by the given problem statement. For example, inputs of an image classifier are three-dimensional arrays, where the second and third dimensions correspond to the height and width of the image and the first dimension as the channel in RGB color space as shown in figure 2.1, this commonly known as CHW layout. The output of this model would be one or more labels from the codomain.

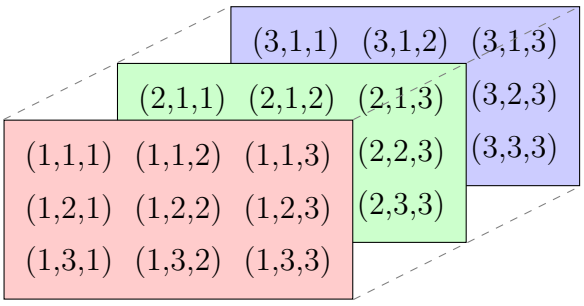

Figure 2.1 *RGB array*

2.1.1 Function and Model

In mathematics a function describes the relation between a domain X and codomain Y where

$$f \subseteq X \times Y, \tag{2.1}$$

meaning that for every element in the domain X there is exactly one corresponding element in codomain Y. Using arrow syntax this same mapping is expressed as $f : X \mapsto Y$. If we consider the discovered model as a function, we can view the domain X as the input of a model and codomain Y as the prediction space of the model. For a 10 class classifier codomain Y would be defined as

$$Y = \{0, 1, 2..., 9\}. \tag{2.2}$$

The purpose of a model then, is to map it's input to the codomain in such a manner in which useful information can be acquired from the mapping. The domain of the network depends on the problem statement. For an image binary classifier f using RGB images as input the formal definition would be the following

$$f : X \in \mathbb{Z}^{C \times H \times W} \mapsto \{0, 1\}. \tag{2.3}$$

Depending on the problem statement we give the elements in the codomain descriptive labels. For example if the goal of the model was to tell if an image has a person, 0 might be labeled "*No*" and 1 labeled as "*Yes*". The previous equation abstracts the model into singular function, but developers tend to think models as a series of multiple functions. By dividing the function into multiple consecutive domains and codomains we can get a better understanding what's happening in the model. One model might consist for to consecutive operations g and h

$$h : X \in \mathbb{Z}^{C \times H \times W} \xrightarrow{g} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_8 \end{bmatrix} \in \mathbb{Z}^8 \xrightarrow{f} \{0, 1\} \tag{2.4}$$

where g maps RGB image domain to a codomain of length 8 is acts as the domain for function h which then maps it to domain of the overall function.

2.2 Deep Learning

Deep learning is a subcategory of machine learning that focuses on using deep neural networks as the model. Deep neural networks are a specific case of multilayer perceptrons where there is at least one hidden layer between input and output layer

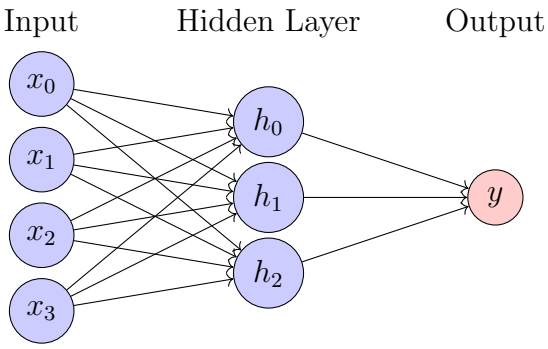

Figure 2.2 *Simple fully connected neural network with two layers.*

to introduce non-linearity. DNNs try to solve the problem of choosing a good model for a problem statement, by allowing the developers to discover a suitable function using a training algorithm.

DNN model is weighted graph where nodes are grouped into mostly sequential layers and vertices connect nodes of consecutive layers with simple operations. The vertices hold weights and biases signifying the amount of association between nodes. Essentially what the graph models is, is a N-dimensional space which the non-linear function inhabits, where each of the vertices introduces its own dimension. This means that increasing the amount of parameters in a model allows it to approximate more complex function, generally increasing the ability to perform more complex tasks. [36]

DNNs differ from other ML methods by their adaptable general architecture that can facilitate problems magnitudes more difficult than traditional methods allow. This ability to solve increasingly more difficult problems come with a cost of needing large amount of computing resource and training data.

2.2.1 Fully connected layer

Fully connected layer or dense layer is a common operation in neural networks. It maps each input element to each output element with and associated weight between each connection, where the weight represents the strength of the connection. Essentially this means that if an input value is high and the weight is high, the output is also higher. Fully connected layer is implemented as a matrix multiplication between layer's weights and layer's input,

$$y = x^T W \tag{2.5}$$

where y is the output of the layer, W is the weight matrix for that layer and x is the input.

Figure 2.2 shows a simple fully connected network. In the figure each node is

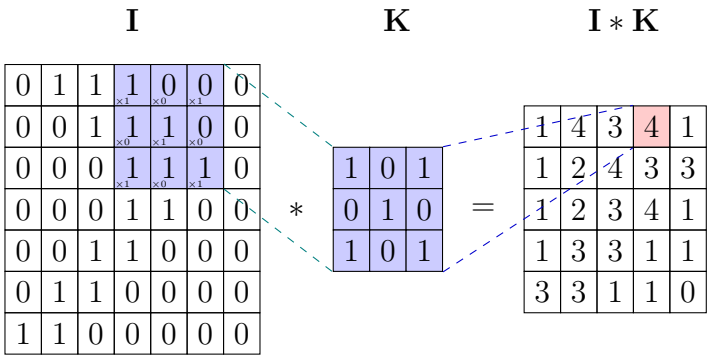

Figure 2.3 *Visualization of 2D convolution being done for 7x7 input with 3x3 kernel. Adapted from [32].*

an element in one layer's input x and the vertices are the weights that connect each input to each output element. The second layer in the figure is a hidden layer, since it doesn't have input or output nodes from the network.

Many classifier networks use a fully connected layer to do the final step in classification to squash the prediction dimensions to correspond with the expected codomain dimensions. This allows for a network to be used for different sized codomains. In such a case the last FC-layer is known as classifier and preceding set of layers is called a backbone. If for example we have a network architecture is good at classifying images, and we have two different problem definitions. From which first is to classify cars images from 20 different manufacturers and second is to classify animal images to ruminants and monogastrics. We can just modify the FC classifier to have 20 outputs for the first problem and two for the second and train with different datasets.

2.2.2 Convolutional neural networks

Convolutional neural networks are a type of neural networks that heavily utilize convolution operations. Convolution is used to extract features from data. Convolution is defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(t - x)g(t)dx, \tag{2.6}$$

where f and g are functions to be convolved. For computer science the discrete convolution is often more interesting

$$(f * g)(i) = \sum_{m} f(i - m)g(m). \tag{2.7}$$

For DNNs we often think about convolution in terms of inputs(f) and kernels(g), where input is the useful data we want to extract features from and kernels are the specifically selected values that can extract the wanted features from data.

When working with images, for example in a neural network used for classifying objects in images, we use the two-dimensional expansion of the convolution operation

$$Conv2D(i,j) = (K \star I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.8}$$

where I is the input and K is one kernel, m is the width of the kernel and n is the height.

In high-level deep learning frameworks convolution is often implemented as cross-correlation but still called convolution [14, 25, 27]. This is also what we have done in our implementation

$$Conv2D(i,j) = CrossCorrelation(I,K) = (K \star I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.9}$$

To produce output of one layer one needs to calculate $Conv2D$ for all the positions in the resultant output matrix for all the kernels.

The choice of kernel size determines what kind of features are extracted. Large kernels capture broad features, where small kernels capture finer details. The specific weights are found during training and that defines which features are to be extracted. Producing good feature maps is thus critical to the performance of the trained CNN.

2.2.3 Depthwise Separable Convolution

As seen in the previous section, convolution is a costly operation to calculate, when the size of inputs and amount of features grows. This is especially difficult for low power hardware that can't parallellize convolution and has to process the operation sequentially.

One common solution for this problem that is often seen in embedded applications, is to use grouped convolution. In grouped convolution we separate the input channels and filters into G groups so that the convolution is only performed among inputs and filters belonging to the same group. The groups essentially behave as completely independent convolution layers from in respect to the other groups. In the end after each input channels has been processed the resulting features get concatenated back into one tensor which has the equal size as the output of same convolution without groups with less calculation. [17, 37]

The computational cost of standard two-dimensional convolution is

$$\text{Cost}_{Conv2D} = K_x \cdot K_y \cdot I_x \cdot I_y \cdot C_{in} \cdot C_{out} \tag{2.10}$$

where K_x, K_y, I_x, I_y are the dimensions for single kernel and input channels, C_{in} is the number of input features and C_{out} is the number of output features. Whereas the computational complexity of grouped convolution is

$$\text{Cost}_{Grouped} = K_x \cdot K_y \cdot I_x \cdot I_y \cdot \frac{C_{in} \cdot C_{out}}{G} \tag{2.11}$$

where G is the number of groups. By setting the number of grouped to $G = C_{out}$ we arrive at the special case of grouped convolution called depthwise convolution.

In depthwise convolution each input channel only interacts with its corresponding set of feature kernels giving us a computational complexity which is not dependent on the number of input channels. This effectively removes any interaction across channels, which can limit the amount of features that a CNN can express. To introduce intra-channel interaction to depthwise convolution, we often follow depthwise layer with a pointwise convolution layer. Pointwise convolution is just standard 2D convolution with a filter of size 1×1, this ensures that output and input have same sized channels. Depthwise followed by pointwise is so common combination that it has its own name, depthwise separable convolution, which has the computational complexity of

$$\text{Cost}_{Separable} = K_x \cdot K_y \cdot I_x \cdot I_y \cdot C_{in} + I_x \cdot I_y \cdot C_{in} \cdot C_{out} \tag{2.12}$$

As shown in [37] depthwise separable convolutions can greatly reduce the amount of network parameters and thus reduce the total number of operations during inference without having a noticeable effect on the model accuracy. This makes it a popular layer structure in neural network targetted for mobile devices, like MobileNet [18].

2.2.4 Bias

In addition to $Conv2d$, bias is another important concept in neural networks. Bias is a constant value applied to output channel of the preceding operation. In CNNs when applied after $Conv2d$ the purpose of bias is to signify the importance of each extracted feature. If bias is small or negative it means that the feature is not important for the particular class it's being applied. If bias is large or positive it means that the feature is important. [14]

In mathematic notation bias is defined as such

$$y = x^T W + b \tag{2.13}$$

where if xA^T is the non-biased output of a particular channel in layer, b is the bias applied to the whole channel as a constant value. Many DL frameworks consider bias part of the convolution operation and fuse them together into one layer as is done with for example Tensorflow [25].

2.2.5 Activation functions

The combination of convolution and bias form one example of an affine transformation, a linear transformation between two space. To enable neural networks to recognize non-linear features, we need to introduce non-linear operations between the linear affine transformations. Traditionally we used sigmoid or tanh functions, which are defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.14}$$

and,

$$tanh(x) = \frac{e^x - e^{-x}}{e^x - e^{-x}}, \tag{2.15}$$

to introduce non-linearity. Both functions suffer from the fact that they are expensive to calculate and exhibit the vanishing gradient problem. [14]

Because of these problems DNNs have largely moved to using Rectified linear units (ReLU) for layer activations. ReLU is a simpler operation, moving negative values to zero and doing nothing for positive values, defined as

$$ReLU(x) = \begin{cases} 0, & \text{for} \leq 0 \\ x, & \text{otherwise.} \end{cases} \tag{2.16}$$

ReLU is essentially a special case of the clamp operation, show in equation 2.26, where the low bound is set to 0, and high bound as the maximum signed value of the given bit width. This is also how some DL frameworks implement ReLU activation [6].

Figure 2.6 shows how ReLU and sigmoid non-linearly scale values close to $x = 0$. The ReLU is always zero when x is less than zero, and x when greater than zero, making its calculation a matter of checking the signed bit for each element. Sigmoid on the other hand has a curved shape, highlighting the fact that it's more expensive

to calculate.

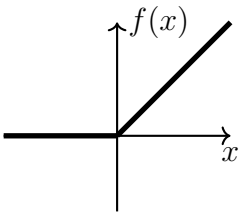

Figure 2.4 *Rectified linear unit (ReLU)*

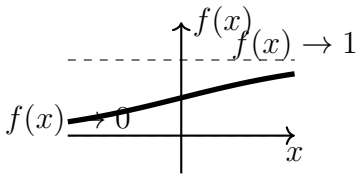

Figure 2.5 *Sigmoid activation*

Figure 2.6 *Comparison of ReLU and Sigmoid activation functions*

Combining two-dimensional affine transformation and ReLU gives as the basic Conv2D layer found in most image classification network, like Resnet [16] and MobileNet [17]. And as with bias, some DL frameworks fuse activation function with the convolution operation to form one layer. Giving use the common definition of a single two-dimensional layer in

$$ReLU(Conv2D(I, K) + \hat{b}), \tag{2.17}$$

where I and K are the respective input and weight tensors of the layer, $\hat{b}$ is the vector containing channelwise bias values and $ReLU$ function for the hidden activation of the layer.

2.3 Layer graphs

As mentioned, DNNs are neural networks with one or more hidden layers. Generally the amount of layers correlates to better prediction results, due to the increasing amount learnable parameters being able to capture more complex features.

The relationships between layers can be presented as graphs, where nodes are layers or fused layers and paths are the data flow directions. Figure 2.7 displays the equation 2.17 as a simple graph were data flow is always from one layers output to next layers input.

Figure 2.7 *Feed forward relationship between Conv2d, bias and ReLU layers.*

For clarity this combination of 2D convolution, bias and ReLU is usually fused into single layer node. Different neural network frameworks use slightly different terminology relating to the meaning of operation, layer and fused layer. For example Tensorflow [25], a popular framework for training, considers Conv2D, bias and ReLU

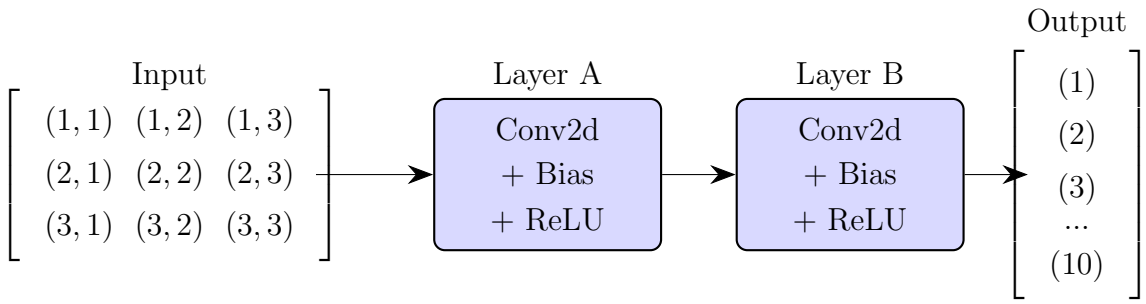

Figure 2.8 *Simple feed forward network with two Conv2D with bias and ReLU activation layers.*

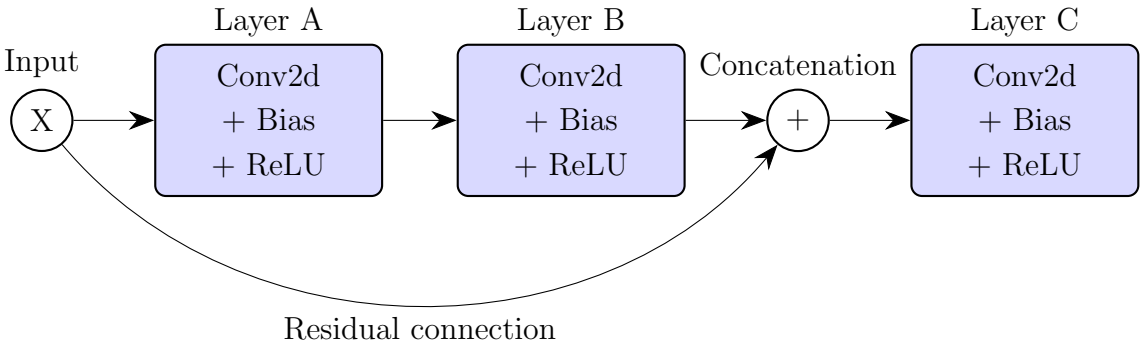

Figure 2.9 *Residual feed forward network with fused layers.*

separate layers and the combination of a fused layer, whereas in Pytorch [28] the combination of Conv2D, bias and ReLU is considered one layer. For the purposes of this work, we use Tensorflow naming scheme.

The simplest kind of relation is a feedforward relationship where the output of layer A is the input of layer B as shown in the figure 2.8. In a simple feedforward connection one instance of a layer output is always exactly input for one layer, without recursion or branching.

Connection can also branch, and the same layer output can be used as input for multiple nodes. Resnet [16] heavily utilizes what are called residual connections. With residual connection an input of a layer is used in multiple parts of a feed forward network. Figure 2.9 shows and example of a residual connection, where input of layer A is used again after concatenation as part of input for layer C.

2.4 Neural Model Training

To perform a given task all neural networks need to be trained. With training the network parameters (weights and biases) are tuned in such a way that the network approximates a function performant in the task. This same concept applies to most model based ML-methods but with neural network this training is most often done with the backpropagation algorithm. The first step of backpropagation

is the forward pass. Forward pass is essentially inference, where the network is given inputs from a known set of inputs with associated ground truth labels for which the network performs predictions. The predictions are then compared with the ground truth with a loss function to calculate current network error. The choice of loss function is critical and poor choice can heavily effect the models potential to discover suitable parameters. Some common loss functions are mean squared error, cross entropy loss and L1 loss.

After forward pass comes the backwards pass, where the parameters of the network are adjusted to minimize the loss function for the given inputs. The tuning is done layer by layer starting from the output layer and working towards the input layer, hence the name backpropagation. To tune the parameters in a way that maximally improves the prediction results, we need to minimize the value of the loss function. To accomplish this we calculate the gradient decent of the network. Essentially gradient descent describe derivative of the n-dimensional space housing the function approximation of the network. Each parameter introduces its own dimension and thus adding a parameter to the derivative.

$$g_K = \nabla L(\hat{y}, z_K) = \frac{\partial L}{\partial a_K} \cdot \frac{\partial a_K}{\partial z_K}, \tag{2.18}$$

where g is the gradient of the last layer's activation in relation to the expected activation as signified by K, L is the loss function, $\hat{y}$ is the ground truth or expected prediction, a_K is the activation of the output layer or the prediction and z_K is the input of the layer. To find the gradient for the next layer we apply the same chain rule, since we know that the activation of the last layer is dependent on the activation of the previous layer

$$g_{K-1} = \nabla L(\hat{y}, z_{k-1}) = g_K \cdot \nabla L(z_K, z_{K-1}) \tag{2.19}$$

$$= \frac{\partial L}{\partial a_K} \cdot \frac{\partial a_K}{\partial z_K} \cdot \frac{\partial z_K}{\partial a_{K-1}} \cdot \frac{\partial a_{K-1}}{\partial z_{K-1}}. \tag{2.20}$$

This same chain rule is applied for the whole graph to produce the complete gradient of the network with final activation in relation to the input x

$$\nabla L(\hat{y}, x). \tag{2.21}$$

Since it's known that the activation of a layer is the affine transformation we know that

$$z_K = a_{K-1} W_K + b_K \Rightarrow \frac{\partial L}{\partial W_K} = \frac{\partial L}{\partial z_K} \cdot a_{K-1} \tag{2.22}$$

where W_K is the vector of weights for layer K. To tune the weights of a particular

layer we need to define an additional term, a learning η which describe the amount of movement towards the minimum value of the loss function. We can then use the following equation to calculate new weights for all the layers

$$W_k = W_K - \eta \frac{\partial L}{\partial W_K}. \tag{2.23}$$

Biases are also updated with a similar equation

$$b_k = b_K - \eta \frac{\partial L}{\partial b_K}. \tag{2.24}$$

Model training is computationally expensive process and in the case of some of the largest models can take months to train. For this reason it's common to have separate hardware for training and for inference. Training is generally done on GPUs or NNAs with some high-level training network. Whereas inference can be done on different kinds of hardware configurations from server farms to microcontrollers, since the inference is less computationally expensive.

Most training frameworks use 32-bit floating point values to represent rational numbers. [27][25]. This allows for the best accuracy in the training. After training the weights and biases can be moved to the target device for inference. Even in the cases where the target model is to be quantized, the network is first trained with high accuracy floating points and quantized afterwards, to represent the rational values with integers to ensure high model accuracy.

2.5 Quantization

When training DNN models with high-level tools like Pytorch, models are built to use floating point values. In recent years big players like NVIDIA have started to utilize more and more quantized integer models ***[Needs citation]***. This is due to the fact that as the amount of parameters in models like GPT, has been growing exponentially ***[Needs citation]***. Often there are significant performance gains available by reducing the granularity of the parameters without a major loss in model accuracy as shown [11, 21]. Standard floating point value has a width of 32-bits, whereas int8 which is the most common integer type in DNNs has just the 8 bits. Thus, when less granularity is acceptable similarly performing integer based accelerator can do 4 times the calculations when compared to a floating point accelerator. Some models reduce that amount of granularity even more and have layers using 4-bit or 2-bit integers. With 2-bit integers one can do 16-times as many calculations in comparison to floating points.

It's also possible to have only parts of the model quantized. For these cases it might be necessary to have additional conversion layers to go from floating point

inputs to integers and backwards. This can be useful for the cases where the target platform is only able to accelerate quantized layers, but the developer want's to use well proven subnetwork to ensure accuracy while the rest of the network is hardware accelerated to improve performance.

There are multiple approaches to performing post training quantization, but it essentially always involves representing particular floating point range of a values in layer inputs or outputs with integers fixed to a certain zero-point. Tensorflow uses the affine quantization to quantize the floating point parameters to integers

$$x_Q = clamp(0, 255, round(\frac{x_{float}}{\Delta}) + z) \tag{2.25}$$

where x_{float} is the original parameter values, x_Q is the quantized parameter value, z is the zero-point and Δ is the scaling factor discovered during quantization. The clamp operation limits the input to a given range as such

$$clamp(a, b, x) = \begin{cases} a, & \text{for } x \leq a \\ x, & \text{for } a < x < b \\ b, & \text{for } x \geq b \end{cases}. \tag{2.26}$$

To convert quantized parameters back to the original parameter values affine quantization uses the following operation

$$x_{float} = (x_Q - z)\Delta. \tag{2.27}$$

This back conversion has no need for the clamping operation or rounding, since the floating point range is larger than the 8-bit integer range.

Another choice for quantization is the uniform symmetric which is similar to the affine quantization, but the zero-point is always set as 0. This ensures that the negative and positive sides are equal, which is desired of signed integer based accelerators. The equation for quantizing a parameter with the uniform symmetric quantization is as follows

$$x_Q = clamp(-128, 127, round(\frac{x_{float}}{\Delta})). \tag{2.28}$$

From the equation we can see that when compared to the affine quantization the zero-point term has been removed and the range has been shifted by -128. Converting back to the original parameter is as simple as applying the scaling factor

$$x_{float} = x_Q\Delta. \tag{2.29}$$

Some integer accelerators might support performing the quatizations in hardware, but generally it's done in software.

2.6 Validation and DNN inference evaluation metrics

Model validation is a practice of evaluating the quality of predictions of a neural network. Validation can be done during model training, as well as after training has finished. The purpose of training validation is to detect overfitting, by testing the so far trained model between training loops with data it has not been trained on. The idea is to simulate inference, so that it performs equally well for unseen data as it does with training data.

Post training validation ensures that the trained model still performs in a new platform or environment. For example after a model has been trained on a high-level framework and moved on to a different run time on a another platform, it should be validated to confirm that the new stack works as expected. Different runtime have different implementations of operation and thus the conversion between models, might need additional work from the developer to ensure compatibility.

To evaluate the accuracy of a neural network we measure the amount of correct predictions relative to incorrect predictions. The suitable metric depends on the used codomain's dimensionality and the problem definition. A metric suitable for 100-class classifier might not suit binary classifier.

A single prediction from a binary classifier can have four possible results. Prediction is true positive (TP) when ground truth is positive and the classifier correctly predicted it to be positive. Prediction is true negative when the ground truth class is negative and the classifier predicts it as negative. False positive (FP) and false negative (FN) predictions happen when the classifier fails to correctly classify the input.

2.6.1 Top-1

Top-1 is the most straightforward evaluation metric for multi-class classifiers. In top-1 we simply take the highest classified class and compare that to ground truth. If the predicted class is the same as the ground truth the predictions is counted as correct. This is repeated for all inputs in the validation set, and the final accuracy is determined by the relation of correct predictions to the total number of predictions done.

$$\text{Top-1 Accuracy} = \frac{1}{N}\sum_{i=1}^{N}\delta(\hat{y}_i, \text{argmax}(f(x_i))) \tag{2.30}$$

where N is the total number of inputs in the evaluation, δ is the Kronecker delta function, $\hat{y}$ is the correct classification of the input, x_i is the input to the classifier

and $f()$ is the classifier.

2.6.2 AUC

Area under the curve (AUC) is a metric for evaluating binary classifiers. It compares the relative amount true positive predictions to false positive predictions. The main benefit of AUC over simple accuracy is that it balances uneven datasets. For binary anomaly detection it's common that the amount of non-anomalous samples is magnitude larger than the amount of anomalous samples. Most ML libraries approximate the AUC with a discrete AUC using the Trapezoidal Rule [27]

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{2.31}$$

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} \tag{2.32}$$

$$\mathrm{AUC} = \int_{\mathrm{FPR}=0}^{\mathrm{FPR}=1} \mathrm{TPR}(\mathrm{FPR})\, d(\mathrm{FPR}) \tag{2.33}$$

$$\approx \sum_{i=1}^{n-1} \frac{(\mathrm{FPR}_{i+1} - \mathrm{FPR}_i) \cdot (\mathrm{TPR}_{i+1} + \mathrm{TPR}_i)}{2}, \tag{2.34}$$

where TPR is the true positive rate, the relative amount of correct positive predictions from all the positive inputs, and FPR is the false positive rate, relative amount of false positive predictions to all negative inputs.

2.7 System-on-chip

The limited growth in single-core performance due to limitations of Dennard scaling [12], has shifted the focus of CPU designers towards homogeneous multicore architectures. and the need overcome Moore's law chipmakers have been moving even futher towards heterogeneous architectures. Heterogeneous architectures consist of conventional von Neumann CPUs and unconventional computing elements working together to perform calculations. Unconventional computing elements, might include GPGPUs, FPGAs and custom logic units. [8]

System-on-chips (SoCs) are a form of heterogeneous computing, where multiple different computing elements and electronic systems are integrated into one circuit. [15] SoCs generally include CPUs, memories, IO interfaces and specialized accelerators, but conceptually there is no set definition on what SoC needs to include. This essentially produces a single functional entity that can easily integrated into multitude of different general processing workloads.

What separates SoCs from other heterogeneous architectures is its level of integration between computing elements. Interconnects that connect the chips computing elements can be run on very high bitrates, greatly reducing latency. Unified memory architectures and DMAs enable different components to use the same data without needing to rely on the main computing units for access, improving the system's coherency. The single chip design of SoCs give them a noticeable advantage in terms of physical size and energy consumption, when compared to traditional heterogeneous architectures. This has made SoC a popular option for mobile and edge devices, where the physical size and the decreased energy consumption has significantly been able to improve performance.

In recent times SoC have also begun being seen more in consumer laptops, effectively handling desktop workloads as is the case with the Apple M-series of chips as well as the Qualcomm Snapdragon Elite X chip, Intel Lunar Lake.

[10]

2.8 Deep Learning Accelerators

Deep Learning Accelerators (DLAs) or sometimes called Neural Processing Units (NPUs) are hardware accelerators, that accelerate common neural network operations. In many networks, this means accelerating convolution and dense layers by parallellizing the calculation of output elements. Due to the size of DNNs an accelerator cannot usually fit the whole model in to its memory. This necessitates a need for off-chip memory to store the model, which the accelerator accesses for new data between layers. The consecutive nature of most DNN models also makes it impossible to have a pipelined execution of layers, next layer computation generally cannot begin before the previous is finished. For this reason DLAs usually feature just one central computing element.

In desktop applications and data center workloads neural networks have been accelerated with GPUs, due to their ability to perform linear-algebra operations like matrix multiplication with high amount of parallellity. To improve power efficiency, it's becoming more common for mobile devices to have dedicated DLAs instead of GPUs. Companies such as Apple and Qualcomm now include multiple mobile DLA's in their SoCs to run applications like face recognition on phones using their chips.

In addition to SocHub's Headsail-DLA other organization have developed their own ASIC based accelerator's for DL workloads. DianNao is one of the first ASIC based accelerators targeting convolutional deep neural networks [7]. It's based on a pipelined NFU, where the multiply and accumulation operations are implemented as a pipeline alongside bias and sigmoid activations. This is unlike more modern designs where the main computational element is a computational array.

Eyeriss [4] is an early example of a MAC array based DLA for accelerating convolution networks, focused on high energy efficiency and performance, by minimizing the amount of data transfers. Eyeriss implements convolution 2d operation with bias and ReLU, and it uses 16-bit floating points. Convolutions are executed in a 12 x 14 PE-array, where each PE performs the multiply-accumulate operation for one input feature element at a time. Eyeriss loads current layers parameters from the off-chip dram into it's 108KB global buffer, from which the operation parameter are scattered for the PE-array. PEs heavily utilize scratch pads to improve data access times, by enabling data reuse of the kernels.

3 Methodology

This section goes over the technologies used to complete this project in detail. First we discuss the hardware used in the SocHub's Headsail SoC and the parts affecting the decision made in the software design in detail. After this we present the used software stack necessary to run neural networks on the described hardware. We will also discuss the virtual prototype on which the majority of the software development took place on, and how it differs from the hardware.

3.1 Headsail

Headsail is the third Soc build by the SocHub research group [29]. Headsail has two RISCV CPUs, one 32-bit meant for booting up the system called SysCtrl and one 64-bit 4-core CPU called HPC, meant for running the actual applications, based on the CVA6 [41]. Headsail includes a wide variety of different peripherals, one of which is a custom built the Deep Learning accelerator. For I/O headsail has UART, SPI and I2C connectivity. In addition to CPU bootrams, Headsail features 256 megabytes of SDRAM and 128 kilobytes of shared SRAM. This abundance of memory gives application space developers lots of flexibility for their applications. Considering that even small neural networks need megabytes of memory, large memory will be vital for the success of this project.

3.1.1 DLA

Headsail's DLA is a MAC array based accelerator, which provides the following operations: Conv2D, Bias, ReLU. The operations are implemented as a pipeline, meaning that the order of operations is always the same. During one layer cycle the operations need to be executed in the following order: Conv2d, Bias, ReLU. This is the most commonly found order in modern neural networks so it suits most use cases. In addition to these operations DLA can perform bit shifting for results of the MAC array and the post-processing pipeline. Bias and ReLU can be skipped in the case neither of them is needed in the given layer. In this case Conv2D output is used directly and is capped to fit the 8-bit width of the output. The particular operations are configured from the register interface of the DLA. DLA has a simple 32 bit RISC-V based controller CPU, that can be used to drive the DLA parallel to normal HPC execution, but it's also possible to control the DLA directly from the main CPUs. Which is what we have opted to for this project.

The input and weights are feed to the MAC array from the 16, 32 kilobytes sized memory banks. The input and weights need to be written into separate banks and

the data needs to be continuous and in the case the data doesn't fit into a single bank the neighboring bank needs to be used. The bias values for the post-processing step are fetched from the Headsail's SDRAM and can be located in any arbitrary point of memory.

3.2 TVM

TVM is a machine learning compiler framework by Apache. Among other features TVM includes, multiple runtimes, accelerator backends, optimizers, and a machine learning library for building and training models. The variety of features allows for TVM to be used to implement a complete machine learning workflow, or TVM can be used to implement part of the workflow with other tools.

TVM implements it's own graph representation for neural network with it's Relay IR graphs, which similarly to other graph representation represents network layers as nodes in a abstract syntax tree. In the graph data flows is signified by the directional vertices connecting the nodes, in such a way that the output of the node the data travels from acts as the input for the child node at the end of the vertex. The Relay IR support multitude of different transformations to manipulate the operations performed in the network. This enable developers to write transformation passes to modify certain nodes of patterns of nodes to ensure that the graph is compatible with different hardware configurations.

TVM can be extended to support additional hardware accelerators by implementing a custom code generation module for the target hardware. In principle the developer defines external C symbols that provide the operation implementations which TVM then injects into the Relay IR models. During runtime TVM then calls these external symbols instead of the default operations provided by the TVM Relay library.

It's possible to generate Relay IR models from other graph formats with TVM. For example common formats like Tensorflow, Torch and ONNX [9] models are officially supported by TVM. This allows developers to build and train their models with tools they might prefer over TVM, and use TVM as a compiler/runtime.

TVM has the ability to take most of the common ML training graphs and convert them to TVM Relay graph. Figure 3.1 shows the conversion of a quantized Tensorflow Conv2d layer into corresponding relay graph. From the figure we can see that TVM separates nodes into smaller entities, where each node performs one options, instead of the fused approach of Tensorflow. This gives developers more control over which operations to assign for which hardware. During model compilation TVM is able to optimize the graph and allocate acceleratable nodes to suitable accelerators. [6]

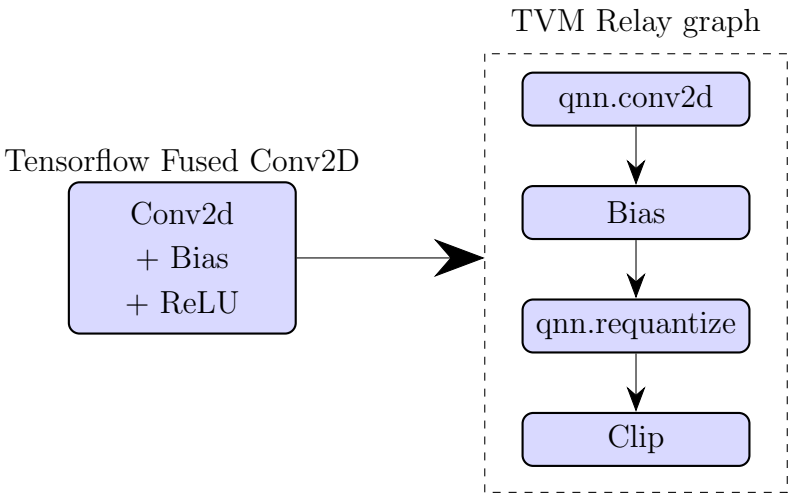

Figure 3.1 *Tensorflow layer to TVM relay graph conversion*

3.2.1 Runtimes

The function of a neural network runtime is to enable other parts of the program to make predictions using the neural network. To do this runtime needs to know when to apply which operation and with which parameters. TVM offers two different runtimes. First is the graph executor runtime. The graph executor takes the graph representing the neural network and traverses it in order to know which operations to execute. The graph executor also needs a separate data structure for the neural network parameters, which the graph has mappings for. During execution the executor fetches the necessary parameters for each operation based on the reference in a given node matching to the parameters.

The other option for a runtime is the Ahead-of-Time runtime (AOT) which takes the same graph and parameters as with graph executor, but instead of dynamically fetching the operations and parameters during execution, the AOT runtime compiles the graph into executable C code or machine code. The AOT then produces a simple API with entry points for input data and execution call, for running predictions in a program. When the program calls for prediction the graph traversing is done by simple calling a next node in the graph as a function, where the execution of a specific operation is defined programmatically, and the necessary parameters are set in place.

The main difference between the runtimes is that graph executor is more dynamic. The executed network and parameters can be redefined during runtime. The AOT is more rigid. All the possible networks need to be embedded directly into the binary of the program. This rigidity comes with simplicity. The API of the AOT is really simple to use, consisting of only the data input and run call. Graph executor, requires more setup from the program, such as parsing the JSON to obtain

the graph, and dynamically loading the parameters for each node. For the purposes of this project we settled on using the AOT.

3.2.2 Graph Transformations

To assign calculations for an accelerator the Relay graph of a network needs to be transformed, in such a way that the resulting graph is compatible with the device. In addition to conversion from a different framework to relay, the relay graph needs to be legalized for a specific target and acceleratable patterns needs to be assigned for suitable hardware.

In TVM the first transformation pass is the legalization, where the graph is traversed, and certain parameters are recalculated to fit the target. For example when executing a unsigned 8-bit quantized network most models use a zero-point of -128 to mimic signed behaviour. If this network is run on an accelerator with support for signed integers, this zero point needs to be changed to 0, since there is no need for the adjustment. If this network is run on an accelerator with support for signed integers, this zero point needs to be changed to 0, since there is no need for the adjustment. This can be done in the legalization pass.

The other transformation pass is the graph annotation. By default, all the operation are assigned to the CPU. With graph annotation certain patterns of nodes can be annotated as to be executed with additional accelerators. The patterns are similar to regexes, where a sequence of nodes, for example Conv2D followed by a bias node can be fused together into single composite node.

During code generation with the AOT runtime, TVM traverses the graph and generates code to execute each node. If the node or composite node is annotated to be executed for an accelerator TVM refers to the corresponding code generation backend to produce code for executing the operation. This exposes the way for developers for integrating new accelerators for TVM.

3.2.3 BYOC

"Bring your own codegen" or BYOC, is one of the possible APIs for integrating new devices into TVM. With BYOC, developers define target specific relay backend to generate C code for executing the annotated patterns in the model graph. The backend is created by defining `CallNode` operators that are called when the backend traverses a network graph. These operators extract the needed information of the nodes for use with the target's deep learning API. Target backend is defined as a C++ module in TVM relay backend and can output either C code or object code when supplied with TVM's LLVM integration.

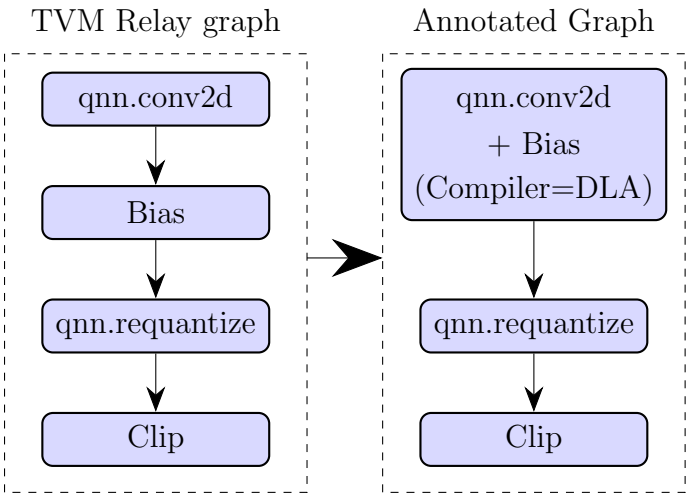

Figure 3.2 *Graph transformation for DLA-VP*

3.2.4 TVM on baremetal

TVM also provides a tool to run TVM models on bare-metal platforms called microTVM. MicroTVM is only dependent on the C standard library and thus can be used in any bare-metal system that has a working C-toolchain.

MicroTVM works by generating platform independent C-source code from Relay IR-models, which can then be integrated with the microTVM AOT runtime to produce executable binaries to run the network.

The DNN operation implementation are provided by TVM contrib library. With custom code generation it's also possible to define bare-metal compatible accelerator nodes, which the TVM runtime is able to assign layers for during the C source code generation. [6]

3.2.5 TVM quantization

TVM implements quantization with its own QNN-dialect [19]. The QNN-dialect separates quantized and non-quantized operations from each other by categorizing the quantized operations under the `qnn.op` class. QNN operations have additional arguments to indicate scaling factor and zero-point for inputs and outputs that are not present in floating point operations. In addition to having quantized version of the DL operations the QNN dialect introduces operators for moving data across from non-quantized domain to quantized and back. This is done with the `qnn.quantize`, `qnn.dequantize` and `qnn.requantize` operators. `qnn.quantize` performs affine quantization to a floating point tensor to produce a corresponding quantized integer tensor, where as `qnn.dequantize` reverse the affine quantization. `qnn.requantize` converts a quantized tensor to another quantized tensor corresponding with a different scale than the original.

TVM enables compatibility of quantized networks with other major DNN frameworks like tensorflow with QNN-dialect aware graph parsing. When model from another framework is parsed into Relay graph, the framework specific quantization operations get converted to QNN nodes. The resulting graph is then passed through canonicalization and legalization passes to produce relay graph that can be annotated for a target. Canonicalization pass converts the QNN nodes into relay operations. For example `qnn.Conv2d` gets broken into, dequantization, floating point convolution, bias, requantization to interger values and clipping operations. In legalization pass the canonicalized relay graph is transformed into relay graph that is compatible with the target hardware. For example this might include rewriting zero-points in requantization nodes, or recasting inputs to fit the target device. After legalization the graph can be annotated and acceleratable patterns can be assigned for the targets.

3.3 Renode

Renode is a system emulator, which enable developers to rapidly develop hardware dependant software. In essence Renode allows the user to specify exactly which kind of hardware they want to target, down to the implementation of specific peripherals and memory addresses. This streamlines the process of HW/SW integration, since hardware and software can be developed in parallel, which in return reduces the total production time for products.

Renode models a wide variety of different processors and peripherals, but it is also expandable with custom components that are either baked directly into the binary (source code extensions in C#) or with dynamically loaded Python peripherals. Python peripherals are more limited when compared to the C# peripherals, since they can't be connected to the clock domain of the system and can only change state when their memory region is accessed. Renode's Python peripherals are also executed in a separate IronPython container that this moment, only support modules from Python standard library.

While Renode is an operation accurate emulator, the Python API isn't. When Renode makes a request to the Python API, it counts as one clock cycle even when realistically the Python API's corresponding hardware implementation would take more than one cycle. The consequence of this that we cannot accurately benchmark DLA in Renode. The benefit of the Python API is in rapid development of hardware components.

Renode integrates with the Robot testing framework [38] to enable automated testing of Renode devices. This makes it possible to use principles of continuous integration and continuous delivery for Renode projects and hardware dependent software.

3.4 MLPerf Tiny

MLPerf Tiny is a benchmarking suite for benchmarking ML inference in low power targets, like MCUs with Deep Learning Accelerators. [1] MLPerf Tiny consists of four benchmarks meant to target different use cases shown in table 3.1.

Table 3.1 *Tiny Performance Benchmarks, from [1]*

Benchmark	Dataset (Input Size)	Model (TFLite Model Size)	Quality Target (Metric)
Keyword Spotting	Speech Commands (49x10)	DS-CNN (52.5 KB)	90% (Top-1)
Visual Wake Words	VWW Dataset (96x96)	MobileNetV1 (325 KB)	80% (Top-1)
Image Classification	CIFAR10 (32x32)	ResNet (96 KB)	85% (Top-1)
Anomaly Detection	ToyADMOS (5x128)	FC-AutoEncoder (270 KB)	.85 (AUC)

Each of the four tasks uses a different model, dataset and problem definition, to ensure testing of wide amount of workloads. Each of the tasks has a minimum quality requirement to be considered acceptable for the benchmark. For the three multi-class classifiers the used metric is top-1 and for anomaly detection the metric is AUC.

Since MLPerf Tiny aims to suit most kinds of targets, they give the users lot of freedom in the implementation of the models. In the benchmarks the only performance metric is the actual inference times, as long as the quality requirement is fulfilled. This allows for devices with limited IO capabilities a fair comparison against more capable devices. There is no limitation on the model's data type so MLPerf Tiny can be used to compare between signed and unsigned integer accelerators, as well as with FP8 and FP32 accelerators. In this project we used the pretrained reference models offered by MLPerf for the benchmark, but the specification also allows for training of new models as long as the architecture is preserved.

3.4.1 Image Classification

The image classification task aims to classify images from the CIFAR-10 dataset [22]. CIFAR-10 popular dataset that includes 32x32 RGB color images, each belonging to one class. The 10 classes of CIFAR-10 are labeled as, airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck.

To do this classification MLPerf Tiny uses Resnet [16]. Resnet is an implementation of ImageNet ***[Cite this]***, where between every pair of layers there is a residual connection added from the pair preceding the previous pair. The backbone architecture introduced in ImageNet uses multiple consecutive layers of 3x3 kernel size 2D convolutions Backbone of the network is built from multiple consecutive 3x3 convolution layers which are in addition to simple feed forward connections. Resnet architectures vary by the depth of the network, i.e. number of layers. For example Resnet-50 has 50 layers and Resnet-34 has 34. The image classification task uses

custom ResNet-9, which lacks the downsampling pooling layers to compensate for the low resolution of CIFAR-10 dataset.

3.4.2 Visual Wake Words

In the Visual Wake Words (VWW) task, the goal is to identify if at least one person is present in an image from the MSCOCO 2014 dataset [23]. The training dataset consists of RGB images preprocessed to the size of 96x96 pixels, featuring at least one person. The testing set, which the benchmark uses features images of the same size, but some have people and some do not.

The model chosen for the VWW is MobileNetV1 [17] Backbone of MobileNet consists of 13 depthwise-separable convolution layers, which are followed by a classifier consisting of average pooling layer and a FC-layer. This architecture makes it the largest model in the benchmark both in terms of parameters and the number of layers.

3.4.3 Keyword Spotting

The keyword spotting task is an audio processing task where the goal is to identify spoken keyword from multiple sound sources. Data used dataset is the Speech Commands V2 dataset [40], consists of short clips where one of 30 words possible words are pronounced. From these words ten are used as keywords alongside background noise and rest of the words as unknown, to produce 12 labels. The samples are converted to spectograms, allowing us to use a proven image classification network for the classification.

MLPerf Tiny uses model called DS-CNN for the KWS task. DS-CNN is very similar to MobileNet, in that it heavily utilized depthwise convolutional layers for the backbone. [34] Where MobileNet used 13-layers DS-CNN only has four depthwise-separable convolution layers. Which makes it the smallest model in the benchmark, both by the number of layers and parameters. Similarly to MobileNet, DS-CNN has a average pooling layer followed by a FC-layer for classifier.

3.4.4 Anomaly Detection

The dataset for this task is the DCASE2020 dataset [20]. DCASE2020 consists of sound samples from the following labels: slide rails, fans, pumps, valves, toy-cars, toy-conveyors, from which the AD task only detects anomalies from the toy-cars.

For the model AD task uses a custom FC-AutoEncoder [2]. Originally proposed by [33], AutoEncoder consists of three parts. First is the encoder which encoder the input and downsamples it to latent space. Second part is the latent space which holds the useful high-level features of the input. Last is the decoder which decodes

the latent features and upsamples these features back to original dimensionality of the input. [5] The used FC-AutoEncoder model consists of 10 FC-layers with ReLU activations, from which the first 4 make up the encoder, fifth forms the latent space, the four after latent space make up the decoder and last layer performs the classification.

4 Implementation

This section covers the actual implementation of the used software stack in detail and the specific use cases developed on top of the stack. As well as the MLPerf Tiny benchmark used to evaluate the performance of the DLA.

4.1 Headsail-VP

To enable software development for Headsail before the arrival of the ASICs, we modeled the hardware as a Renode virtual platform, which we call Headsail-VP. With Headsail-VP we aimed to replicate the complete memory map of Headsail, with both processors and some of the peripherals. The aim with the virtual platform was to kick-start software development for the ASIC and to give developers a more streamlined development environment. We hoped that Headsail-VP would enable faster software development, by removing some friction associated to working on the ASIC, as well as giving use the possibility of modifying the design in the case of hardware bugs. Additionally, this knowledge in developing virtual prototypes of chip could allow us to better define the hardware specs based on the software demonstrators we develop in for the future chips. Renode's integration with RobotFramework allowed us to setup a CI/CD pipeline with Github actions from the beginning, ensuring the functioanlity of new features.

4.1.1 DLA-VP

Since wanted to develop a use-case for the Headsail DLA with the virtual platform, but because the DLA is a custom ASIC design we had to model it in Renode ourselves. From the two possible options of adding new peripherals to renode we chose to use the Python periheral API. This decision was due to the developer's confidence in writing Python rather than C#, and the intrest in seeing if a peripheral

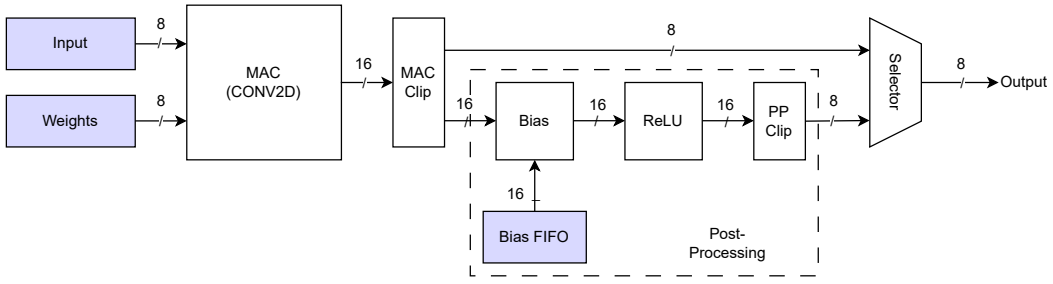

Figure 4.1 *Architecture of accelerated DLA flow in Headsail with TVM runtime and a Pytorch model*

of this size could be succesfully implemetend with the more limited API.

The DLA-VP supports all the same operations as the ASIC implementation, with identical register interface and same data buffer architecture. This allowed us to develop the driver for the DLA completely on the VP first. After the ASICs where ready we could confirm that the DLA driver was indeed usable on both VP and ASIC. Figure 4.1 shows the high-level structure of the DLA, featuring the input buffer as colored block and the functional blocks in white. This high-level processing pipeline is followed with both the ASIC implementation and in DLA-VP.

DLA-VP implements the register interface as a list of same length. Since numbers in Python can be infinite in length we had to carefully sanitize all the transactions to the registers, so that no accesses over 8-bit values were possible. This same sanitation logic also applied to writing results to the output buffer, since on ASIC all output buffer entries are signed 8-bit values.

Since Renode Python peripherals don't have a clock, the state of the peripheral can only be changed when a CPU reads or writes to an address that is registered for the peripheral. DLA-VP is designed to run a processing loop after each write to its memory region, that checks if the state of DLA-VP is ready for operation execution and if yes performs the operation per configuration. Read accesses don't change the devices internal state, so the processing loop isn't executed on them. After the executing write call, the result of the operation can be read on the next clock cycle from the peripheral's memory region.

To verify that the DLA-VP functionally behaved like the hardware implementation of the DLA, we implemented some of the same verification tests done by the hardware team with our software stack. Using the same inputs as the RTL tests with DLA-VP we expect the results to be identical. We implemented the same tests for the low-level driver as well as the high-level driver to ensure that the DLA-VP, as well as the complementing software stack, matches the behaviour of the hardware implementation. This included reimplementing the bugs and design oversights that reduce the accuracy and possibly performance. We integrated these same verification tests to our CI/CD to ensure changes that would break the functional correctness of the DLA would not get merged.

4.1.2 Result width limitation

A major limitation of the DLA's design is its limited bit width of the output elements. A single element in the output tensor is represented with a signed byte and thus has the value in the range from -128 to 127. This can be seen from the figure 4.1 where the bit-width of data is reduced in half after the MAC clip and PP clip blocks. This inevitably has the consequence that we will lose some accuracy of results when we move data back to the CPU. The amount of inaccuracy depends on

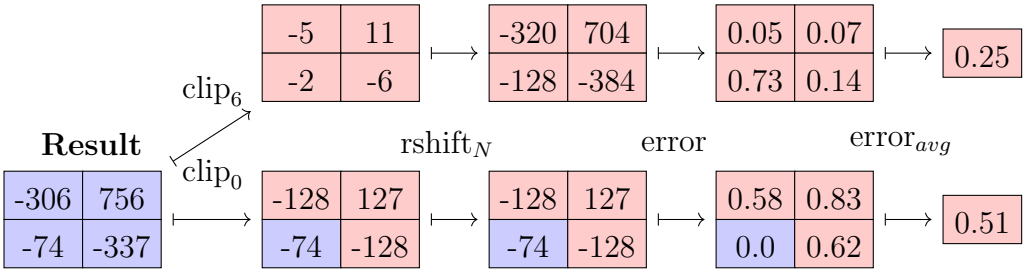

Figure 4.2 *Example of the result reading procedure from DLA with two different clipping values, and their effect on the retrieved values.*

the amount of clipping we perform in the clipping step. DLA has the ability to clip any sequence of 8-bits from the 16-bit sequence that is each result tensor element.

Figure 4.2 shows and example of the clipping behaviour for two different clipping value for two different clipping values. In the figure the Result matrix is some arbitrary convolution result from the MAC array, which gets applied clipping values of 0 which means that we don't clip the result, giving us the 8 least significant bits as the read result. The second clipping amount is 6. In the driver we perform right shifting to reverse the effect of the clipping giving as results in the correct magnitude. By taking the difference between the original values and the right shifted values we can get the element-wise and average amounts of error in for both clipping values.

In figure 4.3 we have repeated this procedure for all nine possible clipping values and plotted the results. From these results we can see that for this particular result the best accuracy is retrieved with clipping amount of 3. For different matrices the optimal clipping value will be something different. Choosing this result poorly can cause complete error of the results as shown with the result of clipping value of 8. This behaviour is the most significant cause on incompatibility between the DLA and DNNs trained on high-level frameworks, such a Tensorflow lite, and mitigating its effect were a major challenge for this project.

A secondary problem cause by the limited bit-width of the DLA relates the bias. Some DNN model rely on bias values close to the maximum value of a signed 32-bit value. This is a problem for the DLA since, it only support bias values up to 16-bits. This causes some results channels to lose their relative values to equalized due to bias values getting saturated, when the bias values are larger than the maximum 16-bit values.

4.2 Board Support Package

We started the actual software development by writing a simple BSP for Headsail. Doing this simultaneously with the refining virtual prototype helped us better understand the complete structure of the SoC.

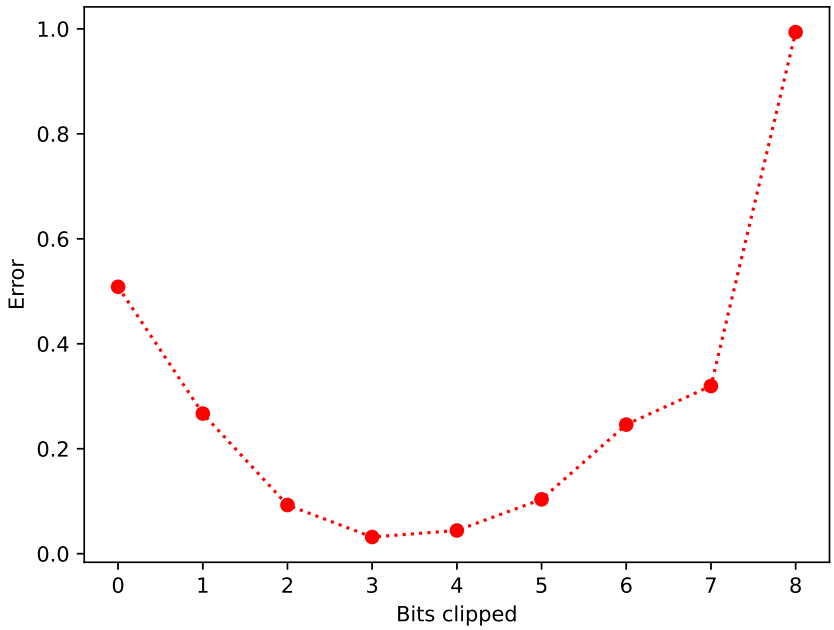

Figure 4.3 *Simulation of the effect of different result clippings to the layer.*

The BSP is written in `no_std` Rust, since it is the default language of choice for SoCHub software projects. Rust has a healthy embedded software ecosystem, with libraries supporting `no_std` environments, like our target hardware. Most importantly the riscv-rt [39], whichh is a library that provides startup code and interrupt functionality to bare-metal RISC-V devices.

The Headsail-BSP implements some basic drivers to interact with the hardware's I/O functionality with peripheral UART and CPUs internal timers. BSP also enables interrupts for both CPUs with the CLINT and PLIC peripherals. We also implement basic test bench definitions to enable consistent automated testing setup for CI/CD.

Rust features allow us to host both HPC and Sysctrl specific code in one code base with minimal effort. This allows us to use same peripheral drivers with CPU specific memory maps, reducing the amount of duplicated code. Using Rust feature flags we can choose between HPC or Sysctrl builds, to target a specific CPU. Using this same principle, we can isolate VP and ASIC specific behind their own feature flags.

4.2.1 HPC initialization

Before we can execute binaries on HPC we need to turn it on and initialize it from Sysctrl. For this purpose we run a small `HPC-init` binary on startup. The initialization program enables the interconnect between the CPUs and turns on the cores of

HPC. In addition to turning on HPC it enables and configures most of the peripherals, such as the SDRAM, DLA and clocks. We provide the initialization program as a part of Headsail-BSP, and have plans to integrate it into a SPI based SD-card boot flow to enable easier workflow with the SoC. On Headsail-VP `HPC-init` isn't run at all since all the peripherals are configured automatically during startup, making this procedure specific to the ASIC.

4.2.2 Dynamic Memory Allocation

Embedded applications often rely on just statically allocated memory, this makes memory management simple and generally perfomant. The nature of neural workloads is such that we need the ability to dynamically allocate and deallocate memory. Specifically, we want to be able to dynamically allocate input and output buffer for layers. Since layers have different sized input and outputs with large amount of elements, it isn't feasible to statically allocate them all.

Since we support two different programming languages, we had to have solution for both. For C this was accomplished by porting Newlib, which we discuss in detail in section 4.4. For Rust we experimented with different possibilities but settled on using an external library called Good Memory Allocator [24], which uses bins holding different sized chunks that can dynamically be allocated. Since Good Memory Allocated only depends on the Rust core library, not requiring the full standard library and has been well tested, it suited our purposes well.

4.2.3 Memory Map and RISC-V code models

The CPUs in Headsail, Sysctrl and HPC work, on a semi-unified memory map trough two interconnects. One 32-bit interconnect and one 64-bit interconnect, which allows both of the CPUs to reach external memory regions. The semi-unified comes from the fact that even though the CPUs can both access the same memory regions with same addresses, HPC accesses need to be appended with an extra bit to signify external memory access. This extra bit has the unfortunate effect that it makes every external memory access 64-bits. This mandates that the C-compiler needs to support `medany` code model [35], which not all instances of riscv-gcc support.

The code model dictates which memory addresses the binary can be mapped to. On RISC-V the standard code model if `medlow` which enable the code to be linked in a 2 gigabyte long region starting from address $0x0$. Since all external accesses from HPC start with 33rd bit set as one, it would impossible to link any HPC program to memory outside of it's bootram. Thus, we need to use `medany` code model which enables using any 2 gigabyte continuous region of memory.

The disadvantage that comes from using `medany` code model is that it prevents

Program 4.1 *Example call to DLA high-level API*

```
pub struct LayerConfig {
    pub input_bank: Option<MemoryBank>,
    pub kernel_bank: Option<MemoryBank>,
    pub output_bank: Option<MemoryBank>,
    pub bias_addr: Option<u32>,
    pub pp_enabled: bool,
    pub relu_enabled: bool,
    pub bias_enabled: bool,
    pub input_size: Option<InputSize>,
    pub kernel_size: Option<KernelSize>,
    pub padding: Option<Padding>,
    pub stride: Option<Stride>,
    pub mac_clip: Option<u32>,
    pub pp_clip: Option<u32>,
    pub simd_mode: Option<SimdBitMode>,
}
```

certain linker relaxation to be performed, which in turn negatively affects the performance of the system. In our testing we observed up to a 5 percentage drop in performance when switching from binary linked with `medlow` to `medany` when targeting HPC's bootram.

4.3 DLA Driver

To make interfacing with the DLA possible we wrote a driver for the accelerator as a part of the Headsail-BSP, which we simply call the DLA driver. The driver is implemented as separate Rust module which can be included with the rest of the BSP using optional feature flags, when invoking the build. This is done so that we can reduce the size of the BSP binary when application don't need the DLA. The driver for the DLA is divided into two parts. First is the lower level API that handles the register level interfacing with the DLA. Second is the high-level API which implements a user facing interface for the main operations of the DLA.

4.3.1 Layer struct

The layer configuration struct is the central component of the DLA driver. It defines all the necessary information needed to run a single layer on the DLA and is passed for the `init_layer` method in the low level API to run the specified layer. The DLA doesn't wait for a particular run command, rather after setting `READ_A_VALID` and `READ_B_VALID` bits in the `BUF_CTRL` register to signal all input and kernel data has been set, the DLA starts immediately executing the current configuration.

Program 4.1 shows the design of the struct. The first three fields control the data locations with the DLA memory banks. It should be noted that the driver nor the

DLA don't enforce any kind of overlapping protection for the data, and it's left at the responsibility of the user to make sure these areas don't overlap. To make this easier the driver offers a method to calculate suitable memory bank assignments. Bias is handled differently from the other data locations since DLA view it as FIFO where every channel takes the next element.

The `pp_enabled`, `relu_enabled`, `bias_enabled` fields control the use of the post-processing unit. The first enables the unit, and the following ones choose which of the two post-processing operations are executed for the layer. For example when calculating basic Resnet layer with both operations all the three fields need to be enabled.

The `input_size`, `kernel_size`, `padding` and `stride` fields control the dimensions of the 2D convolution. `input_size` and `kernel_size`, define the height and width of the inputs and weights as well as the channels counts. Number of input channels for input and number of output channels for the kernel. `padding` fields allows for settings the amount of padding for all directions separately as well as the values used as the padding. `stride` tells the amount of space between samples in the input in vertical and horizontal direction separately. The default stride of 1 means that every input element is sampled, whereas stride of 2 means that after a sample, we move two spaces to the particular direction etc.

4.3.2 Reading Layer Results

To get output from the DLA we first need to define it's output address. This address can be any memory address visible to the control processors, but in the driver we have limited the number of possible output addresses only to the DLA's internal memory banks. For setting the output bank for the given layer the driver writes to `DLA_PP_AXI_WRITE_ADDRESS` register the wanted output address.

To read the output from the DLA we need to know how many elements the output has for any given operation. This information can be derived from the following equation

$$W_{\text{out}} = \frac{W_{\text{in}} + P_{\text{left}} + P_{\text{right}} - K_x + 1}{S_x} \tag{4.1}$$

$$H_{\text{out}} = \frac{H_{\text{in}} + P_{\text{top}} + P_{\text{bottom}} - K_y + 1}{S_y} \tag{4.2}$$

$$\text{No. output elements} = W_{\text{out}} \cdot H_{\text{out}} \cdot Channels_{out}, \tag{4.3}$$

where W is width, H is height, P is padding to either left, right, bottom or top, K is the shape of the kernel and S is the stride in horizontal or vertical direction. Multiplying the shape of a single output channel with the total number of output

channels gives use the total amount of output elements. Since DLA can only output 8-bit values we know that the number of bites to read per layer is equal to the amount of output elements.

As explained in section 4.1.2 the output of the DLA is limited to the signed 8 bit range (from -128 to 127) but the convolution is calculated in signed 16-bit range (from -32768 to 32767) the DLA needs to clip half of the bits away. Using the `mac_clip` and `pp_clip` fields we can control which of the consecutive 8-bits we want to use as the result. By default, DLA stores the 8 most significant bits, but by increasing the clipping values we can move the extraction window towards the less significant bits. By moving towards LSB we essentially gain granularity between the values at the cost of losing range in the high values. This is desired when the absolute maximum value of a layer is small. In most cases the optimal extraction window is somewhere between MSB and LSB, thus making it difficult to predict which exact clipping value to use. When the results of a layer are read from the DLA the driver shifts the read values back by the same amount as they were clipped in the accelerator to match the magnitude of the calculations.

To find an optimal clipping amount for each layer, we proposed a bias maximum value based heuristic

$$P = \max\left(\log_2\left(\frac{\max\left(|\max(\hat{b})|,\ |\min(\hat{b})|\right)}{127}\right),\ 127\right) + 1 \tag{4.4}$$

$$\text{Bits clipped} = \begin{cases} P, & \text{for } P < 8 \\ 8, & \text{otherwise} \end{cases} \tag{4.5}$$

where $\hat{b}$ is the vector containing all the bias values associated with a given layer. From this vector we find the value with the greatest absolute value, and choose the number of bits needed to represent it as the amount of clipping to use incremented by one. In out testing this heuristic somewhat improved the predictions results when compared to choosing a static amount of clipping. We limit the number of clipped bits to high of 8 since, the bit width of the convolution result is 16-bits.

4.3.3 C-Interface

To enable TVM BYOC to call the DLA drivers high-level interface we needed to create C-wrappers for the function calls. This was done by implementing a Foreign Function Interface (FFI) for the Rust function calls with the Cbindgen library [31]. Cbindgen generates C headers from the Rust code that enables C programs to call Rust functions when linked with the proper static library. For the static library we build the headsail-bsp that implements the high-level operation calls which the FFI

Table 4.1 *Newlib Syscalls and Implementation Status*

Syscall	Description	Implemented (Bool)
`exit`	Terminates the process	Yes
`close`	Closes a file	No
`fstat`	Gets file status	Yes
`getpid`	Gets the process ID	No
`isatty`	Tests if a file descriptor is a terminal	Yes
`kill`	Removes a process	No
`link`	Creates a hard link to a file	No
`lseek`	Re-positions read/write file offset	No
`open`	Opens a file	No
`read`	Reads from a file	Yes
`sbrk`	Moves end of heap pointer	Yes
`stat`	Retrieves file status	No
`times`	Returns process times	No
`unlink`	Deletes a name or a file	No
`wait`	Waits for process to change state	No
`write`	Writes to a file	Yes

interface uses.

In the FFI we also define a entry point specifically for the TVM. As previously mentioned we always allocate qnn.conv2d + add pattern for the DLA to execute. The code generation backend extract all the necessary information from the Relay nodes of this pattern and generates a function call to call `dla_tvm_qnn_conv2d_bias` function from the FFI. In addition to just calling the wanted Conv2D + bias operation from the BSP, the FFI function slices the data buffers to Rust slices and performs certain value conversions, like clipping the 32-bit bias values to 16-bit values. After the operation has been done and the result rest the FFI implementations shifts the values left by the same amount as was clipped in the DLA. These values are then copied to the buffer defined by the TVM codegen.

4.4 Porting Newlib

We wanted to use TVM's AOT runtime to execute inference in the benchmark we had to provide it with a C standard library implementation. For this purpose we chose to port Newlib for Headsail, since it's known to be relatively easy to adapt for new platforms.

Newlib is an implementation of the C standard library meant for use in embedded devices [30]. It is separated into two different parts. First is the Newlib core which implements the actual standard library for different CPU ISA's. Since Newlib already has support for RISC-V we didn't need to modify Newlib core in any way.

Program 4.2 *Minimal implementation of the fork() syscall in Newlib Libgloss*

```
int _fork() {
    return -1;
}
```

Second part of Newlib is called Libgloss, which implements the platform dependent features. For this port we decided to only implement the minimum required functionalities of Libgloss.

Porting Libgloss involves implementing 16 system calls, crt0 and a linker script. From the 16 system calls only some are mandatory for us to implement, because we are targeting only bare-metal applications without a filesystem. The table 4.1 shows all the Libgloss system calls with column 3 displaying if we implemented the call. The unimplemented calls still need to be defined in the Libgloss source for linking purposes, but they don't need to do anything except return an error. Program 4.2, shows and example of a minimal implementation for the fork system call in Headsail's Newlib port. The purpose of the Fork system calls is to duplicate a process, but since we don't support multithreading or any other form of process concurrency it will never be called, thus having it return error is correct behaviour. We use this same approach in the rest of the system call that we didn't implement.

The system calls can be implemented either in non-reentrant or reentrant way. Re-entrant system calls are thread safe but require an additional argument, re-entrancy structure, to be passed. Re-entrancy structure holds local values specific for that instance of the function call, whereas the non-reentrant version of the function refers to shared global variables. For single threaded applications on headsail implementing the non-reentrant systems calls was enough. [3]

In addition to the system calls Libgloss needs to have a crt0, which is a small program that includes a `_start` symbol, a call to the `main` function, some global definitions, and optionally hardware initializations. The `_start` symbol signifies the beginning of a C program and is used by the linker to place the program to start from the correct memory address. After declaring the `_start` symbol, our crt0 clears the bss segment, registers the `exit` function, initializes the UART and finally calls the `main` function.

The linker script is the final component needed by the Libgloss. It tells the linker how to link the object files produced by the compiler. Our linker script maps the whole SDRAM on headsail as it's total available memory region, which it then splits into the text section, data and read-only data sections and various different initialization sections. Additionally, the linker reserves space for the stack and heap data structures according to the linker script definitions. Figure 4.4 shows an

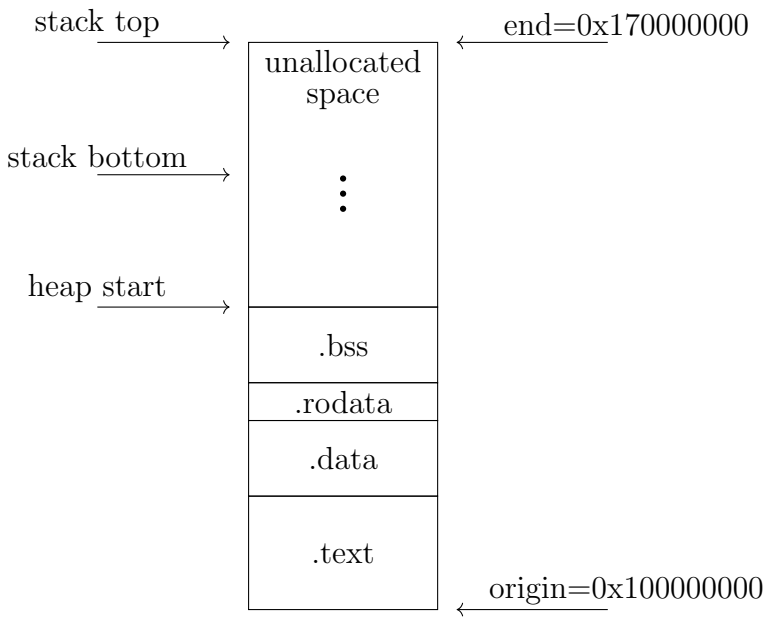

Figure 4.4 *Example configuration of linked program in memory*

example of how a binary is placed in to the Headsail's SDRAM according to the defined linker script. We always place the `_start` start symbol at the beginning of the SDRAM, followed by data sections. By default, the linker allocates 4 megabytes to the stack and rest is allocated to the heap. The allocations are controlled by specific flags defined in the linker script, that are visible to program.

4.5 TVM transformation scheme and code generation

After trial and error we settled on a TVM graph transformation behaviour that provided us with the most stable results in all the tested models.

We implemented our own QNN legalization pass for TVM in which, we look for a TVM canonicalized version of a TFlite quantized conv2d fusion layer. In TVM relay representation this pattern is composed of 4 nodes: `qnn.conv2d`, `nn.bias_add`, `qnn.requantize`, and `clip`. We also capture an optional pair of nodes at the end of main block. If the fusion layer is followed by `qnn.add` and `clip` we capture those nodes as well. This combination is seen in ResNet architecture as the operations to concatenate main branch and the skip branch.

From the found pattern we remove all the zero point correction from the `qnn.conv2d` and `qnn.requantize` nodes, if they are present. We learned that the TFlite models offered by MLPerf Tiny tended to use affine quantization with zero point being close to 128. Value this high doesn't work for Headsail DLA, since all base values over 0 would get clipped. We also remove the zero-point from the optional skip connection concatenation node if it was captured, even if the operation is not assigned to the accelerator.

After legalization we do pattern matching for two patterns. First is the standard 2D convolution followed by bias and second is grouped convolution followed by a bias. These patterns are assigned to be executed on the DLA using the high-level API symbols from the FFI.

In the code generation backend we traverse the legalized graph and operate on all the patterns that have been assigned for execution on the DLA. From each pattern we extract the embedded operation parameters to produce valid C code to call the high-level API operations. We also generate code to dynamically allocate memory for the intermediary results. Some of the operation parameters need to be dynamically extracted from the graph information.

Appendix A shows an example of the code generated for one convolution layer with the Headail TVM backend. First we include all the necessary libraries to each generated file. Then we generate the function definition that takes four parameters, from the AOT runtime. The first two are the buffers for layer input and weights, third one is the bias FIFO buffer, and last one is the result buffer for that layer. The body of the function consists of allocating an intermediary buffer for the results. This is needed in where we have multiple sequential convolutions in a row, which is not the case in this example. Next we have the actual API call which gets as it's parameters all the necessary information extracted from the original graph representation of the layer. Finally we copy the result of the operation from the intermediary buffer to the runtime output buffer and deallocate the intermediary buffer. TVM AOT runtime expected each external codegen call to return a 0 upon succesful completion.

4.6 Benchmarking

For the DLA there are two things we can benchmark. First is to look at the amount of convolution operations we can execute per time unit. This kind of throughput benchmarking is fine but doesn't tell us much about the real-life CNN performance, since it doesn't take into an account what proportion of the CNN workloads is actually Conv2d and the proceeding Bias and ReLU operations.

The second option is to model real-life use-cases of CNNs and benchmark the inference of the models. For this purpose we chose to use MLPerf Tiny Benchmark from MLCommons, which we describe in detail in section 3.4. From these benchmarks the Anomaly Detection is unacceleratable with the Headsail DLA since it uses FC-AutoEncoder network, which is based on fully connected layers and thus has no operations which our accelerator can execute.

Figure 4.5 shows the sequence of running the MLPerf Tiny benchmark on Headsail. We use a simple Python program running on a host PC as the runner, which reads the benchmark samples from the dataset and send them to Headsail over

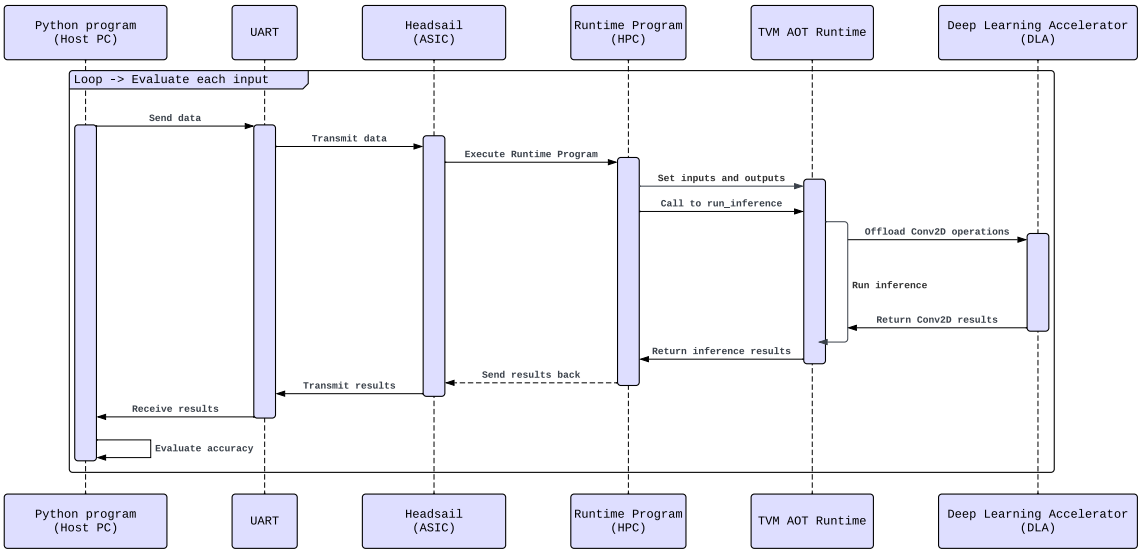

Figure 4.5 *Sequence diagram for MLPerf Tiny benchmarking on Headsail.*

UART. Since MLPerf Tiny only measures the performance of actual inference, we decided to use the relatively slow UART over SPI or I2C for the simplicity. HPC on Headsail runs a simple C-program to receive the data over UART and to set that data as input for the TVM AOT runtime. After setting the input and output the program calls the AOT runtime API to run inference for the given sample. AOT runtime then sequentially executes the model graph with the annotated graph patterns allocated for the DLA. Result of the inference is then transmitted over UART to the host PC, where the Python program receives it, finds the top-1 classification from the vector and updates the evaluation metric, before sending the next input for inference.

We also want to compare how the DLA compares to running the same benchmark without the accelerator. To do this we can invoke the TVM AOT runtime to be built without graph annotations, allocating all the convolution to HPC. This is necessary to conclude if the accelerator is actually able to accelerate the workloads meant for it.

The results of this benchmarking serve two purposes. First is to evaluate the suitability of the DLA for executing pretrained int8 quantized models. In an ideal case DLA can take any arbitrary quantized CNN and improve it's inference performance in comparison to just running it on the HPC. If the DLA reaches accuracies close to the ones of the reference models, we would consider it a success. The second purpose of this benchmark is to actually evaluate the performance of the HPC and the DLA ASIC in CNN workloads.

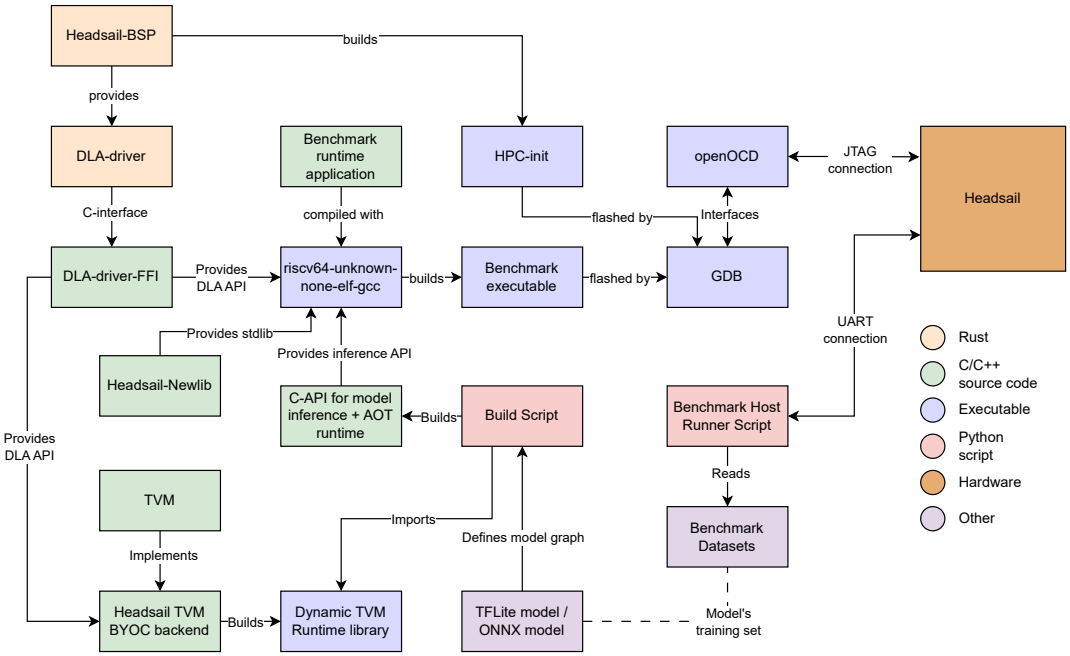

Figure 4.6 *Software architecture of accelerated DLA flow in Headsail with TVM runtime and a TFlite model*

4.7 Complete DLA Software Architecture

The figure 4.6 presents the software architecture of TVM annotated model deployment flow with Headsail DLA. The end goal of this flow is to build and executable binary that includes the AOT runtime with one of the MLPerf Tiny models to run inference on samples received over UART. The build flow is orchestrated with CMake and the model code generation with a Python script.

The first part of the build process is the TVM code generation. We read the targeted TFlite model with a Python build script and import it to TVM Relay graph. TVM then performs QNN-legalization pass for the graph defined in the Headsail's Python API and assign the defined patterns for execution on the DLA, producing a DLA compatible relay graph. This graph is then compiled to C source code with the TVM dynamic runtime which has been compiled with support for Headsail BYOC backend, which uses the symbols from DLA-driver's C-interface to execute the CNN operations. The DLA directives are embedded into the AOT graph to produce complete pipeline for executing inference on the specific model. This source code is then linked with our standard library implementation, the runner application and DLA-driver using riscv64-unknown-none-elf-gcc compilerstack to produce an executable binary that can be run on HPC.

Both of the CPUs in the Headsail are equipped with JTAG connector which we can connect to with OpenOCD [26]. We configure OpenOCD to provide us with

a GDB [13] interface, which we use to flash the executables. First we run HPC's initialization code on Sysctrl to turn on HPC and all the necessary peripherals, after which we can flash the benchmark executable onto HPC and begin the benchmark.

Running the benchmark is done with simple Python script which reads one input from the targeted benchmark task at a time and writes them to Headsail over UART for inference. The script then waits for the runner program on Headsail to respond with the result vector of the inference, from which it finds the top-1 classification and updates the precision score for the benchmark, before sending the next input.

Even though Headsail is the third SoCHub Soc, it had little existing software support for C. Previous SoCs had only support for Riscv-rt in rust. So a major part of this project involved setting up a Headsail compatible C-toolchain. Since Headsail has RISC-V CPUs we could use an already existing riscv-gnu-toolchain for the compiler, but we still had to set up a C standard library for the chip with custom version of Newlib Libgloss. Also, due to specific memory addressing decisions in the hardware, we needed to use `medany` code model compatible compiler and standard library when targeting the 64-bit processor.

5 Results

After finishing development of the software stack described in the chapter 4 we ran the MLPerf Tiny benchmark for two configurations. First being on the Headsail-VP with only single HPC core, and second on Headsail-VP with single HPC core and the DLA-VP working in tandem. Unfortunately due to problems with the ASIC, we were not able to include any results from the Hardware and thus this benchmark only serves the purpose of validating the ability of the Headsail to execute the reference models.

Table 5.1 *MLPerf Tiny benchmark results for HPC and HPC with DLA.*

Task	**Target**	**Accuracy**	**Precision**	**Recall**	**Passes MLPerf Tiny**
Keyword Spotting (KWS)	**HPC**	0.90	0.92	0.90	Yes
	HPC+DLA	0.90	0.92	0.90	Yes
Image Classifcation (IC)	**HPC**	0.88	0.88	0.88	Yes
	HPC+DLA	0.69	0.74	0.70	No
Visual Wake Words (VWW)	**HPC**	0.83	0.83	0.84	Yes
	HPC+DLA	0.50	0.50	0.50	No
Anomaly Detection (AD)	**HPC**	-	-	-	Yes
	HPC+DLA	-	-	-	No

The table 5.1 presents the results of the MLPerf Tiny benchmarks, for HPC standalone and HPC with 2D convolutions assigned to the DLA. As noted previously the Anomaly Detection task is unacceleratable due to the lack of convolutions, and thus it has equal performance between the runs.

When run without the DLA, just with HPC, Headsail passes the accuracy requirements for all the tasks as can be seen from the last column of the table. However this was not the case for runs using the DLA. With the exception of the keyword spotting task all the task failed to pass their required accuracy threshold. Since anomaly detection is not a CNN we couldn't accelerate it on the DLA.

As mentioned previously Renode isn't fitting for performance evaluation, and thus it isn't feasible to benchmark DLA-VP with MLPerf Tiny. We can however run the MLPerf Tiny tasks to see if the DLA reaches the required accuracy requirements of the benchmark. This essentially verifies the complete software stack and was especially useful during the development of DLA graph legalization pass.

Next sections go over the results from each of the tasks in more detail.

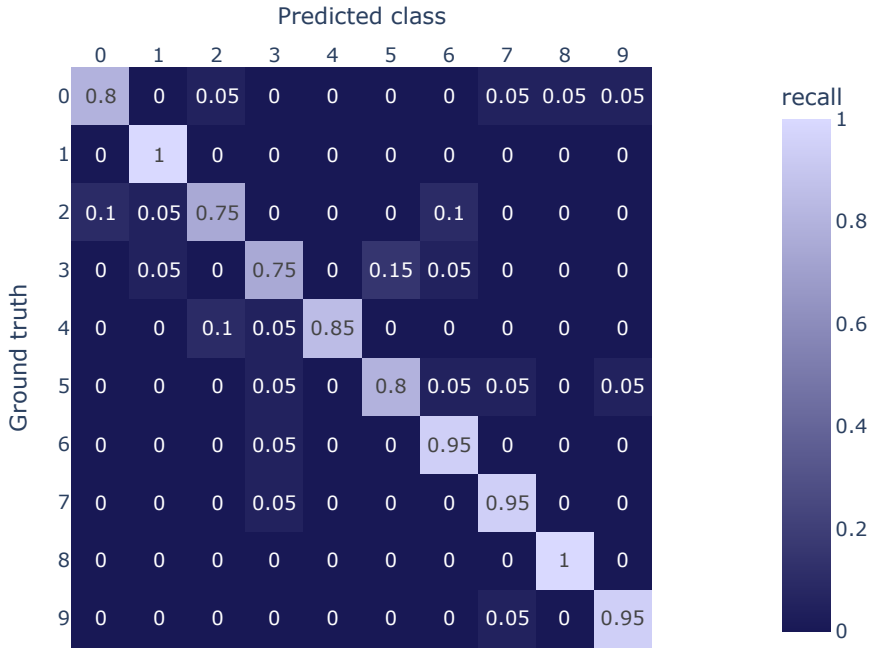

Figure 5.1 *Confusion matrix of image classification task results without DLA (N=200).*

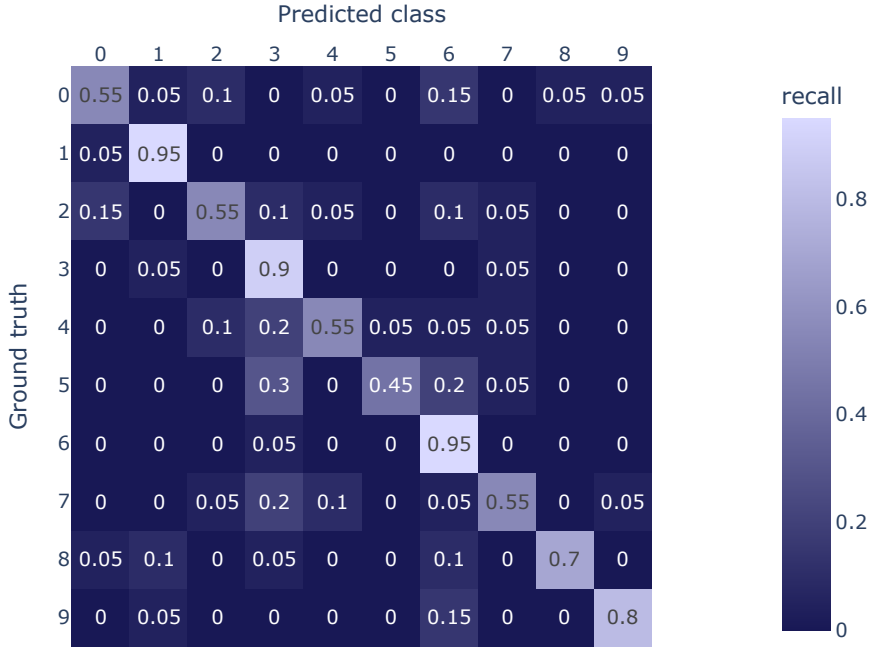

Figure 5.2 *Confusion matrix of image classification task results when using bias heuristic (N=200).*

5.1 Image classification

Figure 5.1 shows the results for running the IC task without the DLA as a class confusion matrix. With accuracy of 0.88 it easily passes the required accuracy threshold of 85% for submitting the IC benchmark scores. This result is inline with reference model, and as mentioned in section 3.4.1 the evaluation metric for the IC

task is Top-1, and thus we are only interested if the inference identified the correct class, no matter what was its confidence on that answer.

Figure 5.2 shows the results for the MLPerf Tiny image classification benchmark. From the result we can see that there is a major difference classification accuracy between classes. Classes 1, 3 and 6 have higher recall than the minimum asked by MLPerf Tiny, but entries in classes 0, 2, 4, 5 and 7 are misidentified almost half of the time. With total accuracy of 0.69 Headsail-DLA doesn't reach the accuracy threshold of the task.

When compared to the runs with DLA we can see that there are some similarities in which classes get confused with which classes, but the error seems to get exaggerated with the DLA. This discrepancy is almost completely explained by the clipping behaviour, since when DLA-VP is modified to output the complete result, this difference between the runs disappears and DLA performs similarly as the HPC only run.

5.2 Keyword spotting

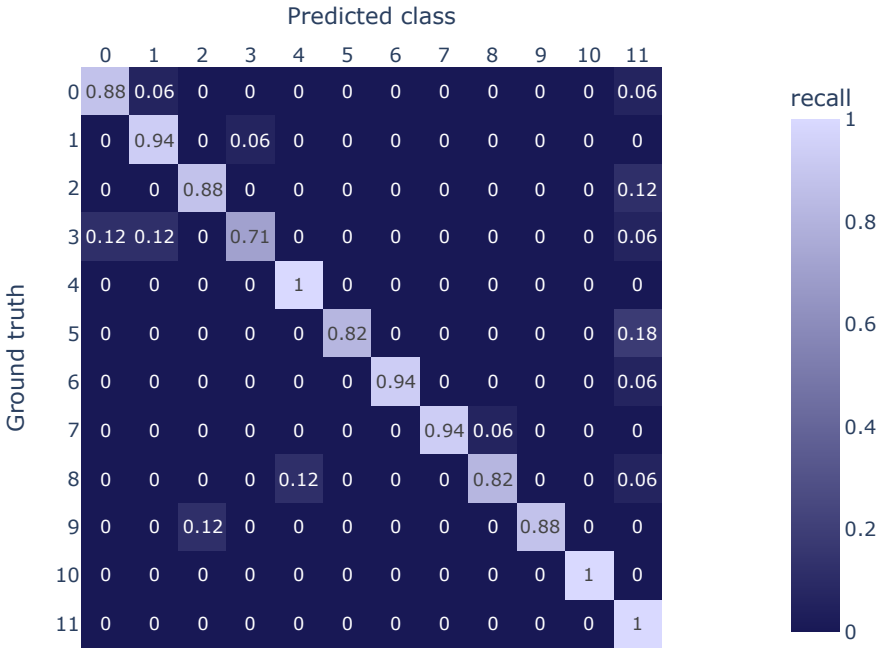

Figure 5.3 *Confusion matrix of keyword spotting task results without DLA (N=200).*

Figure 5.4 shows the results for the MLPerf Tiny keyword spotting benchmark ran on Headsail-VP with the DLA as a class confusion matrix. Similarly, as with the image classification task the evaluation metric was Top-1 accuracy. For the keyword spotting the DLA performed well, exactly hitting the required 90% minimum accuracy requirement.

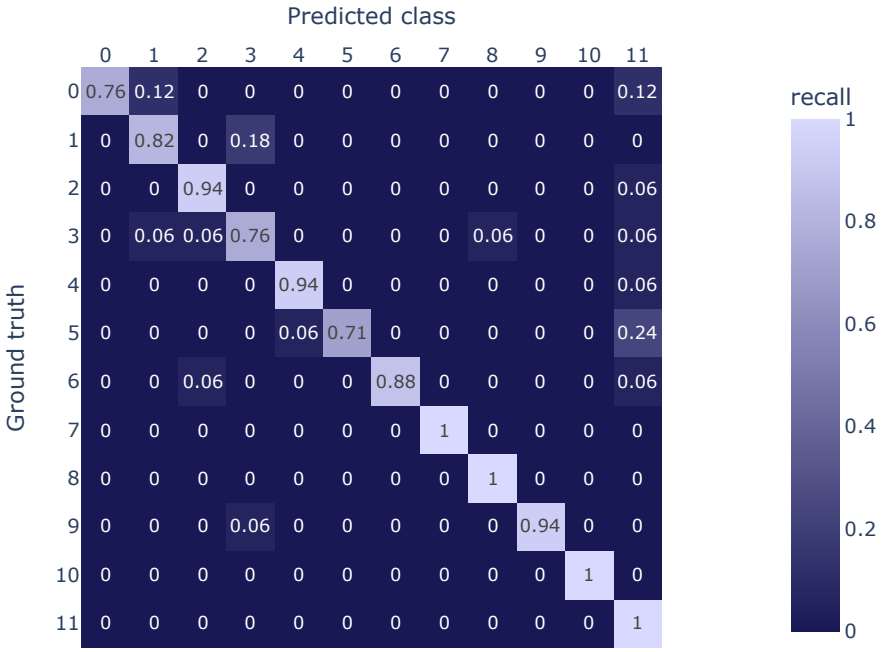

Figure 5.4 *Confusion matrix of keyword spotting task results when using bias heuristic (N=200).*

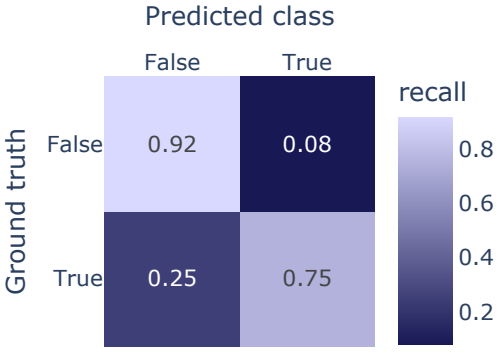

Figure 5.5 *VWW task on Headsail-VP without DLA*

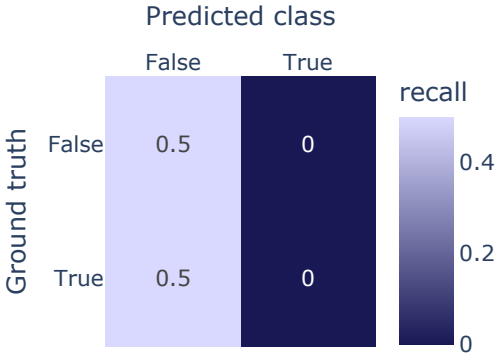

Figure 5.6 *VWW task on Headsail-VP with DLA*

Figure 5.7 *Visual wake word task results on Headsail*

The CPU only run of the benchmark performed as expected, reaching the required accuracy. From the confusion matrix shown in figure 5.3 we can see that the CPU only run performed more evenly in regard to which classes it could correctly classify, than the DLA run which had more classes it struggled with as well as more classes which it could perfectly classify.

5.3 Visual Wake Words

Figure 5.7 shows the results of the MLPerf Tiny visual wake word task for Headsail-VP. Subfigure 5.5 shows the results of the benchmark for the CPU only run. This

run exceeds the required 80% accuracy threshold with an accuracy of 83%. From the confusion matrix we can see that the reference model has a bias towards false predictions. This effect can be seen exaggerated in the benchmark run on DLA in figure 5.6. From the results of the DLA run we can conclude that the MLPerf Tiny reference model is not compatible with the DLA at this point, since accuracy of 0.5 for a binary classifier implies a random results.

At this moment it's unclear why this task performed so poorly on the DLA, but what we are seeing from the results is that the inherent weight and biases of the particular model. This is supported by the results of the HPC only run where the predictions steered noticeably towards False predictions. One possible issued we noticed is that the VWW reference model has very high bias values in its last convolution layer. As discussed earlier in section 4.2 might result in channel saturation issues when the bit-width of the bias is limited.

5.4 Anomaly Detection

The FC-AutoEncoder model used in the anomaly detection task isn't acceleratable on Headsail DLA, and thus we have results only for the HPC only run. The results are inline with what is expected from the reference model, passing the set AUC metric with a value of ***[ADD RESULST HERE]***.

6 Conclusions

During this Master's Thesis we were able to develop a virtual replica of a large ASIC subsystem in DLA-VP, that based on verification testing functions identically to its hardware counterpart. This gives us confidence in the claim that the Python Peripheral API in Renode can be successfully used to prototype more complex hardware components. Which in turn can be used to develop driver and application space software, before hardware is taped out. This might enable software developers to take greater part in hardware design than before using more familiar tools.

We were also to develop a software stack to capable for deploying, executing and accelerating inference for CNNs on the Headsail with the custom Deep Learning Accelerator. The stack consisted of a driver for the accelerator integrated in to the SOC's board support package, with a simple to use API for scheduling convolution operations. We also developed a TVM backend for the accelerator to generate C-source code with the TVM BYOC API, which allowed us to easily deploy models for the device. With Newlib we provided a C standrad library implementation compatible with Headsail, making it possible to build more complex C programs for the platform.

With MLPerf Tiny benchmark we were able to evaluate the accuracy of the DLA-VP when executing pretrained quantized convolutional neural networks. The result of which was that the DLA can handle some models. We were not yet able to run the benchmark on the actual ASIC, due to hardware issue, but we did in some limited manner observe that binaries for the Headsail-VP were compatible with the ASIC.

We also gained lot of understanding of the Headsail platform, embedded software architectures, the use of neural networks in embedded devices and general experience building software close to the hardware.

6.1 Future Work

As the accuracy results show there is space for improving the inference accuracy with the software stack. Different graph transformations should be considered to counteract the resolution limitations caused by the result clipping in the DLA. We also know that the driver is less the optimal when it comes down to moving data through the different software layers, and this should be addressed by carefully examining the driver implementation and the TVM facing interface.

As we have seen from other deep learning accelerators, the main bottleneck in these devices is generally the data transferring. Reducing or parallelizing data trans-

fers can provide us with noticeable performance increases. One way to introduce parallellity would to introduce preemptive weight buffering to the driver. By writing the weights of the next layer while the previous layers is still being processed should improve inference throughput.

From hardware point of view this project has highlighted many limitations regarding the DLA when considering actual use-cases for it. To enable the use of better data locality principles the next iteration of the Headsail could include a FIFO-based scaler unit in between Bias and ReLU steps. This would allow allocating the qnn.requantize operations to the accelerator, which would reduce the amount data transfers between the TVM runtime and the driver, possible improving performance while also fixing the issue caused by result clipping the original design, while keeping the output width as 8-bits. This would give better compatability with pretrained models.

To improve DL support on Headsail, there is a need for DLA specific training flow to be implemented with one of the major DL frameworks. This would enabled the creation of models with full compatibility with the DLA. Since development and training of models takes a lot of time and computing resource it wasn't included in this work but should be pursued with more time.

As discussed earlier in this work, fully connected layers are still widely used as major component in auto-encoders and classifiers layers in many networks, and thus accelerating them can introduce significant performance increases for some networks. To enable accelerating all the tasks in MLPerf Tiny we should consider if the next iteration of the MAC array could be modified to support fully connected layer without greatly increasing the area in the chip.

In addition to deep learning applications the DLA could be used in other workloads. Kernel based image processing applications can benefit from similar acceleration as the DNN applications we have showcased in this work. For example Gaussian filters for blurring and edge detection can be implemented with similar windowing approach as two-dimensional convolution is implemented in the accelerator.

References

[1] Colby Banbury et al. "MLPerf Tiny Benchmark". In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks* (2021).

[2] Colby Banbury et al. *MLPerf Tiny Benchmark*. arXiv:2106.07597 [cs]. Aug. 2021. DOI: 10.48550/arXiv.2106.07597. URL: http://arxiv.org/abs/2106.07597 (visited on 08/08/2024).

[3] Jeremy Bennett. *Howto: Porting Newlib: A Simple Guide*. Tech. rep. Application Note 9. Licensed under Creative Commons Attribution 2.0 UK: England & Wales License. Embecosm, July 2010. URL: https://www.embecosm.com/appnotes/ean9/html/index.html.

[4] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". In: *IEEE Journal of Solid-State Circuits* 52.1 (Jan. 2017). Conference Name: IEEE Journal of Solid-State Circuits, pp. 127–138. ISSN: 1558-173X. DOI: 10.1109/JSSC.2016.2616357. URL: https://ieeexplore.ieee.org/document/7738524 (visited on 10/21/2024).

[5] Shuangshuang Chen and Wei Guo. "Auto-Encoders in Deep Learning—A Review with New Perspectives". en. In: *Mathematics* 11.8 (Jan. 2023). Number: 8 Publisher: Multidisciplinary Digital Publishing Institute, p. 1777. ISSN: 2227-7390. DOI: 10.3390/math11081777. URL: https://www.mdpi.com/2227-7390/11/8/1777 (visited on 11/06/2024).

[6] Tianqi Chen et al. "TVM: an automated end-to-end optimizing compiler for deep learning". In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594. ISBN: 9781931971478.

[7] Tianshi Chen et al. "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning". In: *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ASPLOS '14. New York, NY, USA: Association for Computing Machinery, Feb. 2014, pp. 269–284. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541967. URL: https://doi.org/10.1145/2541940.2541967 (visited on 10/29/2024).

[8] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. "Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs?" In: *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. ISSN: 2379-3155. Dec. 2010, pp. 225–236. DOI: 10.

1109/MICRO.2010.36. URL: https://ieeexplore.ieee.org/abstract/document/5695539 (visited on 10/28/2024).

[9] ONNX Runtime developers. *ONNX Runtime*. https://onnxruntime.ai/. Version: x.y.z. 2021.

[10] Marco Di Natale and Alberto Luigi Sangiovanni-Vincentelli. "Moving From Federated to Integrated Architectures in Automotive: The Role of Standards, Methods and Tools". In: *Proceedings of the IEEE* 98.4 (Apr. 2010). Conference Name: Proceedings of the IEEE, pp. 603–620. ISSN: 1558-2256. DOI: 10.1109/JPROC.2009.2039550. URL: https://ieeexplore.ieee.org/document/5440059 (visited on 10/28/2024).

[11] Sorin Draghici. "On the capabilities of neural networks using limited precision weights". In: *Neural Networks* 15.3 (Apr. 2002), pp. 395–414. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(02)00032-1. URL: https://www.sciencedirect.com/science/article/pii/S0893608002000321 (visited on 10/29/2024).

[12] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark silicon and the end of multicore scaling". In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. ISSN: 1063-6897. June 2011, pp. 365–376. URL: https://ieeexplore.ieee.org/document/6307773 (visited on 10/28/2024).

[13] Free Software Foundation. *GNU Debugger (GDB)*. Version 13.2, Accessed: 2024-11-08. 2023. URL: https://www.gnu.org/software/gdb/.

[14] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. Cambridge, MA, USA: MIT Press, 2016.

[15] David J. Greaves. *Modern System-on-Chip Design on Arm*. ARM Education Media, 2021, pp. 2–8.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[17] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV]. URL: https://arxiv.org/abs/1704.04861.

[18] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. en. Apr. 2017. URL: https://arxiv.org/abs/1704.04861v1 (visited on 10/03/2024).

[19] Animesh Jain, Shoubhik Bhattacharya, Masahiro Masuda, Vin Sharma, and Yida Wang. *Efficient Execution of Quantized Deep Learning Models: A Compiler Approach*. en. arXiv:2006.10226 [cs]. June 2020. URL: http://arxiv.org/abs/2006.10226 (visited on 09/18/2024).

[20] Yuma Koizumi et al. *Description and Discussion on DCASE2020 Challenge Task2: Unsupervised Anomalous Sound Detection for Machine Condition Monitoring*. arXiv:2006.05822. Aug. 2020. DOI: 10.48550/arXiv.2006.05822. URL: http://arxiv.org/abs/2006.05822 (visited on 10/18/2024).

[21] Raghuraman Krishnamoorthi. *Quantizing deep convolutional networks for efficient inference: A whitepaper*. arXiv:1806.08342. June 2018. DOI: 10.48550/arXiv.1806.08342. URL: http://arxiv.org/abs/1806.08342 (visited on 10/24/2024).

[22] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. 2009.

[23] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. arXiv:1405.0312. Feb. 2015. DOI: 10.48550/arXiv.1405.0312. URL: http://arxiv.org/abs/1405.0312 (visited on 10/18/2024).

[24] MaderNoob. *galloc - good_memory_allocator*. Version 0.1.7. 2024. URL: https://crates.io/crates/good_memory_allocator/0.1.7.

[25] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[26] OpenOCD Development Team. *Open On-Chip Debugger (OpenOCD)*. Version 0.12.0, Accessed: 2024-11-08. 2023. URL: https://openocd.org/.

[27] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. arXiv:1912.01703 [cs, stat]. Dec. 2019. DOI: 10.48550/arXiv.1912.01703. URL: http://arxiv.org/abs/1912.01703 (visited on 10/07/2024).

[28] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG]. URL: https://arxiv.org/abs/1912.01703.

[29] A Rautakoura et al. "Ballast : Implementation of a Large MP-SoC on 22nm ASIC Technology". eng. In: Fabelo, H. 2022.

[30] Inc. Red Hat and Various Contributors. *Newlib: A C Library*. https://sourceware.org/newlib/. 2023.

[31] Mozilla Research. *cbindgen: Generate C/C++ headers for Rust libraries*. Accessed: 2024-10-22. 2021. URL: https://github.com/mozilla/cbindgen.

[32] Janosh Riebesell and Stefan Bringuier. *Collection of standalone TikZ images*. Version 0.1.0. 10.5281/zenodo.7486911 - https://github.com/janosh/tikz. Aug. 9, 2020. DOI: 10.5281/zenodo.7486911. URL: https://github.com/janosh/tikz (visited on 01/01/2023).

[33] David E. Rumelhart and James L. McClelland. "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. Conference Name: Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations. MIT Press, 1987, pp. 318–362. ISBN: 978-0-262-29140-8. URL: https://ieeexplore.ieee.org/document/6302929 (visited on 11/06/2024).

[34] Kashif Shaheed et al. "DS-CNN: A pre-trained Xception model based on depth-wise separable convolutional neural network for finger vein recognition". In: *Expert Systems with Applications* 191 (2022), p. 116288. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2021.116288. URL: https://www.sciencedirect.com/science/article/pii/S0957417421015943.

[35] SiFive. *All Aboard Part 4: RISC-V Code Models*. https://www.sifive.com/blog/all-aboard-part-4-risc-v-code-models. Accessed: 2024-11-11. 2024. URL: https://www.sifive.com/blog/all-aboard-part-4-risc-v-code-models.

[36] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* 105.12 (Dec. 2017). Conference Name: Proceedings of the IEEE, pp. 2295–2329. ISSN: 1558-2256. DOI: 10.1109/JPROC.2017.2761740. URL: https://ieeexplore.ieee.org/document/8114708 (visited on 10/28/2024).

[37] Mingxing Tan and Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. arXiv:1905.11946. Sept. 2020. DOI: 10.48550/arXiv.1905.11946. URL: http://arxiv.org/abs/1905.11946 (visited on 11/06/2024).

[38] Robot Framework Team. *Robot Framework*. Accessed: 2024-11-06. 2008. URL: https://robotframework.org/.

[39] The Rust Embedded - RISC-V Team. *riscv-rt: Minimal runtime / startup for RISC-V CPU's*. https://crates.io/crates/riscv-rt. Accessed: 2024-11-12. 2024.

[40] Pete Warden. *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. arXiv:1804.03209 [cs]. Apr. 2018. DOI: 10.48550/arXiv.1804.03209. URL: http://arxiv.org/abs/1804.03209 (visited on 08/09/2024).

[41] F. Zaruba and L. Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (Nov. 2019), pp. 2629–2640. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114.

APPENDIX A. Headsail DLA TVM Backend Codegen Example

Program 1 *Code generated by BYOC backend for Headsail DLA to execute Conv2D with bias pattern.*

```
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <tvm/runtime/c_runtime_api.h>
#include <dlpack/dlpack.h>
#include <dla_driver.h>

//This was generated with headsail codegen
int tvmgen_default_headsail_main_0(int8_t* headsail_0_i0, int8_t*
    ↪ headsail_0_i1, int* headsail_0_i2, int* out0) {
  int* buf_0 = (int*)malloc(73728);

  dla_tvm_qnn_conv2d_bias(headsail_0_i0, headsail_0_i1,
      ↪ headsail_0_i2, buf_0, 3, 96, 96, "HWC", 8, 3, 3, 3, "HWCK",
      ↪ 8, 0, 1, 0, 1, 0, 2, 2, 0, 5);

  memcpy(out0, buf_0, 73728);
  free(buf_0);

  return 0;
}
```

APPENDIX B. Something completely different

You can append to your thesis, for example, lengthy mathematical derivations, an important algorithm in a programming language, input and output listings, an extract of a standard relating to your thesis, a user manual, empirical knowledge produced while preparing the thesis, the results of a survey, lists, pictures, drawings, maps, complex charts (conceptual schema, circuit diagrams, structure charts) and so on.