Tampere University

Väinö-Waltteri Granat

Zero to DLA: Building Software Support For Custom RISCV SoC To Run Complex Neural Networks

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

Abstract

Väinö-Waltteri Granat: Zero to DLA: Building Software Support For Custom RISCV SoC To Run Complex Neural Networks
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

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 smart phones.

This interest in neural networks has coincided with the industry's move to heterogenous System-on-chip solutions being used in consumer and professional devices, to imporove 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 it's 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 a image classifier are 3 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.

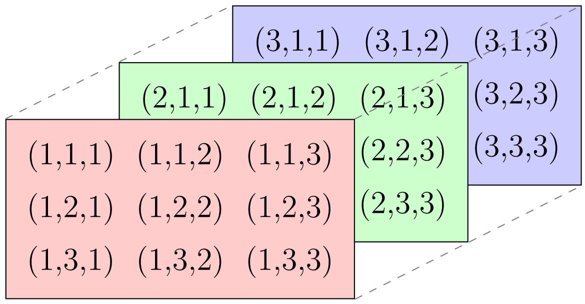

Figure 2.1 *RGB array*

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 a image binay 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 consists 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

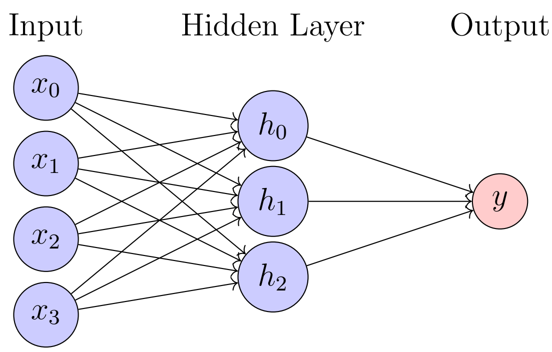

***Figure* 2.2** *Simple fully connected neural network with two layers.*

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 a N-dimensional which the non-linear function inhabits, where each of the vertices introduces it's 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. [30]

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 a element in one layer's input x and the vertices are the weights that connect each

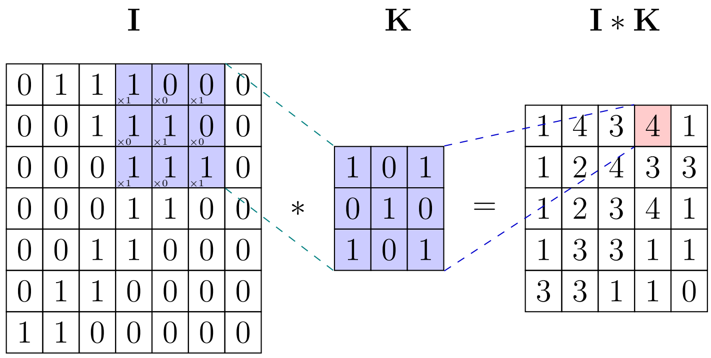

Figure 2.3 *Visualization of 2D convolution being done for 7x7 input with 3x3 kernel. Adapted from [28].*

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 layer 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 a useful operations 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 specific 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, it natural to 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 neural networks convolution is often implemented as cross-correlation but still called convolution, 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 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 non important for the particular class it's being applied. If bias is large or positive it means that the feature is important.

In mathematic notation bias is defined as such

$$y = x^T W + b \tag{2.10}$$

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.

2.2.4 Activation function

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

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.11}$$

and,

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} - e^{-x}}, \tag{2.12}$$

to introduce non-linearity. Both functions suffer from the fact that they are expensive to calculate and exhibit the vanishing gradient problem. [10]

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

$$ReLU(x) = \begin{cases} 0, & \text{for} \leq 0 \\ x, & \text{otherwise.} \end{cases} \tag{2.13}$$

Figure 2.6 shows how ReLU and sigmoid non-linearly scale values close to $x = 0$.

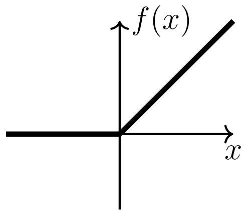

Figure 2.4 *Rectified linear unit (ReLU)*

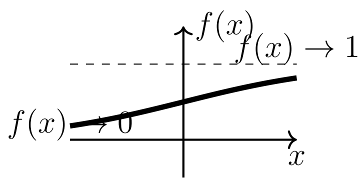

Figure 2.5 *Sigmoid activation*

Figure 2.6 *Comparison of ReLU and Sigmoid activation functions*

Combining 2 dimensional affine transformation and ReLU gives as the basic Conv2D layer found in most image classification network, like Resnet [13] and MobileNet [14].

$$ReLU(Conv2D(I, K) + b) \tag{2.14}$$

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 are presented as graphs, where nodes are layers or fused layers and paths are the data flow directions. Figure 2.7 displays the equation 2.14 as a simple graph were data flow is always from one layers output to next layers input.

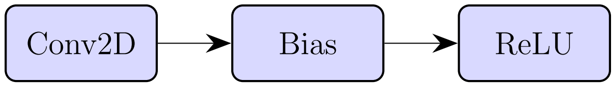

Figure 2.7 *Feed forward relationship between Conv2d, bias and ReLU layers.*

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 [21], a popular framework for training, considers Conv2D, bias and ReLU separate layers and the combination of the a fused layer, where as in Pytorch [23] 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.

Connection can also branch, and the same layer output can be used as input for multiple nodes. Resnet [13] 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.

[Write about depthwise conv2d]

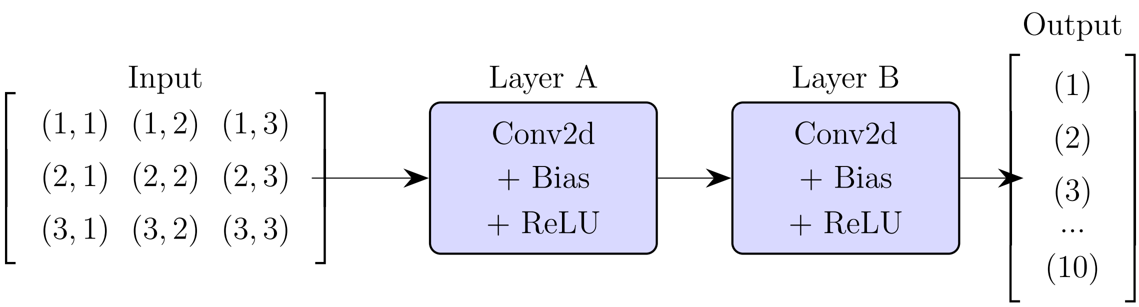

Figure 2.8 *Simple feed forward network with two Conv2D with bias and ReLU activation layers.*

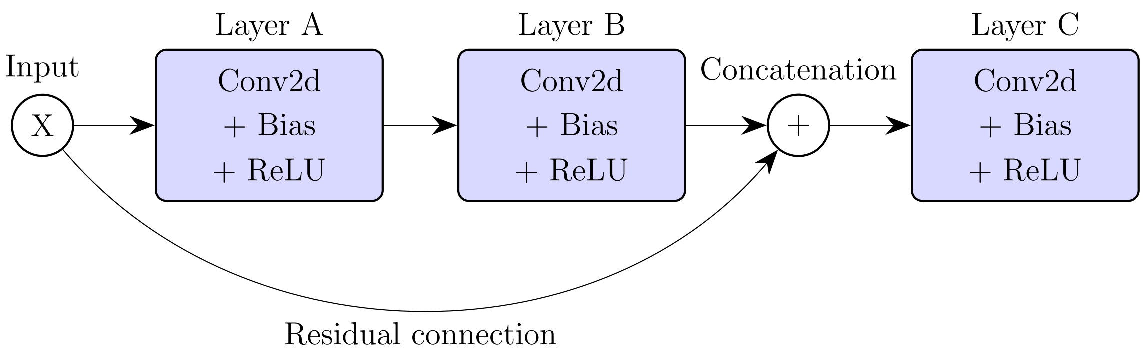

Figure 2.9 *Residual feed forward network with fused layers.*

2.4 Neural Model Training

To perform a given task all neural networks need to be trained. This means that 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 apply 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 a 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 the 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 it's own dimension and thus adding a parameter to the derivative.

$$g_K = \nabla L(\hat{y}, z_K) = \frac{\partial L}{\partial a_K} \cdot \frac{\partial a_K}{\partial z_K}, \tag{2.15}$$

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 dependant on the activation of the previous layer

$$g_{K-1} = \nabla L(\hat{y}, z_{k-1}) = g_K \cdot \nabla L(z_K, z_{K-1}) \tag{2.16}$$
$$= \frac{\partial L}{\partial a_K} \cdot \frac{\partial a_K}{\partial z_K} \cdot \frac{\partial z_K}{\partial a_{K-1}} \cdot \frac{\partial a_{K-1}}{\partial z_{K-1}}. \tag{2.17}$$

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

$$\nabla L(\hat{y}, x). \tag{2.18}$$

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

$$z_K = a_{K-1} W_K + b_K \Rightarrow \frac{\partial L}{\partial W_K} = \frac{\partial L}{\partial z_K} \cdot a_{K-1} \tag{2.19}$$

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

$$W_k = W_K - \eta \frac{\partial L}{\partial W_K}. \tag{2.20}$$

Biases are also update with a similar equation

$$b_k = b_K - \eta \frac{\partial L}{\partial b_K}. \tag{2.21}$$

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. Where as inference can be done on different kinds of hardware configurations from serverfarms to micro-controllers, since the inference is less computationally expensive.

Most training frameworks use 32-bit floating point values to represent rational numbers. [24][21]. 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 quantisized, the network is first trained with high accuracy floating points and quantisized 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 operations In recent years big players like NVIDIA have started to utilize more and more quantized integer models. This is due to the fact that as the amount of parameters in models like GPT, has been growing exponentially. Often there are significant performance gains available by reducing the granularity of the parameters without a major loss in model accuracy [8, 17]. Standard floating point value has a width of 32-bits, where as 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 quantisized. 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 quantisize the floating point parameteres to integers

$$x_Q = clamp(0, 255, round(\frac{x_{float}}{\Delta}) + z) \tag{2.22}$$

where x_{float} is the original parameter values, x_Q is the quantisized 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

$$\begin{aligned} clamp(a, b, x) &= a \qquad \text{if } x \leq a && (2.23)\\ &= x \qquad \text{if } a < x < b && (2.24)\\ &= b \qquad \text{if } x \geq b && (2.25)\end{aligned}$$

To convert quantisized parameters back to the orignal paramter values affine quantization uses the following operation

$$x_{float} = (x_Q - z)\Delta. \tag{2.26}$$

This back conversion has no need for the clamping operation or rounding, since the floating point range is larger then 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 of the equal, which is desired of signed integer based accelerators. The equation for quantizising a parameter with the uniform symmetric quantization is as follow

$$x_Q = clamp(-128, 127, round(\frac{x_{float}}{\Delta})). \tag{2.27}$$

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

$$x_{float} = x_Q \Delta. \tag{2.28}$$

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 emulate inference, so that the performs equally well for unseen data as it does with training data. The point of validating after training is to ensure 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 networks 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.

$$\text{Top-1 Accuracy} = \frac{1}{N}\sum_{i=1}^{N}\delta(\hat{y}_i, \text{argmax}(f(x_i))) \tag{2.29}$$

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 [24]

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.30}$$

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} \tag{2.31}$$

$$\begin{aligned}\text{AUC} &= \int_{\text{FPR}=0}^{\text{FPR}=1} \text{TPR}(\text{FPR})\, d(\text{FPR}) && (2.32)\\ &\approx \sum_{i=1}^{n-1} \frac{(\text{FPR}_{i+1} - \text{FPR}_i)\cdot(\text{TPR}_{i+1} + \text{TPR}_i)}{2}, && (2.33)\end{aligned}$$

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 [9], has shifted the focus of CPU designers towards homogeneous multi-core architectures. To overcome Moore's law chip makers have been moving towards heterogenous architectures. Heterogeneous architectures consists 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. [6]

System-on-chips (SoCs) are a form of heterogenous computing, where multiple different computing elements and electronic systems are integrated into one circuit. [12] 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 heterogenous architectures is it's 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 systems coherency. The single chip design of SoCs give them a noticeable advantage in terms of physical size and energy consumption, when compared to traditional heterogenous 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.

[7]

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 it's 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 prvious 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.

DianNao is one of the first ASIC accelerators targeting convolutional deep neural networks [5]. It's based on a pipelined NFU, where the multiply and accumulation operations are implemented as a pipeline along side bias and sigmoid activations. This is unlike more modern designs where the main computational element is an computational array.

Eyeriss [3] 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.

MAC or Multiply-Accumulate operation is a ***[Explain how MAC array works.]*** Neurocube [22], Flexflow [20] Google Edge TPU [11],

3 Methodology

This section goes over the technologies use 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. 1

3.1 Headsail

Headsail is the third Soc build by the SocHub research group [25]. 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 [32]. Headsail includes a wide variety of different peripherals, one of which is a custom build the Deep Learning accelerator. For I/O headsail has UART, SPI and I2C connectivity. In addition CPU bootrams, Headsail features 256 megabytes of SDRAM and 128 kilobytes of shared SRAM. This abundance of memory gives lots of space for

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 either or both of the aren't 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 configure from the register interface of the DLA. DLA has a simple 32 bit RISCV 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.

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

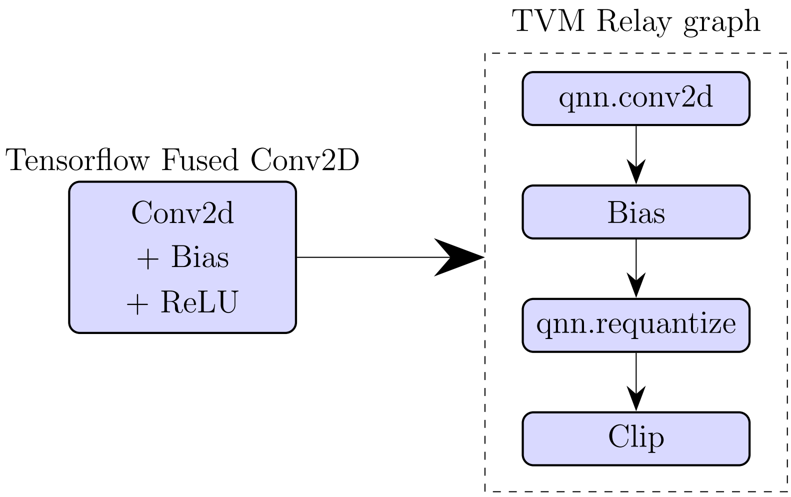

Figure 3.1 *Tensorflow layer to TVM relay graph conversion*

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 has it's own graph representation for neural networks called Relay IR. Like the traditional graph representation Relay IR represents network layers as nodes in a abstract syntax tree, where the data flow of the networks is shown as the relationship between parent and child nodes, where parent nodes output is the input of the child node.

TVM is able to 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 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 quantisized 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. [4]

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 operations 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, for it's simplicity.

3.2.2 Graph Transformations

To assign calculations for an accelerator the Relay graph of a network needs to be transformed. 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.

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 quantisized network most models use a zero-point of -128 to mimic signed behaviour. If this network is run on a accelerator with support for signed intergers, this zero point needs to be changed to 0, since there is no need for the

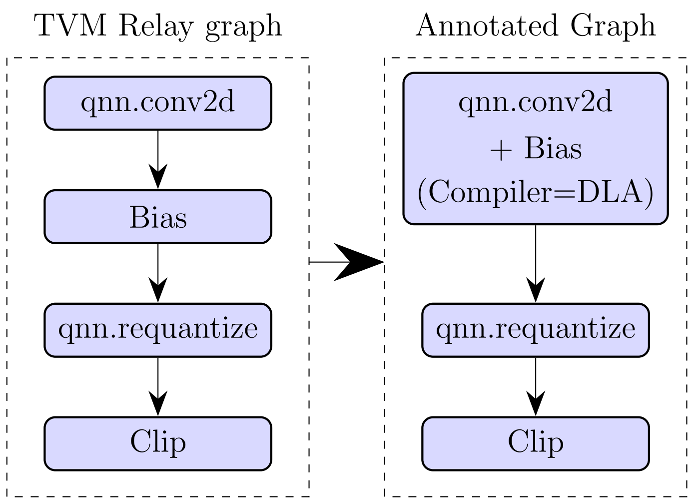

Figure 3.2 *Graph transformation for DLA-VP*

adjustment. If this network is run on a accelerator with support for signed intergers, 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 offered APIs for integrating new devices into TVM. In the BYOC API developers define how they want the annotated patterns to be executed on their accelerator.

In the case of the Headsail DLA we offer a high level calls Conv2d, Conv2d + Bias, Conv2d + ReLU and Conv2d + Bias + ReLU. BYOC for Headsail DLA then extracts the needed values from the relay graph and generates C code with the extracted values to call the high level API.

In addition to the C++ backend for BYOC we define a python API for performing the necessary legalization and annotation for the graph.

3.2.4 TVM on baremetal

TVM also provides a tool to run TVM models on baremetal platforms called microTVM. MicroTVM is only dependant on the C standard library and thus can be used in any baremetal 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 c-runtime to produce executable binaries to run the network.

With custom code generation it's also possible to define baremetal compatible accelerator nodes, which the TVM runtime is able to assign layers for during the C source code generation. [4]

3.2.5 TVM quantization

TVM implements quantization with it's own QNN-dialect [15]. The QNN-dialect separates quantized and non-quantized operations from each other by categorizing the quantized operations under the `qnn.op` class. The QNN operation have additional arguments in to indicate scaling factor and zero-point for inputs and outputs In addition to having quantizised version of the DL operations the QNN dialect introduces operations for moving data across from non-quantisized domain to quantisized and back. This is done with the `qnn.quantize`, `qnn.dequantize` and `qnn.requantize` operations. `qnn.quantize` performs affine quantization to a floating point tensor to produce a corresponding quantisized integer tensor, where as `qnn.dequantize` reverse the affine quantization. `qnn.requantize` converts a quantizised tensor to another quantisized tensor corresponding with a different scale than the original.

TVM enables compatibility of quantisized networks with other major DNN frameworks like tensorflow with QNN-dialect aware graph parsing. When model from a another framework, the framework specific quantization operations get converted to QNN nodes. The resulting graph is then passed trough 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 int8 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 software development framework, which enables developers to use principles of continuous integration when writing hardware dependent code. In essence Renode is a hardware emulator which 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. This project implements the DLA hardware design as a dynamic python peripheral.

While renode is a 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.

3.4 TinyPERF

MLPerf Tiny is a benchmarking suite for benchmarking ML inference in low power targets, like MCUs with Deep Learning Accelerators. [1] TinyPerf 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 TinyPERF aims to suit most kinds of targets, they give the users lots 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 TinyPERF can be used to compare between signed and unsigned integer accelerators, as well as with FP8 and FP32 accelerators.

3.4.1 Image Classification

The image classification task aims to classify images from the CIFAR-10 dataset [18]. CIFAR-10 includes 32x32 RGB color images, each belonging to one class. The 10 classes of CIFAR-10 are labeld as, airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck.

To do this classification TinyPERF uses Resnet. Resnet is an implementation of ImageNet, 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 on person is present in an image from the MSCOCO 2014 dataset [19]. The training dataset consists of RGB images preprocessed to 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 [14] Backbone of MobileNet consists of 13 depthwise 2 dimensional convolution layers each followed by a $K = 1x1$ 2D convolution. After backbone there is a simple average pooling layer followed by binary classifier.

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 [31], 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.

TinyPERF 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. [29] Where MobileNet used 13-layers DS-CNN only has four depthwise 2D convolution layers each followed by a pointwise convolution layer. Which makes it the smallest model in the benchmark, both by the number of layers and parameters. Similarly to MobileNet DS-CNN uses FC-layer for classifier.

3.4.4 Anomaly Detection

The dataset for this task is the DCASE2020 dataset [16]. 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 [**FC-Auto**]. The model consists of 10 FC-layers with ReLU activations, from which 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.

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 TinyPerf benchmark used to evaluate the performance of the DLA.

4.1 Headsail-VP

To enable software development for Headsail before the arriving 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 benefit of developing a virtual versions of Headsail was that we hoped it would enable faster software development, due to the limited accessibility of the ASIC as well as the possibility to modify the design in the case hardware bugs. Additionally this knowledge in developing virtual prototypes of chip could allows us the better define the hardware specs based on the software demonstrators we develop.

4.1.1 DLA-VP

The virtual prototype of the DLA was implemented as a Renode python peripheral. The DLA-VP supports all the same operations as the corresponding 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.

DLA-VP implements the register interface as a list, with the same length. Since numbers in python can be infinite length we had to carefully sanitize all the transactions to the registers so that no accesses of 8-bit values happened.

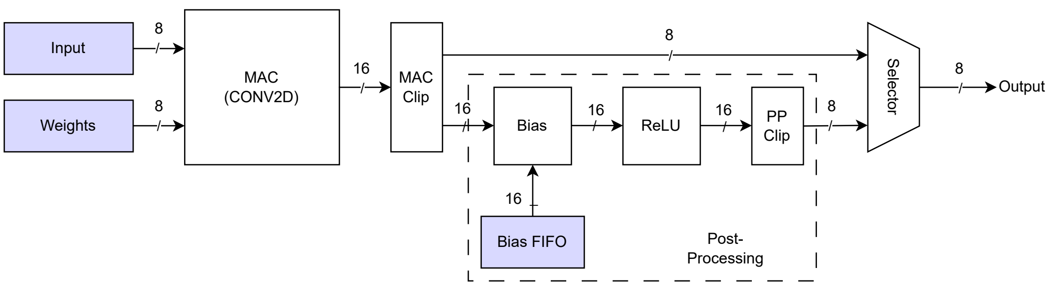

Figure 4.1 *Architecture of accelerated DLA flow in Headsail with TVM runtime and a Pytorch model*

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 it's memory region. The processing loop 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 peripherals memory region.

4.2 Software support

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 RISCV 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.

We also developed a Board Support Package for Headsail, which provides drivers for the different peripherals in the SoC. Most importantly for this project, the driver for the DLA is included in the BSP. The DLA driver is implemented in two layers. First is the low level layer which implements functions that directly target the register interface in the DLA to drive the hardware. The second is the high level layer which abstracts the low level layer to provied simple calls for the 3 different DLA operations. The high level layer also provides the external symbols for DLA operations used by the TVM code generation.

4.3 Porting Newlib

Newlib is an implementation of the C standard library meant for use in embedded devices [26]. We chose to use Newlib for our C standard library in Headsail since it's known to be relatively easy to port for new platforms. Newlib 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 RISCV we didn't need to modify Newlib core in anyway. Second part of Newlib is called Libgloss, which implements the platform dependant features. Since Headsail is a custom platform we needed to implement most of the libgloss for it from scratch.

Porting libgloss involves implementing 16 syscalls, crt0 and a linker script. From the 16 syscalls only some are mandatory for us to implement, because we are targeting only bare-metal applications. The table 4.1 shows all the libgloss syscalls 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 do anything except return an error. For example the fork syscall duplicates 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.

Program 4.1 *Minimal implmentation of the fork() syscall in Newlib Libgloss*

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

The system calls can be implemented either in non-reentrant or reentrant way. Reentrant system calls are thread safe but require an additional argument, reentrancy structure, to be passed. Reentrancy strucutre holds local values specific for that instance of the function call, where as 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. [2]

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

Table 4.1 *Newlib Syscalls and Implementation Status*

In addition to the syscalls libgloss needs to have a crt0, which is a small program that includes a `_start` symbol, call to `main` function and some global definitions and optionally hardware initalizations. The `_start` symbol signifies the beginning of a C program and is used by the linker to place the program to start from the

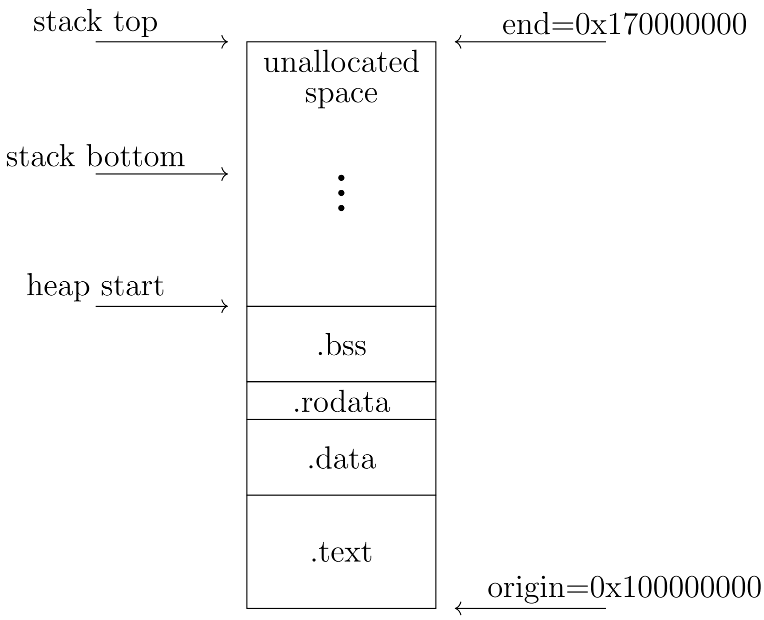

Figure 4.2 *Example configuration of linked program in memory*

correct memory address. After declaring the `_start` symbol out crt0 clears the bss segment, registers the `exit` function, initializes the UART and finally calls the `main` function.

The linker script 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 sections to 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 4.2 shows an 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.

4.4 DLA Software Architecture

The figure 4.3 shows the high level software architecture of TVM annotated model deployment flow with Headsail DLA. First we use high level operations in DLA driver to provied external symbols for Conv2d, Bias and ReLU. These are referenced in the Headsail build of TVM dylib when built with the Headsail custom code generation option.

In other branch we train and optimize a quantisized convolutional neural network with Pytorch and convert the produced model into a ONNX graph. We then use a python script to load in the Headsail TVM dylib which is used to generate C source

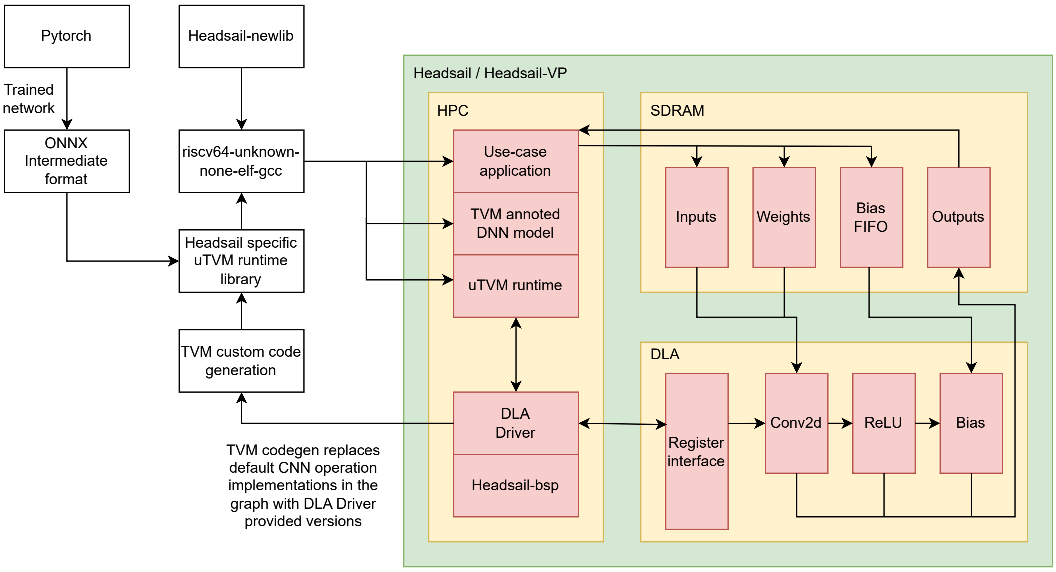

Figure 4.3 *Architecture of accelerated DLA flow in Headsail with TVM runtime and a Pytorch model*

code for the model from the annoteted ONNX graph. This code now includes calls to the DLA driver operations.

Finally we produce an executable binary by combining the microTVM C runtime from the Headsail TVM build, generated C source code for the model, Headsail-bsp for the board functions, Headsail-newlib for the C standard library and finally the use case program.

4.5 DLA Driver

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. The driver is implemented as Rust module separate from the rest of the BSP.

4.5.1 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_A_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 4.2 *Example call to DLA highlevel 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>,
}
```

Program 4.2 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 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 channels 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` tell 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, where as stride of 2 means that after a sample, we move two spaces to the particular direction etc.

4.5.2 Output

To get input from the DLA we first need to define it's input address. Theoretically this address can be any memory address visible to the control processors, but in the

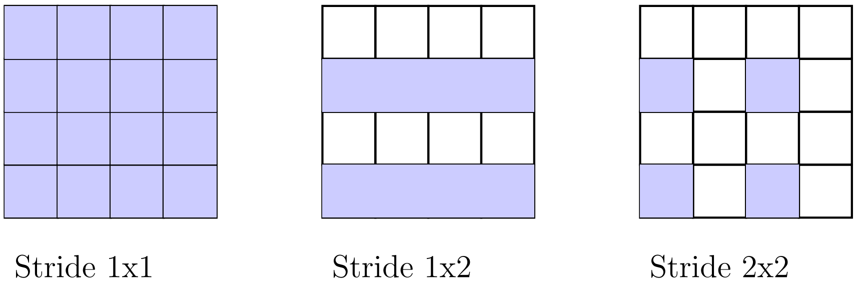

Figure 4.4 *Effect of stride to sampling*

driver we have limited the number of possible output addresses only to the DLAs 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

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

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

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

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.

Since 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.

4.5.3 C-Interface

To enable TVM BYOC to call the DLA drivers high-level interface we needed to great 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 [27]. 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 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.

4.6 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 fine but doesn't tell us much about the real-life CNN performance, since it doesn't take in to account what proportion of the CNN workloads is actually Conv2d and the proceeding Bias and ReLU operations.

For this reason we need to also do benchmarks with actual CNNs. For this purpose we use MLPerf Tiny Benchmark from MLCommons. 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.

5 Result

Task	HPC	HPC+DLA
Keyword Spotting	100	30
Visual Wake Words	150	50
Image Classification	120	60
Anomaly Detection*	100	100

Table 5.1 *TinyPerf benchmark results for HPC and HPC with DLA.*

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

6 Conclusions

The goals for this project, were to:

1. Create a virtual counter part of the DLA.
2. Experiment with developing Firmware with a virtual prototype.
3. Develop a use case to demonstrate the capabilities of the DLA and DLA-VP.
4. Benchmark the DLA ASIC.

It's safe to say that we accomplished all these goals.

References

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

[2] 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`.

[3] Yu-Hsin Chen et al. "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).

[4] 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.

[5] 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).

[6] Eric S. Chung et al. "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).

[7] 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).

[8] 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).

[9] Hadi Esmaeilzadeh et al. "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).

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

[11] Google. *Edge TPU*. 2024. URL: https://cloud.google.com/edge-tpu.

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

[13] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[14] 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.

[15] Animesh Jain et al. *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).

[16] 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).

[17] 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).

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

[19] 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).

[20] Wenyan Lu et al. "FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. ISSN: 2378-203X. Feb. 2017, pp. 553–564. DOI: 10.1109/HPCA.2017.29. URL: https://ieeexplore.ieee.org/document/7920855 (visited on 10/21/2024).

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

[22] *Neurocube: a programmable digital neuromorphic architecture with high-density 3D memory: ACM SIGARCH Computer Architecture News: Vol 44, No 3*. URL: https://dl.acm.org/doi/abs/10.1145/3007787.3001178 (visited on 10/21/2024).

[23] 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.

[24] 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).

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

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

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

[28] 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).

[29] 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.

[30] Vivienne Sze et al. "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).

[31] 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).

[32] 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. Something extra

Program 1 *Example call to DLA highlevel API*

```
void dla_conv2d(const int8_t *input_data,
                const int8_t *kernel_data,
                int8_t *output,
                uintptr_t input_channels,
                uintptr_t input_height,
                uintptr_t input_width,
                const char *input_order,
                uintptr_t kernel_amount,
                uintptr_t kernel_channels,
                uintptr_t kernel_height,
                uintptr_t kernel_width,
                const char *kernel_order,
                uint32_t pad_top,
                uint32_t pad_right,
                uint32_t pad_left,
                uint32_t pad_bottom,
                int32_t pad_value,
                uint32_t stride_x,
                uint32_t stride_y,
                uint32_t mac_clip,
                uint32_t pp_clip);
```

Program 2 *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_9(int8_t* headsail_9_i0, int8_t*
    ↪ headsail_9_i1, int* headsail_9_i2, int* out0) {
  float tvmgen_default_headsail_main_9_const_1[1] = {1.000000, };
  int tvmgen_default_headsail_main_9_const_0[1] = {0, };
  int* buf_0 = (int*)malloc(32768);

  conv2d_bias(headsail_9_i0, headsail_9_i1, headsail_9_i2, buf_0,
     ↪ tvmgen_default_headsail_main_9_const_0,
```

```
    ↪ tvmgen_default_headsail_main_9_const_1, 16, 32, 32, "HWC",
    ↪ 32, 16, 1, 1, "HWCK", 32, 0, 0, 0, 0, 0, 2, 2, 0, 0);
  memcpy(out0, buf_0, 32768);
  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.