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

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

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 necessary knowledge in topics in Deep Learning that are needed to understand the development of the DLA software.

2.1 Machine Learning and Deep 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 ML is always a model. Model 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. Turns out that some problems are hard to describe mathematically, and therefore finding a good function for modeling a problem might be nearly impossible in some cases. For example going back to the counting people in a picture describe, how do you event tell a computer what a person looks like?

It turns out that instead of carefully building a mathematical model it's easier to discover it. This is where artificial neural networks come in place. Neural networks are behind most bleeding edge advances in fields such as computer vision, image generation, AI chatbots and autonomous decision making systems.

Deep Learning is a subset of Machine Learning which focuses on multilayer perceptrons, neural networks with more than one hidden layer. The goal of each DNN is to approximate a particular function. In mathematics a function describes the relation between a domain A and codomain B where

$$f \subseteq A \times B, \tag{2.1}$$

meaning that for every element in the domain A there is exactly one corresponding element in codomain B.

Intuitively domain A can be viewed as the input of a multilayer perceptron and codomain B as the output of the network. For a binary classifier codomain B would

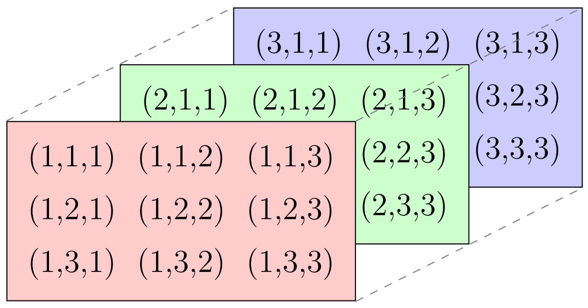

Figure 2.1 *RGB array*

be defined as

$$B = \{0, 1\}, \tag{2.2}$$

and for 10 class classifier

$$B = \{0, 1, 2..., 9\}. \tag{2.3}$$

The domain of the network depends on the problem statement. For example for a image classifier the input can be viewed as a 3 dimensional array, 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 purpose of the neural networks is map the values of array to the codomain in such a manner in which useful information can be acquired from the mapping. Now that we know the domain and the codomain of our DNN we need to discover a function that represents the relation between them.

2.1.1 Fully connected layer

Fully connected layer or dense layer is a

2.1.2 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

$$(f * g)(t) = \int_{-\infty}^{\infty} f(t-x)g(t)dx, \tag{2.4}$$

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

$$(f * g)(i) = \sum_{m} f(i-m)g(m). \tag{2.5}$$

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

$$Conv2D(i,j) = (K \star I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.6}$$

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

$$Conv2D(i,j) = CrossCorrelation(I,K) = (K \star I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.7}$$

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

2.1.3 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

$$y = xA^T + b \tag{2.8}$$

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.

2.1.4 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

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.9}$$

and,

$$tanh(x) = \frac{e^x - e^{-x}}{e^x - e^{-x}}, \tag{2.10}$$

to introduce non-linearity. Both functions suffer from the fact that they are expensive to calculate and exhibit the vanishing gradient problem. [4]

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

$$ReLU(x) = \begin{cases} 0, & \text{for} \leq 0 \\ x, & \text{otherwise.} \end{cases} \tag{2.11}$$

Figure 2.4 shows how ReLU and sigmoid non-linearly scale values close to $x = 0$.

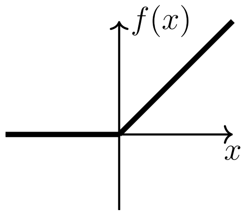

Figure 2.2 *Rectified linear unit (ReLU)*

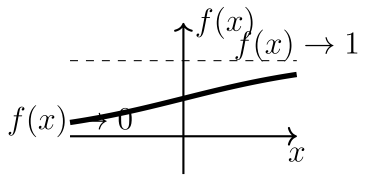

Figure 2.3 *Sigmoid activation*

Figure 2.4 *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 [5] and MobileNet [6].

$$ReLU(Conv2D(I, K) + b) \tag{2.12}$$

2.2 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 2.5 displays the equation 2.12 as a simple graph were data flow is always from one layers output to next layers input.

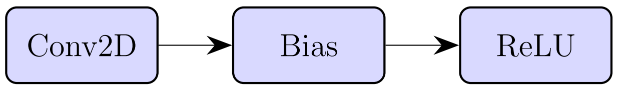

Figure 2.5 *Feed forward relationship between Conv2d, bias and ReLU layers.*

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 [7], a popular framework for training, considers Conv2D, bias and ReLU separate layers and the combination of the a fused layer, where as in Pytorch [8] 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 2.6. Resnet [5] 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 2.7 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]

2.3 Deep-learning accelerators

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.

In recent times there has been a growing need to also run neural networks in

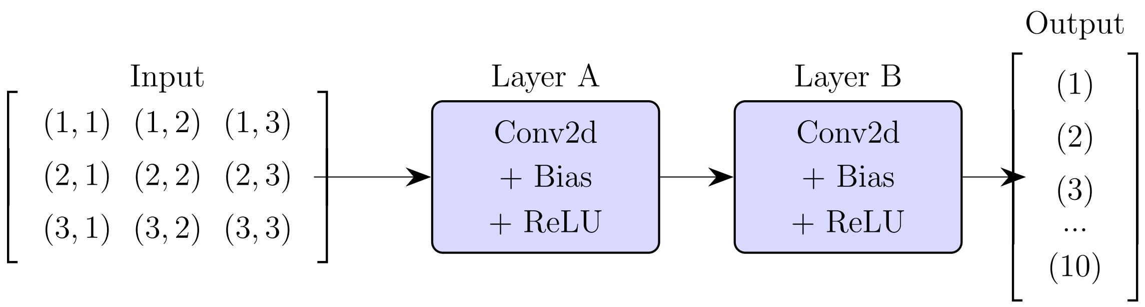

Figure 2.6 *Simple feed forward network with two Conv2D with bias and ReLU activation layers.*

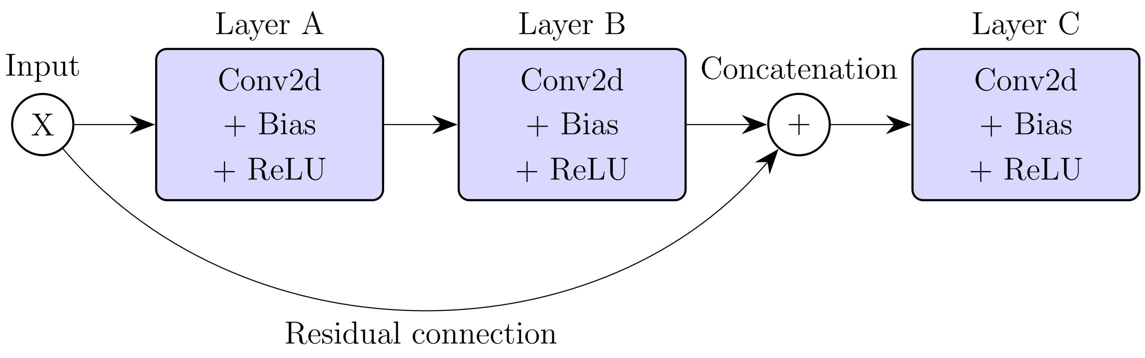

Figure 2.7 *Residual feed forward network with fused layers.*

mobile devices. Traditional GPUs are too power-hungry for these small devices, so a need for more efficient accelerators for these workloads has been born. 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.

2.4 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 paramters in models like GPT, has been growing exponentially, there is are significant performance gains available by reducing the granularity of the paramters. 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 possbile 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.

In the case of Headsail's DLA it supports 8-bit, 4-bit and 2-bit integers, with 4-bit and 2-bit inputs using SIMD operations to linearly increase the amount of calculations per cycle.

2.5 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 compatibiliy.

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.

2.5.1 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.

$$\text{Top-1 Accuracy} = \frac{1}{N}\sum_{i=1}^{N}\delta(\hat{y}_i, \text{argmax}(f(x_i))) \tag{2.13}$$

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.

2.5.2 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 [9]

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{2.14}$$

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} \tag{2.15}$$

$$\mathrm{AUC} = \int_{\mathrm{FPR}=0}^{\mathrm{FPR}=1} \mathrm{TPR}(\mathrm{FPR})\, d(\mathrm{FPR}) \tag{2.16}$$

$$\approx \sum_{i=1}^{n-1} \frac{(\mathrm{FPR}_{i+1} - \mathrm{FPR}_i) \cdot (\mathrm{TPR}_{i+1} + \mathrm{TPR}_i)}{2}, \tag{2.17}$$

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.

2.6 System-on-chip

System-on-chips (SoCs) are a form of heterogenous computing, where multiple different computing elements and electronic systems are integrated into one circuit. 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. Interconnects between SoC components can be run on very high bitrates, greatly reducing latency.

SoC are often used in mobile devices due to tight integration between components enabling small space footprints, in comparison to separately integrated components. ***[Coherency means that different components can works on the same data.]*** 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.

Nowadays SoCs are seen as a solution for the ***[SILICONE APOCALYPSE]***, which describes the lessening increase of advance in computational performance of traditional CPUs due to Moore's Law not applying anymore.

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 [10]. Headsail has two RISCV CPUs, one 32-bit meant for booting up the system called SysCtrl and one 64-bit on called HPC, meant for running the actual applications. Headsail includes a wide variety of different peripherals, one of which is a custom build the Deep Learning accelerator.

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 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.

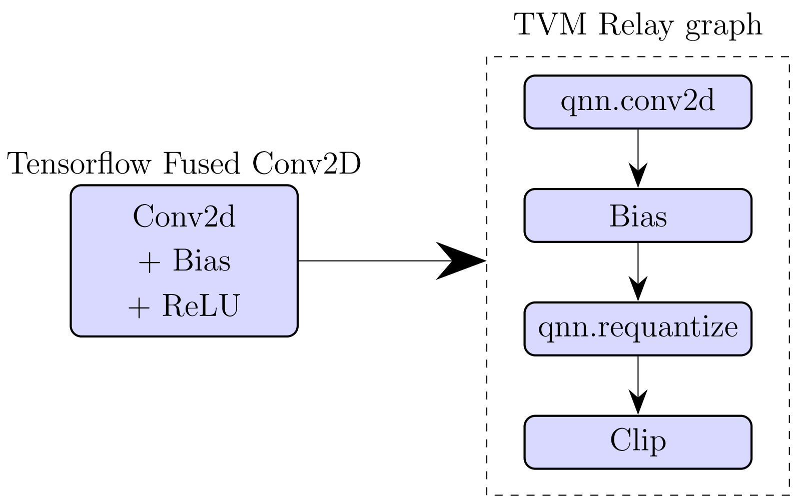

***Figure 3.1** Tensorflow layer to TVM relay graph conversion*

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 accelaratable nodes to suitable accelerators. [3]

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

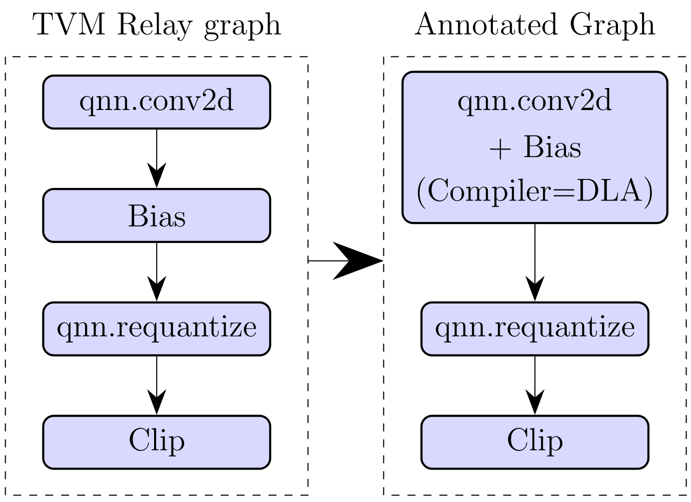

Figure 3.2 *Graph transformation for DLA-VP*

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. [3]

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 consits 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)

3.4.1 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. In TinyPerf benchmark image classification task we use custom ResNet-9, which lacks the downsampling pooling layers to compensate for the low resolution of CIFAR-10 dataset.

3.4.2 Mobilenet

3.4.3 DS-CNN

DS-CNN is a convolutional based on depthwise separable convolutional layers.

3.4.4 FC-AutoEncoder

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 performnace 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.

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.

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.

Headsail-BSP is written in Rust, so to use it with the C based TVM code we also needed to provide a Foreign Function Interface.

4.3 Porting Newlib

Newlib is an implementation of the C standard library meant for use in embedded devices [11]. 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 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*

4.4 DLA Software Architecture

The figure 4.2 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 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.

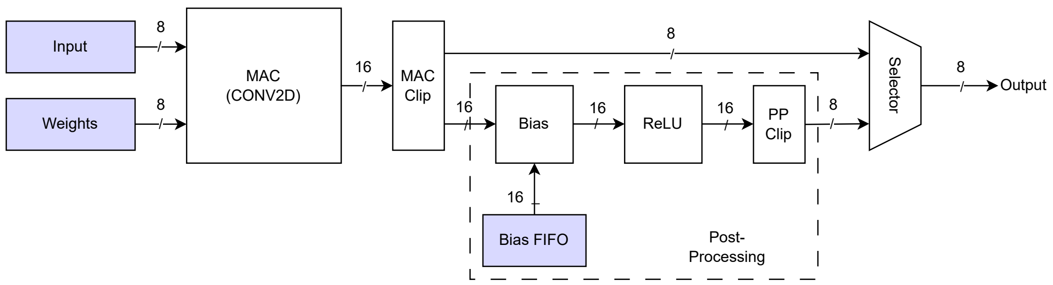

Figure 4.1 *Architecture of accelerated DLA flow in Headsail with TVM runtime and a Pytorch model*

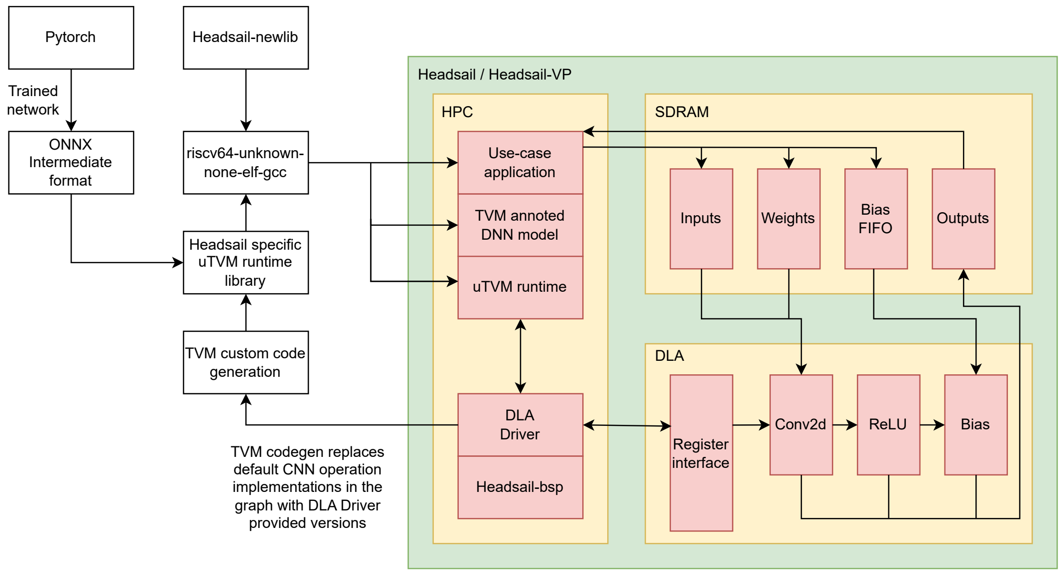

Figure 4.2 *Architecture of accelerated DLA flow in Headsail with TVM runtime and a Pytorch model*

4.5 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] 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.

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

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

[6] 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`.

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

[8] 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`.

[9] 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).

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

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

APPENDIX A. Something extra

Program 1 *Example call to DLA highlevel API*

```c
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.*

```c
#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.