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.

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

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. 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 apporoach to have software support ready before the chip had been manufactured.

2 Background

2.1 Machine Learning and Deep Learning

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 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 that 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

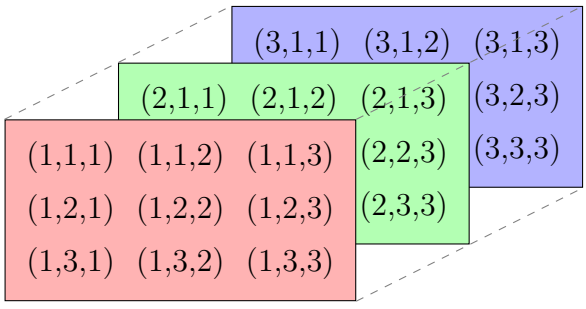

Figure 2.1 *RGB array*

function that represents the relation between them.

2.1.1 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.

2.1.2 Convolution

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

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

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

Rectified linear unit is a common activation function in DNNs defined as

$$ReLU(x) = \begin{cases} 0, & \text{for} \leq 0 \\ x, & \text{otherwise} \end{cases} \tag{2.9}$$

In DNNs, activation functions are used to determined if a particular input is interesting in terms of decisions making. Main benefit of ReLU in comparison to other activation functions, like Sigmoid is that ReLU is computationally light operation.

$$ReLU(Conv2D(I, K) + b) \tag{2.10}$$

2.2 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 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.

2.3 Headsail

Headsail is the third Soc build by the SocHub research group (Rautakoura et al. 2022). 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.

2.3.1 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.

2.4 TVM

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

TVM 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.

During model compilation TVM is able to optimize the graph and allocate acceleratable nodes to suitable accelerators. (Chen et al. 2018)

2.4.1 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 codegeneration 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. (Chen et al. 2018)

2.5 Quantization

When training DNN models with high level tools like Pytorch, models are often build to use floating point operations, since the offer more granularity then intergers. In recent year 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 calculation 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.

3 Methodology

3.1 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.

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 Implementation

4.1 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.

4.2 Porting Newlib

Newlib is an implementation of the C standard library meant for use in embedded devices Red Hat and Contributors 2023. 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. Bennett 2010

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*

4.3 DLA Software Architecture

The figure 4.1 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 build with the Headsail custom code generation option.

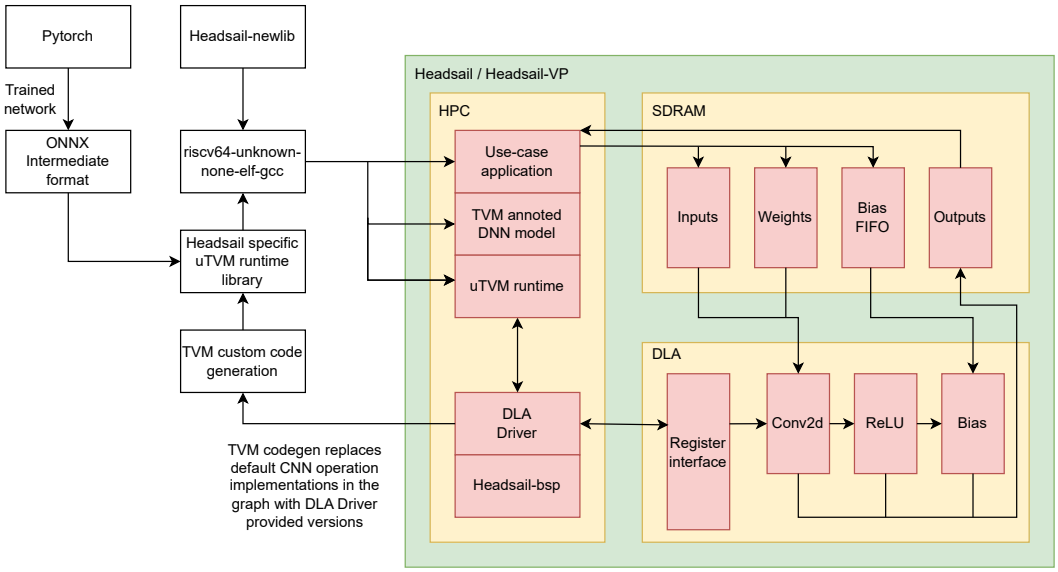

Figure 4.1 *Architecture of accelerated DLA flow in Headsail with TVM runtime and a Pytorch model*

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.

4.4 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, which is a benchmarking suite for benchmarking ML inference in low power targets, like MCUs with Deep Learning Accelerators. (Banbury et al. 2021)

TinyPerf consits of four benchmarks meant to target different use cases shown in table 4.2. 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.

Table 4.2 *Tiny Performance Benchmarks, from (Banbury et al. 2021)*

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)

5 Conclusions

©@

References

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

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

Chen, Tianqi et al. (2018). "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, pp. 579–594. ISBN: 9781931971478.

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

Red Hat, Inc. and Various Contributors (2023). *Newlib: A C Library*. https://sourceware.org/newlib/.

APPENDIX A. Something extra

Appendices are purely optional. All appendices must be referred to in the body text

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.