DEPARTMENT OF COMPUTER SCIENCE

Embedded RISC-V Inference Library & Optimization Pipeline

Supervised by Prof. Simon McIntosh-Smith
In collaboration with Codasip Labs

Thomas Hepworth

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

May 4, 2023

Abstract

The use of Machine Learning (ML) is growing rapidly in the field of embedded systems. At the same time, RISC-V has emerged as an alternative to proprietary ISAs and embedded RISC-V cores are increasingly being used to deploy ML models across a range of industries such as healthcare, transportation, and consumer electronics. With this growth comes the demand for a software ecosystem supporting the efficient optimization of ML models, especially Deep Neural Networks (DNNs). A mature software ecosystem enables this for other ISAs, however many industry standard tools fail to properly support RISC-V. This presents a bottleneck for the adoption of RISC-V for embedded ML. To solve this obstacle I present a minimal version of a complete DNN optimization pipeline, designed to integrate with existing ML frameworks and tools whilst offering support for RISC-V. The pipeline is comprised of a custom graph compiler that quantizes, optimizes, and generates an implementation of an input ML model, as well as a bespoke inference library providing optimized integer-only implementations of common DNN operators. Tools for evaluating the performance of an optimized model are also provided. I demonstrate that my pipeline is able to successfully quantize and deploy a variety of ML models to a RISC-V core with minimal precision loss. I also demonstrate an assessment of the optimizations applied by my pipeline. As well as this I provide a brief survey of the underlying algorithms and equations used to implement DNN quantization tools and inference libraries. I differentiate my implementation of DNN operators by providing variants hand-vectorized using the RISC-V vector intrinsics API. The inference library is designed to easily be made compliant with MISRA C programming safety guidelines.

Dedication and Acknowledgements

I'd like to thank Peter Robertson and the rest of the team at Codasip for offering me the opportunity to undertake this project, as well as Prof. Simon McIntosh-Smith for his supervision and academic guidance.

Thank you to Kerstin Eder, my personal tutor, for her advice and support throughout the four years of my degree.

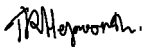
Thanks to Jules Michalski and Tom Broadgate for keeping me company in the library every day.

Thank you to Charles Khoury for his moral support when I was struggling with quantization arithmetic, as well as his donations of strong coffee.

Thanks to Lei Mao for answering some of my questions under his DNN optimization blog posts.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.



Thomas Hepworth, May 4, 2023

Contents

List of Figures

List of Tables

Ethics Statement

This project did not require ethical review, as determined by my supervisor, Prof. Simon McIntosh-Smith

Supporting Technologies

- I used the C programming language and GCC compiler to develop my inference library https://gcc.gnu.org/
- I used the RISC-V GNU toolchain https://github.com/riscv-collab/riscv-gnu-toolchain to compile for RISC-V and test models in the included Spike simulator.
- I used the Python programming language to develop my pipeline scripts and test models https://www.python.org/
- I used the Bash programming language to develop a command line interface to my scripts https://www.gnu.org/software/bash/
- I used the PyTorch machine learning framework https://pytorch.org/ to design and train test models
- I used the ONNX exchange format as input to my pipeline https://onnx.ai/
- I used the SiFive Freedom E SDK https://github.com/sifive/freedom-e-sdk to build models for test hardware and collect performance metrics
- I used the SiFive HiFive1 Rev B RISC-V development board for hardware testing https://www.sifive.com/boards/hifive1-rev-b
- I used the Make https://www.gnu.org/software/make/ and CMake https://cmake.org/ build systems to compile various C codes

Notation and Acronyms

AI	:	Artificial Intelligence
API	:	Application Programming Interface
ASIC	:	Application Specific Integrated Circuit
CNN	:	Convolutional Neural Network
DNN	:	Deep Neural Network
FPGA	:	Field Programmable Gate Array
GeMM	:	Generalised Matrix Multiplication
ISA	:	Instruction Set Architecture
ML	:	Machine Learning
ONNX	:	Open Neural Network Exchange Format
ReLU(6)	:	Rectified Linear Unit
RVV	:	RISC-V Vector

Chapter 1

Introduction

In recent years Artificial Intelligence and Machine Learning have developed considerably, with Deep Neural Network (DNN) architectures often significantly outperforming traditional Machine Learning (ML) methods. Developments have continued to accelerate and the impact of Artificial Intelligence (AI) and ML is felt across a wide range of industries and research areas, supporting innovation in established fields such as medicine, automotive industries, and defense, as well as in emerging technologies such as Internet of Things (IoT) devices. The computational resources required for the deployment of high-accuracy DNN models are a significant bottleneck for the growth of applied ML in industries such as IoT and embedded systems, where large amounts of memory, dedicated hardware, and considerable energy resources are not always available. The growth of the RISC-V ISA is also accelerating, especially in the area of embedded systems. As this continues, so will the demand for tooling to enable the efficient development of AI and ML applications for RISC-V hardware.

Deploying a DNN onto an embedded device comes with challenges. Many embedded systems operate within tight performance and power constraints, meaning a network needs to be optimized for a target's capabilities. Often these optimizations include modifications to the network topology, such as operation fusion, as well as techniques to accelerate execution and compress model size such as quantization. Often, embedded software must be compliant with strict programming standards to ensure safety and security. Tools exist for the deployment of DNNs onto embedded targets, but the RISC-V market is poorly served. Existing libraries either do not meet requirements, focus on large cloud-scale targets, or must be ported to RISC-V, often with the cost of severe feature limitations. I present a lightweight, simple-to-use inference library and optimization pipeline built for RISC-V, which will aid the rapid development and deployment of embedded ML applications.

My project delivers a prototype version of a complete optimization pipeline, consisting of several core components:

- An inference library providing core DNN operations such as GeMM, convolution, and activation functions, implemented using exclusively integer arithmetic for efficient performance on cores without hardware floating point support.
- A graph optimizer, taking a network in the popular Open Neural Network Exchange (ONNX) format, applying graph optimizations, quantization, and memory optimizations, and exporting a C model of the network implemented with the inference library.
- Tools for comparing the network performance in terms of both accuracy and computational performance.

These components are packaged as a single tool with a simple command line interface, allowing a user to convert a trained DNN model from a graph representation, specifically the widely-adopted Open Neural Network Exchange (ONNX) format, into a compressed, quantized, integer-only implementation in C. The pipeline output can easily be compiled into a library and linked to an existing project.

Chapter 2

Background

2.1 The RISC-V ISA

RISC-V is an open-source Instruction Set Architecture (ISA) originally developed by the University of California, Berkeley, and now maintained by RISC-V international, an open consortium of over 3000 companies, government agencies, and academic institutions. RISC-V is a modular ISA, consisting of a base that defines instruction encoding and contains a small number of instructions, and an arbitrary selection of extensions that define groups of instructions supported by the processor. The base ISA and extensions implemented by a processor can be represented by a mnemonic, for example, a RV32IMAC processor implements the RV32I the 32-bit base integer instruction set, as well as the M extension for integer multiplication, A extension for atomic instructions, and C extension for compressed instructions.

2.2 The ONNX Exchange Format

The Open Neural Network Exchange (ONNX) format[1], an open source originally developed by Facebook and Microsoft. ONNX provides a graph representation of a deep-learning model and is designed to enable interoperability between various machine-learning frameworks and devices. The format defines an extensive set of operators, which can be used to represent a variety of common DNN archetypes such as Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and transformers.

ONNX is the most widely supported exchange format and supports most common ML frameworks including PyTorch[2], TensorFlow[3], and Keras among many others. This makes ONNX an obvious format for a user-facing tool to support since it is likely to be compatible with a user's existing workflow. ONNX graphs are easy to work with within code as ONNX is based on the Protobuf[4] format. Protobuf is an efficient, language-neutral mechanism for serializing data, and a Protobuf file can be easily loaded and manipulated from a variety of languages and tools.

Other exchange formats are available, such as the Neural Network Exchange Format (NNEF)[5] which seeks to achieve a more stable specification than ONNX. NNEF is an extremely similar concept to ONNX, aiming to enable greater interoperability between tools and frameworks. The main difference is that NNEF specifically targets EdgeAI applications and promises greater hardware stability, which is enticing for my project. However, current NNEF implementations are closed-source and not as widely adopted as ONNX, making it a less favorable exchange format to prioritize. ONNX's flat-graph description is simpler than NNEF's more abstract model description, which allows for compounded operations. It is possible to convert between NNEF and ONNX, but ONNX's simpler representation, more direct conversion between the exported graph and defined operators, and wide adoption make it favorable over NNEF to support.

The ONNX ecosystem also provides ONNX-Runtime (ORT)[6], a tool for enabling fast, automatic deployment of an ONNX model. However, despite containing optimization features and a C/C++ API, ORT is generally favored for deployments of larger ML models, often on server-scale infrastructure. Some examples exist of ORT being used to deploy models for IoT applications on relatively low-powered hardware such as the Raspberry Pi, but the implementation is not particularly well-optimized or suited to

low-powered embedded systems, requiring a Linux operating system the hardware features of a general-purpose CPU to achieve acceptable performance. Using an ARM processor, the Raspberry Pi also benefits from far superior software and compiler support, achieving better performance than might be seen on a comparable RISC-V chip. Whilst it could be ported to RISC-V, ORT is not currently a viable solution for running a model on a comparatively resource-limited embedded target.

2.3 Existing Optimization Tooling

The hardware landscape of embedded systems is currently dominated by ARM architectures, with Arm currently holding a 90%+ market share in mobile processors[7]. For this reason, several tools exist to support the development of DNN models for embedded ARM targets. However, a significant threat to ARM's market dominance is the emergence of the RISC-V ISA, with analysts predicting that the number of RISC-V-based SoCs will grow by 73.6% by 2027[8]. However, a roadblock in the adoption of RISC-V in the embedded ML space is the lack of software support.

Two of the most established tools for deploying DNNs on embedded targets are Apache's Tensor-Virtual-Machine (TVM) and TensorFlow-Lite[9](TFLite). Both tools offer a means of converting a DNN model from a high-level format to an optimized binary or library ready to deploy on a mobile device or micro-controller. Currently, neither tool currently has officially maintained support for RISC-V, and while it is possible to port both tools to RISC-V this is a troublesome process involving many source file changes and usually comes with a significant performance cost without extensive manual optimization[10].

TFLite adds embedded-focused quantization, compression, and a portable format to the popular TensorFlow library, which is used to design and train DNNs. TFLite is designed for mobile and embedded devices, and TFLite for Micro-controllers (TFLiteMCU) ports this functionality to a number of micro-controllers including Arm Cortex-M Series. Both TFLite and TFLiteMCU aim to enable on-device machine learning by optimizing ML models to run with limited resources. TFLiteMCU requires no operating system support and is designed to operate within only a few kilobytes of memory. Both tools have a convenient workflow where a trained TensorFlow model is first exported to the optimized ".tflite" format, the model parameters are converted to C byte arrays, and then the inference is implemented via a C/C++ API. Only a subset of TensorFlow operations are supported by the TFLite API.

TVM is more general purpose, taking input from a number of high-level libraries and exchange formats, and providing a range of graph-level optimizations such as operation fusion, as well as operator-level optimizations such as quantization and cache tuning. TVM can target general-purpose and server processors, and the MicroTVM variant extends the graph compiler to target microcontrollers. ApacheTVM ingests a model and converts it to an intermediate representation before iteratively transforming each operator into an optimized yet functionally equivalent version. Attempts have been made to integrate RISC-V into TVM [11]. However, currently, TVM's RISC-V integration is incomplete, being based on translations of optimizations designed for other architectures, because of this the generated code is not optimal for RISC-V targets.

TVM Bring Your Own Codegen (BYOC) offers the ability to combine TVM's graph compilation and optimization tools with existing highly optimized implementations of DNN operators. BYOC allows the developer of an ASIC accelerator to use TVM to deploy models using custom versions of operators, allowing TVM to handle graph optimization and any operators unsupported by the custom ASIC software[12]. The number of ML accelerators developed with RISC-V is increasing[13], and in the future, a suite of operators optimized for RISC-V targets could become a back-end for TVM's advanced compilation.

2.4 Graph Optimization

The ONNX ecosystem contains ONNX Optimizer[14], a tool for optimizing an exported graph. The optimizations performed by ONNX Optimizer are generally limited to simplifications of the graph topology, and converting over-complicated operations to concise and simple variants. ONNX optimizer only transforms an ONNX model to a simplified ONNX model and thus is limited to producing operators defined in the ONNX specification. It can fuse operators such as MatMul and Add into a single GeMM operation, but would not be able to fuse, for example, a GeMM and ReLU operation, since a fused GeMM and ReLU operation does not currently exist in the ONNX specification. In my testing, a well-maintained

ML framework such as PyTorch generally exports an ONNX model from which no further optimizations are made by ONNX optimizer. It is not clear whether PyTorch uses ONNX optimizer internally.

Certain lower-level graph optimizations can further increase the efficiency of data transfer, caching, model compression, and inference time [15]. For example, to improve caching in an inference library where tensors are stored in a row-major format, it may be beneficial to "pre-transpose" tensor arguments to certain operators to avoid unnecessary data movement during inference. In other cases, it may be advantageous to take advantage of mathematical properties such as commutativity, associativity, and distributivity to make algebraic simplifications to the graph[15]. The advantages of performing these optimizations are not necessarily exposed to an exchange-format level tool such as ONNX Optimizer, since the optimizations applied may only be beneficial once tuned for the capabilities of the underlying inference library and/or hardware target. TensorFlow offers Grappler[16] for applying this type of optimization.

2.5 Safety Standards in Embedded Software

As embedded ML grows as a field, so will the demand for secure and reliable embedded software. This is especially true in industrial, automotive, or defense applications - where security and reliability are critical attributes of most software [17]. Embedded developers often follow code safety specifications such as MISRA C[18, 19] in order to help reduce reliability issues and ensure code quality is of a high standard. Any software tool intended to be used for safety-critical applications should also conform to these standards, or they will not be appropriate for use as a component of any otherwise conforming code base.

2.6 DNN Quantization

2.6.1 The need for quantization in embedded ML for RISC-V

Defining characteristics of embedded systems include limited computational resources. DNNs generally have large compute requirements [20] which often conflicts with the relatively low memory and cache size of embedded processors. A number of vendor-specific hardware accelerators have been created, such as NVIDIA's Jetson[21] and Intel TigerLake, and with these come specialized software ecosystems for optimization. The availability of hardware accelerators is far lower for RISC-V architectures, mostly being constrained to academia, using non-standard instructions, or only implemented on an FPGA [17]. One notable exception is the Kendryte K210, currently one of the only widely available (to consumers) RISC-V SoCs which contains a neural processing unit. Where hardware accelerators have been developed for RISC-V, the supporting software ecosystem is still underdeveloped compared to, for example, ARM architectures.

Of the RISC-V chips that have been fabricated and are currently available on development boards for sale or pre-order, the majority implement the RV32IMAC base ISA and extensions[22]. On cores implementing RV32/64IMA(C), floating point operations are software floating point only, making the large number of floating point operations used in full-precision DNNs expensive. Inference time is of significant concern in embedded systems, with many embedded systems operating under strict timing constraints, especially in industries such as aviation [23]. In order to both compress the network so that parameters fit in limited memory and to convert floating point operations to integer ones, quantization and integer-only versions of DNN operators can be used.

On hardware supporting vectorization, quantization also enables tighter packing of values into vector registers. The RISC-V vector extension has only recently been ratified[24], however, vector units supporting the proposed specification are already beginning to be developed [25]. These will have a significant impact on the performance and energy consumption of DNNs deployed to RISC-V targets, increasing the efficiency and effectiveness of quantization as more hardware accelerators are developed. The upcoming packed SIMD ("P") extension to the RISC-V ISA may also offer performance improvements on quantized datatypes.

2.6.2 Quantization strategies

A DNN tensor may have a high variance between the values of neighboring elements, and subtle changes in single elements may result in largely different network outputs. For this reason, a quantization strategy

that discreetly maps each individual element from high to low precision is needed. Other strategies such as image quantization techniques which involve methods such as averaging across neighboring elements would be inappropriate for preserving precision in a DNN.

A common method for achieving this involves defining an affine map (a combination of a linear transformation and a translation) from the range of a given tensor to an integer range. The equations and derivations in the following sections were first presented in 2017, and are at the heart of many prominent low-precision BLAS tools such as Google's Gemmlowp[26] (which powers parts of TFLite), NVIDIA's TensorRT [27] and the quantization in Caffe2. Using a mathematically rigorous quantization scheme helps maintain correspondence between training a DNN in floating point, and quantized (integer) inference [26]. I am assuming an input datatype of FLOAT32 as this is the default in PyTorch, and an output type of INT8/UINT8 which provides around 4x network compression without a large precision loss for most input ranges, although the equations described here function for any arbitrary input/output range.

Non-uniform quantization strategies are also possible and can preserve more precision than a linear map. However, the implementation of efficient quantized operators is complicated by non-linear mapping, and performance can suffer[28]. Non-uniform quantization could still be employed to effectively compress a model while preserving more precision.

Dozens of network quantization strategies exist[28], and they can be combined in numerous permutations. However, for a tool compressing and quantizing an arbitrary network, I have focused on typically simpler strategies requiring only minimal adjustment in the network design phase. The result is a quantization system that achieves good compression and good precision on a large number of DNN architectures. For a specific DNN architecture and specific applications, other strategies may provide a better trade-off between inference latency, network size, and network performance. However, without fine-tuning, extreme quantization methods such as binary quantization can result in severe network accuracy degradation[28]. In an automated pipeline, this would add significant complication for a user, and likely cause conversion from a floating point to an integer model to be far less reliable. Options for using these alternatives could always be built into the tool at a later stage.

2.6.2.1 Affine (Asymmetric) Quantization Mapping

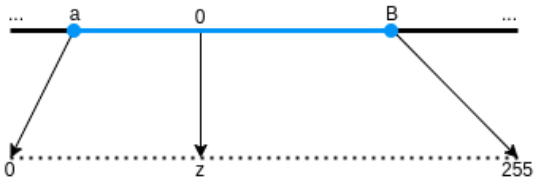


Figure 2.1: Asymmetric Quantization Mapping

Our goal is to map a range of FLOAT32 values in the range $[\alpha, \beta]$ to INT8 or UINT8 values in the range $[-128, 127]$ or $[0, 255]$ respectively. In general, this can be considered an affine map, with a linear transformation scaling the range $[\alpha, \beta]$ to some quantization range $[\alpha_q, \beta_q]$, followed by a translation to ensure that quantizing the value α gives α_q and quantizing the value β gives β_q.

One domain-specific requirement is that the real value 0 should be exactly representable in the quantization scheme. This is important since many common network operators such as pooling may require zero-padding around the boundaries of a tensor, and so an ability to represent zero can have a large impact on the precision of a quantized network. As well as this, certain optimizations exploit network sparsity. It may not be possible to apply all of these optimizations without an exact representation of zero. In practice, this constraint means that the range $[\alpha, \beta]$ must always contain zero, which can be enforced by shifting α and/or β.

An affine map composes a linear function with a translation in order to map one coordinate system to another, shifting the origin whilst maintaining properties such as the ratio between different points in the output space. We can describe the quantization/de-quantization process with the equation $x_{real} = c(x_q + d)$ for some constants c and d[29].

It must be ensured that α maps to α_q and β maps to β_q, providing the linear system:

$$\alpha = c(\alpha_q + d)$$
$$\beta = c(\beta_q + d)$$

Which can be solved to obtain:

$$c = \frac{\beta - \alpha}{\beta_q - \alpha_q}$$
$$d = \frac{\alpha\beta_q - \beta\alpha_q}{\beta - \alpha}$$

From rearranging $x_{real} = c(x_q + d)$ and the requirement that zero be exactly representable, we know that when x_{real} is zero, x_q must be the zero point in the quantized range. To find the point in the integer range corresponding to zero, we must choose d to satisfy $0_{real} = c(x_q + d)$, from which we can obtain $x_q = round(\frac{1}{c}0 - d)$ by rearranging and rounding to convert to an integer, simplifying this we obtain $x_q = -round(d)$.

Renaming c to s (scale) and d to $-z$ (zero point) provides the quantization scheme:

$$x_{real} = s(x_q - z)$$

where

$$s = \frac{\beta - \alpha}{\beta_q - \alpha_q}$$
$$z = round(\frac{\alpha\beta_q - \beta\alpha_q}{\beta - \alpha})$$

where s is our quantization scale represented as a floating point number, and z is our zero point (the quantized integer value mapping to the real value 0). To scale to an integer data type, notice that the range $[\alpha_q, \beta_q]$ will simply be the range of the integer type, and $\beta_q - \alpha_q$ is just the number of integer values in the range of the data type. For an integer data type of bit-width b, we can compute the scale as:

$$s = \frac{\beta - \alpha}{2^b - 1}$$

2.6.2.2 Value clipping

In practice, it is not always effective to use the actual minimum and maximum of a tensor to define the quantization range. Larger ranges reduce the precision of the mapping and often many weights are close to zero, meaning that defining the quantization scale based on outliers far from zero can cause more precision loss than necessary. It is also possible that the network may produce a value beyond the range seen in training, which would then be quantized to outside the range $[\alpha_q, \beta_q]$, potentially causing overflows.

For this reason, the values of the input tensor are often clipped to a certain range before quantizing. There are many techniques for choosing the clipping range such as histogram-based quantization (section 2.6.4), which can be used to reduce the impact of outliers on the precision level of the quantization mapping. This gives the final formula for quantizing a real value clipped to the range $[\alpha, \beta]$ to an integer in the range $[\alpha_q, \beta_q]$:

$$x_q = \begin{cases} \frac{1}{s}\alpha + z, & \text{if } x_{real} < \alpha \\ \frac{1}{s}x_{real} + z, & \text{if } \alpha \leq x_{real} \leq \beta \\ \frac{1}{s}\beta + z, & \text{if } \beta < x_{real} \end{cases}$$

This can be implemented by quantizing to a larger type than the final intended type and then clipping to the chosen range before storing it in memory. For example, we might first quantize to the integer range 0 to 255 but store the result in a 32-bit signed integer type, clip any overflows, and then cast to an unsigned 8-bit integer before writing into a destination buffer.

2.6.2.3 Scale (Symmetric) Quantization Mapping

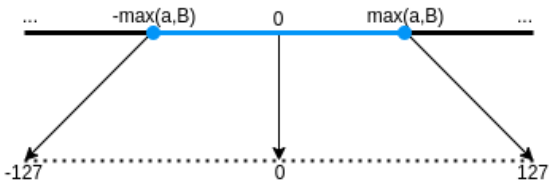


Figure 2.2: Symmetric Quantization Mapping

Scale (symmetric) quantization mapping can be used to quantize FLOAT32 values to INT8, it is a special case of affine (asymmetric) quantization mapping and does not include the zero point. Instead, the real value 0 is mapped to the integer value 0. This carries the risk of greater precision loss if the data to be quantized is not distributed symmetrically around a mean of 0, however, it can sometimes allow for a significant optimization in network operations, exploiting sparsity by allowing the skipping of many integer arithmetic operations if, for example, a multiplication reduction contains a zero.

In practice, weight tensors tend to be fairly symmetrically distributed and centered around zero, which means they can be symmetrically quantized to a suitable integer range with relatively low precision loss. An example of this taken from one of my test models is shown in figure 2.3.

Some implementations of quantized tensor operations limit the quantization range to smaller than the full range of the destination datatype [26]. This allows for optimizations when accumulating in, for example, GeMM operations, as the maximum value of the sum of products of elements can be stored in a smaller datatype. This yields a performance improvement on certain hardware supporting vectorization, as more accumulators can be packed into the same vector register.

In TFLite bias vectors are usually quantized using a symmetric mapping with a destination type of INT32. The large destination type, and therefore higher precision of bias vectors boosts network accuracy [26]. Since bias vectors are added to every activation, imprecision can result in general network bias. The extra memory usage is justified since bias vectors tend to represent only a small proportion of the parameters of a network[26].

2.6.3 Dynamic vs Static Quantization, Activation Simulation

For an effective mapping of floating point values to integer values, knowledge of the range to be quantized is required. The weights of a DNN will not change after training, so their range is known and fixed. However, the network activations depend on the input, and so cannot be known exactly before inference. There are three main approaches to obtaining the floating point range.

2.6.3.1 Dynamic Quantization

Calculating activation ranges during inference is known as dynamic quantization. In dynamic quantization, weights are usually already quantized at inference, but for activations, a floating point input tensor is scanned at inference to find its minimum and maximum, these values are used to define the quantization parameters (scale and zero point) and then the network operation is carried out using integer-only arithmetic as much as possible. However, because the network can not infer the activation output ranges

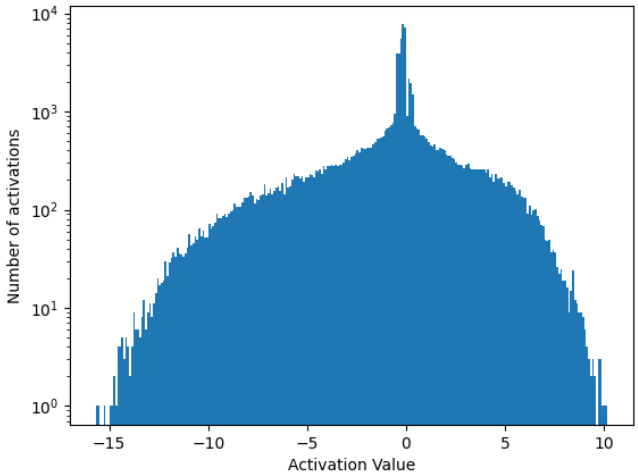


Figure 2.3: Example histogram of the distribution of values in the weight parameter of one layer of the Iris-FC test model

at each layer, a floating point result is calculated. This must then be re-quantized with new quantization parameters before it can be input into the next layer.

Dynamic quantization often has some performance benefits because of the large proportion of integer operations. In networks with large amounts of trained parameters dynamic quantization can help compress the network. However, because of the requirement to re-quantize an activation tensor after every operation, dynamic quantization is most effective on hardware with good support for floating point operations. This makes it unsuitable for many current mass-market RISC-V microprocessors, a large number of which only support software floating point.

Dynamic quantization is often used as a first step to neural network optimization as it has no data requirements and can be easily implemented on large models since it is supported by major libraries such as PyTorch and TensorFlow.

2.6.3.2 Static Quantization

In static quantization, the quantization parameters for activation tensors are decided before inference. They are often learned in training by recording activation statistics as test data is fed to the network or decided post-training using simulated activations on representative data, and various statistical methods discussed in section 2.6.4.

Static quantization results in less-precise results than dynamic quantization because the dynamic range of an activation during inference may be smaller than the range used to decide quantization parameters, wasting precision [28]. In some cases, the activation range may be larger than the quantization range causing value clipping. Despite this, inference time is greatly improved since floating point operations can be entirely removed from inference.

2.6.3.3 Quantization Aware Training

Quantization-aware training introduces quantization and de-quantization layers around every weight and activation tensor in the DNN model, allowing the network to learn using quantized values. Operations performed after a quantization layer can only access the information preserved after quantization, so the network must learn to account for the reduced precision. Quantization layers are non-differentiable, and back-propagating through them must be approximated with techniques such as Straight Through Estimation[30].

The quantization/de-quantization layers are not needed during inference as static quantization can be applied to the final weights and activations of the model. Quantization-aware training has the same

inference latency as static quantization, yet the accuracy loss tends to be smaller due to the improved generality of the network [28].

Novel quantization-aware training techniques exist which further reduce the precision of DNNs for performance and model compression benefits. Binary weight networks with weights constrained to $\{-1, +1\}$ and ternary weight networks with weights constrained to $\{-1, 0, +1\}$ have both been used to compress models by packing weights into few bits and provide a significant performance increase by simplifying accumulations and exploiting sparsity. This is achieved whilst maintaining comparable accuracy to full precision models [31, 32].

2.6.4 Methods for Choosing Quantization Parameters

It follows from the definition of the quantization scheme that the choice of the floating point range $[\alpha, \beta]$ to quantize to an integer range directly determines the level of precision preserved through quantization. Gemmlowp learns this range by taking the exponential moving average of the min/max activation ranges seen during training[26]. A high smoothing constant is used to weight the range decision in favor of more recent values. In post-training quantization, the activation ranges can be recorded by running the full-precision model on representative test data and recording the min/max ranges.

In a convolutional neural network, weights form feature detectors, and activations tend to be high values in input areas matching a feature. In areas of the image outside the feature, weights tend to be small or zero, as these areas should not pass on signals to the next layer. In fully connected layers weights often form a symmetrical distribution around a mean of zero. In both cases, the weight distribution of a network layer can be visualized as a histogram. Histogram-based quantization uses this histogram to attempt to determine more optimal values for the floating point range to be quantized. The term "Histogram-based quantization" can refer to several techniques. In some implementations, the histogram bins at the edges of the weight distribution may be clipped away and the network re-tested, gradually removing range from $[\alpha, \beta]$ until the accuracy of the network falls below a threshold level [33]. In other applications a histogram may be used to choose a clipping threshold in order to minimize the mean squared error between the floating point values and their quantized counterparts [34]. Minimizing loss between an activation tensor and a histogram generally performs better on convolutional networks which are more likely to exhibit asymmetrical distributions of activations.

Convolutional neural network weights often vary significantly in the dynamic range between channels, for this reason, each channel is often quantized separately. This can hugely increase the precision level of the quantized values, at the cost of storing a few more quantization scales and zero points, and more complicated operators. If a model is destined to be quantized, it is generally also good practice to train a DNN making use of regularisation and normalization techniques in order to minimize the range of weights and activations.

2.6.5 Quantized Network Operations

In order to implement DNN operations with integer-only arithmetic, all operations must be converted to use operations on quantized values. In general, this conversion is attained by defining the operation in mathematical terms and substituting every tensor value for its quantized form. Often this involves quantization scale conversion between the input and output tensors, which can be made integer-only using fixed-point arithmetic. In this section, I show the process of deriving a mathematical definition of a quantized operator which can be implemented in the inference library. Again, these equations were first publicized in 2017[26], however, I aim to present the derivations more intuitively than in existing literature to make them easier to implement.

2.6.5.1 Quantized GeMM

The GeMM operation $Y = XW + b$ can be written as

$$Y_{i,j} = \sum_{k=1}^{p} X_{i,k} W_{k,j} + b_j$$

where $Y \in \mathbb{R}^{m \times n}$, $X \in \mathbb{R}^{m \times p}$, $W \in \mathbb{R}^{p \times n}$, $b \in \mathbb{R}^{n}$. Each tensor will be quantized with its own scale and zero point, substituting in the quantization equation (ignoring rounding) for each matrix will give

an integer-only equation:

$$\begin{aligned}
Y_{i,j} &= s_Y(Y^q_{i,k} - z_Y) \\
&= s_b(b_j - z_b) + \sum_{k=1}^{p} s_X(X^q_{i,k} - z_X)s_W(W^q_{k,j} - z_W) \\
&= s_b(b_j - z_b) + s_X s_W \sum_{k=1}^{p}(X^q_{i,k} - z_X)(W^q_{k,j} - z_W) \\
&= s_b(b_j - z_b) + s_X s_W \sum_{k=1}^{p}(X^q_{i,k} - z_X)(W^q_{k,j} - z_W) \\
&= s_b(b_j - z_b) + s_W s_X \left[\left(\sum_{k=1}^{p} X^q_{i,k}W^q_{k,j}\right) - \left(z_W \sum_{k=1}^{p} X^q_{i,k}\right) - \left(z_X \sum_{k=1}^{p} W^q_{k,j}\right) + pz_W z_X\right]
\end{aligned}$$

Where T^q is the quantized representation of a tensor T. In dynamic quantization (2.6.3.1), we can store the real element $Y_{i,j}$ and then re-quantize the result tensor Y in a later stage. In static quantization (2.6.3.2), the quantization parameters s_Y, z_Y would be known when the model is encoded, and to remain fully integer-only just the value $Y^q_{i,j}$ is needed, so the following equation is used:

$$Y^q_{i,j} = z_Y + \frac{s_b}{s_Y}(b_j - z_b) + \frac{s_W s_X}{s_Y}\left[\underbrace{\left(\sum_{k=1}^{p} X^q_{i,k}W^q_{k,j}\right)}_{\text{Term 1}} - \underbrace{\left(z_W \sum_{k=1}^{p} X^q_{i,k}\right)}_{\text{Term 2}} - \underbrace{\left(z_X \sum_{k=1}^{p} W^q_{k,j}\right)}_{\text{Term 3}} + \underbrace{pz_W z_X}_{\text{Term 4}}\right] \quad (2.1)$$

In static quantization, the values z_W, z_X, and the values of the weight tensor W^q are all known before inference. This makes it possible to pre-compute the values of Term 3 and Term 4 to be used during inference.

The implementation of this equation is complicated by several factors. Firstly, the quantization scales s_Y, s_X, s_W are all implemented with floating point values. Thus, the multiplication and division required to re-scale the accumulated values introduce significant computation expense on hardware with only software floating point support (for example, common RV32IMAC cores). This issue can be avoided in static quantization by converting the value $\frac{s_W s_X}{s_Y}$ (known before inference) to some fixed point multiplier M where $M = 2^{-n}M_0$, $n \in \mathbb{Z}^+$, and M_0 is in the range $[0.5, 1]$. This is chosen as the range for M_0 as it guarantees a certain level of accuracy in the fixed point multiplier[35], thus allowing for the floating point operation to be substituted for a fixed point operation consisting of an integer multiplication and bit-shift. Choosing a bias-vector scale such that $s_b = s_W s_X$ allows for the fusion of the bias term into the same accumulation and fixed-point rescaling. Flexibility in the choice of quantization scaling parameter is possible due to the greater quantization range of the bias vectors.

In order to avoid overflows when accumulating Term 1 ... 4 the result of the accumulated multiplications must be stored in a larger typed accumulator. For the common strategy of quantizing weights to INT8 and activations to UINT8 the accumulator must be INT32 or larger. In GeMMlowp, a signed type is chosen to allow the easy addition of signed bias terms to the accumulator[26].

2.6.6 Efficient Quantized Convolution

Other DNN operators can be converted using the same principles as above - substituting real numbers for their quantized representation and replacing floating point operations with fixed points. However, for certain operators such as convolution, it can be faster to express them in terms of highly-optimized GeMM operations.

A common technique for implementing convolution in this way is to first form an im2col patch matrix from the input image and matrix-multiply this with a reshaped version of the convolution kernel [36, 37]. A patch matrix is formed by storing each "window" seen by a filter as it strides over an image as a column in a 2D matrix. If there are multiple input channels, each channel's patch matrix is stacked.

The resulting image can then be multiplied with a re-formatted filter, and a bias can be added in a single GeMM call. The time complexity of convolution remains the same with this technique, but the computational performance is improved by using highly optimized GeMM calls. Im2col convolution demands a large amount of memory to store the constructed patch matrix. An input image of size $O(HWC)$ is transformed to a patch matrix of size $O(HW(K^2)C)$. This has performance implications for embedded devices with limited memory.

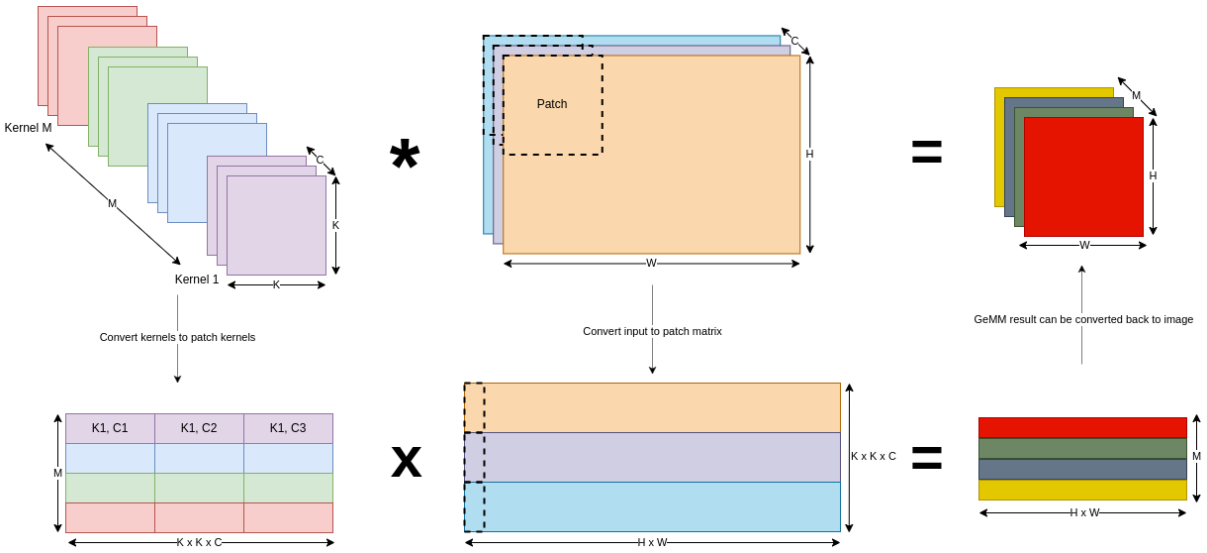


Figure 2.4: Example of Im2col convolution

Im2row convolution works using the same principle as Im2col, but stores the patch matrix row-wise rather than column-wise. Depending on the implementation, Im2row convolution may provide better spatial locality for the elements of a patch matrix in a GeMM call, improving caching. The size of the Im2col/Im2row buffer can be reduced by reusing data between adjacent convolutional windows[37]. Alternative algorithms converting convolution to GeMM requiring far less memory have also been developed. Techniques such as kernel-accumulating perform better than Im2col, with a memory complexity as low as $O(CHW)$[37]. This offers a significant space saving over a large patch matrix.

Chapter 3

Project Execution

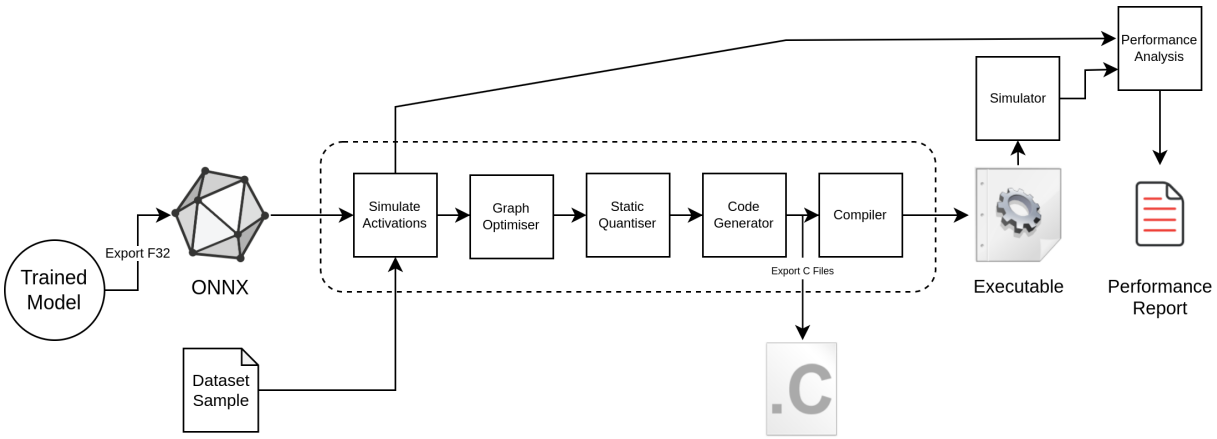


Figure 3.1: Overview of optimization pipeline architecture

3.1 Pipeline Design

For convenience and ease-of-use and I designed the model compilation pipeline to be usable from a single command. This also allows for a model compilation command to easily be added to Makefiles, or included in a CMake build system. My intention was to make integration with existing development and continuous integration workflows as simple as possible. The pipeline is composed of a sequence of Python scripts:

- `simulate.py` takes as arguments an ONNX model and test data. Outputs a file containing the activation statistics for each layer to be used in post-training quantization.
- `export_to_c.py` takes as arguments an ONNX model and a number of flags/options. This script performs quantization, applies certain graph and memory optimizations, and converts the model to C

These scripts can be executed in sequence by running `pipeline.sh`. This is the entry point designed to be the only interface with the user. Flags and options here are passed to the pipeline scripts where appropriate to keep all configuration in one place. The currently implemented command line arguments are:

Required:

- `-m | --model`: Path to the ONNX model to compile.
- `-d | --data`: Path to the test data for simulation. Currently, data is expected as a Pickle file as a list of tuples: `[(<model input>, <target output>)]`.

- `-o | --output`: Path to place the generated C files.

The flags `--input-fmin`, `--input-fmax`, and `--input-dims` are intended to be temporary, but are currently required. Their arguments depend on the model and could be inferred from the input graph.

Optional:

- `-n | --sim-runs`: Number of inferences to run when simulating, default 500
- `-i | --model-input-name`: The name of the ONNX model input node, default "input"
- `--no-shared-mem`: Disable the memory allocation described in section 3.3.3. This can be useful for debugging the inference library as enables each activation to be inspected after inference.
- `--weights-as-i8`: Enables quantizing weights as INT8 (symmetric quantization) and activations as UINT8 (asymmetric quantization).
- `--allow-fusion`: Enables operation fusion during the code generation stage (section 3.3.4.1).
- `--output-target [code | binary]`: Default: `code` . Choose whether to make the pipeline generate C code, or compile code into a static library. Outputting only code may be preferable if the model is to be included in a larger project, or if a code-base is intended to be compiled for multiple targets with support for different RISC-V extensions.

I also provide scripts for evaluating the performance of the generated binary compared to the original model:

- `network_comparator.py` takes as arguments an ONNX model and its compiled RISC-V binary. The output of the model compiled for RISC-V (run in the Spike simulator) is compared with a full-precision version run in ONNX Runtime. This script is used to compare the accuracy between the original and optimized models. The output of the compiled binary is collected with a user-provided script. This may simply read the standard output of the program, but in more complex scenarios it could be easily extended to, for example, read data over a serial connection or interface with a debugger.
- `activation_histograms.py` takes as arguments an ONNX model and test data and generates histograms of every activation tensor. The images generated by this script can be used to inform DNN architecture or compilation decisions.

Example: To run a model through the pipeline a command in the following format is needed:

```
pipeline.sh -m model.onnx -d test_data.pkl -o ./output_dir
```

This will simulate, quantize and optimize the model and generate a self-contained `model.c` and `model.h` inside `output_dir`. These files can then be linked to an existing C project.

3.2 Simulation Stage

As explained in section 2.6.3.2, in order for effective post-training quantization the network must be simulated on representative training data to record the values of the activations of each layer to be quantized. Since the model is being compiled on the developer's general-purpose hardware, ONNX Runtime (ORT) can be used to simulate these activations with reasonable efficiency. First, the serialized ONNX model is extended such that every layer is considered a network output, this will cause ORT to return the values of each activation after each inference. As inference is run on the representative test data set, statistics are recorded on every activation layer. The most important values to record are the layer min/max, but a number of options are available depending on the strategy for choosing quantization ranges. The strategies I have implemented are:

- Capture All: Record the absolute min/max over all activations. This results in the largest quantization range and therefore lower precision.
- Average min/max over all activations: This strategy generally has the effect of clipping significant outliers whilst capturing most of the significant range of the layer's activations.
- Simple histogram-based-quantization: Limit min/max to be a maximum of n standard deviation(s) away from the mean over all activations, effectively clipping off outlying histogram bins. Depending

on the value of n this method tends to either behave similarly to "Capture all" for large n or can clamp values too low causing large amounts of value clipping for small n.

The simulation results are recorded in a quantization parameter file which is exported for the quantization and code generation stage.

3.3 Code Generation

After importing the Protobuf file, an ONNX model object consists of metadata and a graph object containing a list of nodes (containing operator inputs, outputs, and attributes) - and lists of initializers (containing the dimensions and values of the trained parameters in the model). A C implementation is created by iterating through initializers and using the quantization ranges collected during the simulation stage to convert them into tensor structures containing quantized data buffers. Nodes representing operators are converted to calls to the inference library API. At this stage, I also apply operation fusion and a memory compression strategy to reduce the memory footprint of the model.

3.3.1 Quantizing and Allocating Trained Parameters

As previously mentioned, the floating point range used to quantize activation tensors is calculated during the simulation stage of the pipeline. There are no activation values to quantize before inference so for these tensors only quantization parameters (scale and zero point) are generated, and the empty tensor structure is allocated. Every weight and bias of the input model needs to be quantized, and so these values are read by the code generated, and quantization parameters are generated according to the options specified by the user (such as the `--weights-as-i8` flag). The quantization parameters of some tensors are dependent on others. For example, as discussed in section 2.6.5.1, to allow for a simpler integer GeMM operation the bias tensor used in a GeMM operation has a quantization scale equal to the product of the quantization scale of the GeMM input and weight tensors. Dependencies also exist when allocating the structures for convolution patch matrices (section 2.6.6), as the patch matrices contain the same data as the convolution input, so must have the same quantization parameters. A pass over the input graph identifies any dependencies, storing them in a map from a node name to a list of dependencies and a dependency type (a dependency type is either `Product` or `Same`, corresponding to the operation required to compute the dependent scale). When the parameters of the model are quantized, the code generator looks for dependencies and calculates quantization parameters accordingly.

When allocating memory for the model parameters their size must be calculated. This is trivial for most weight and bias tensors as they are typically just a two, or three-dimensional tensor for which the dimensions are directly known. Patch matrices used for convolution are more complicated since their dimensions depend on the size of the input to the convolution layer, as well as the attributes of the convolution operation such as stride, padding, and kernel size.

A C declaration and definition string is generated for every model parameter tensor structure and activation tensor structure. Qualifiers such as `const` are applied where possible. The definitions and declarations are then written into template C files.

3.3.2 Generating the Inference Function

My pipeline generates a single function named "infer". This function contains all of the inference library API calls to operators. The intended usage inside an embedded application is as follows:

- The application obtains input data to the model. This could come from a peripheral, GPIO, over a network, or elsewhere.
- This data is stored directly or copied into the model input tensor structure. It can be quantized using functions exposed by the inference library.
- The infer function is called, executing a sequence of operators.
- The model output can be read or copied from the model output tensor structure. It can be dequantized using functions exposed by the inference library.

I deliberately aimed to have a single entry point requiring minimal interaction between a developer and the internals of the generated model. Outside of the generated code/library, this keeps the model's

code footprint in an embedded application low. This is important because it allows for a model to be redesigned with a completely different topology, and (as long as the input and output shapes are the same) the newly exported model can be integrated into an application without manually changing any code. Only the code generated by the pipeline should change.

This inference function is generated by traversing the model graph in sequence and converting each node to an inference library operator. The C code generator contains a template for each inference library function call, and this template is populated with the appropriate inputs and outputs and written to the body of the "infer" function. At this stage, operation fusion is applied, and parameters for the fixed-point conversion between the quantization scales of different operators are generated as well.

3.3.3 Compressing Memory Usage

MISRA C guidelines disallow any use of dynamic memory[19], this means that all activation tensors and trained parameters must be statically allocated when the model is compiled. Both the activations and trained parameters become tensor structures in the inference library. However, parameters can be stored as constants since they will be needed for every inference. Activations are temporary and can be stored in a shared region of memory where they are overwritten after use. The simplest approach is to give each activation tensor its own unique region of memory. However, this can result in extremely large models, so compression via static analysis of the graph structure is needed.

I devised a simple static memory allocation policy consisting of two regions, each operator either reads from one region and writes to the other, or certain operations can be performed in place. During code generation, my program makes a pass over the input graph labeling regions with a "Region mapping". For example, in my implementation GeMM cannot be performed in place, since a single input matrix element may be needed to calculate more than one output matrix element, so GeMM is given the region mapping (A, B), meaning the inputs will be stored in the region A and the results will be written to region B. The next operator may be a ReLU, which can be performed in-place, and so is given the region mapping (B, B). This region mapping policy only works for models which do not contain branches, which is suitable for my test models. However, it could easily be extended to include an offset in each region mapping, allowing for multiple inputs and outputs. After the mapping is defined for each operator, the code generator will define the input/output tensors so that data is allocated either A or B. Finally, the code defining the size of regions A and B are generated, and they are sized according to the largest tensor stored in each region.

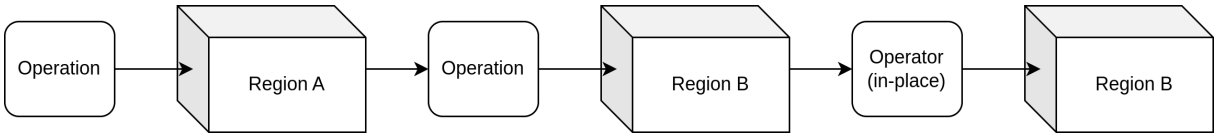


Figure 3.2: Example of assignment of operators and their "Region Mapping"

My static memory allocation algorithm is far from optimal. A better solution would be to share a single static pool of memory between all the operators, keeping track of which regions are used by operators at each layer of the graph and overwriting old data. This is possible without dynamic memory allocation (which is disallowed by MISRA C) since the memory regions used by every operator can be calculated before inference, and the sequence of operator execution is also known. In a best-case scenario, the total size of this static pool would only need to be the size of the sum of the largest layer's inputs and outputs. However, in many instances, my memory allocation still saves a large amount of memory compared to allocating each tensor individually.

Model	Original Size (bytes)	Size After Compression (bytes)	Percent Reduction
Iris FC	331	256	22.6%
MNIST-Conv-Large	19988	10656	46.6%
MNIST-Conv-Small	9628	7840	18.5%
MNIST-Conv-Small-1Channel	11396	8624	24.3%

Table 3.1: Comparison of the memory allocated for activation tensors in each test model, before and after applying my memory compression policy

As shown in table 3.1, my test models showed a significant reduction in the memory required to store activations, usually reducing the memory required by around 20% compared to allocating all activation tensors individually. Note that the size shown in table 3.1 size does not include the space allocated for weights and biases. In cases where high-memory operators are allocated to the same region, even more memory can be saved. This is seen in the MNIST-Conv-Small-1Channel model where the patch matrices used in the convolution layers are allocated to the same region, allowing for a 46% reduction in the memory allocated for activation tensors.

3.3.4 Graph Optmization

Due to the limited capabilities of my target hardware, my testing models are relatively simple compared to larger DNNs designed for more powerful embedded processors. For this reason, only a small number of graph optimizations are possible on my models. I have focused on operation fusion since it tends to consistently provide a performance increase as well as potential model compression benefits on shallow CNNs and fully connected networks.

3.3.4.1 Fusing activation functions

A simple graph optimization that is widely applicable to common DNN architectures is operation fusion. Common operator patterns such as GeMM followed by ReLU can be fused into a single operator for a number of performance benefits. Firstly, the memory footprint of the network is reduced, since output tensors do not have to be generated for the output of both operators, only a single output tensor needs to be allocated. Secondly, inference time is improved because both operations can be performed in the same pass over the input tensor data, improving caching performance as well as simply executing fewer instructions overall than executing the operators separately.

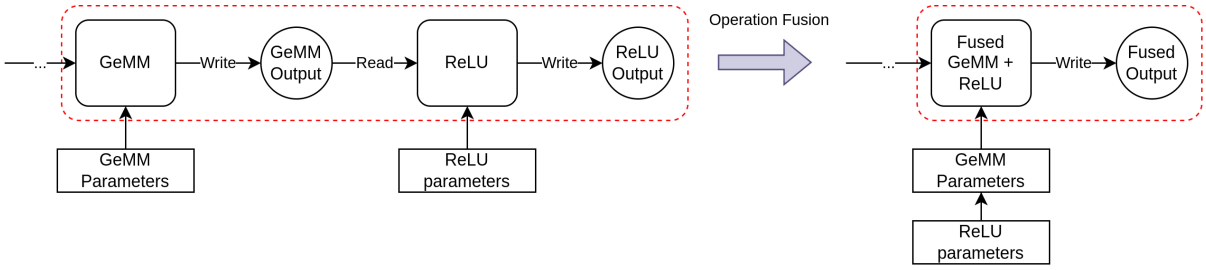


Figure 3.3: Example of the operation fusion applied in the graph optimization stage

In a DNN optimized with post-training quantization, operation fusion must be performed after the simulation stage as the quantization parameters for both original operators are needed for reproducing the behavior of the original network with a fused operator. A mathematical expression for a quantized fused operator can be derived using a similar method to a single quantized operator, substituting the expression for the result of the first operator for the input to the second.

In my graph compiler operation fusion is performed at the point when an operation node is converted to an API call to the inference library. Conditions are checked to see if a fusion is possible, for example, if the current node is a GeMM operation and its output node feeds exclusively into a ReLU node, fusion is possible.

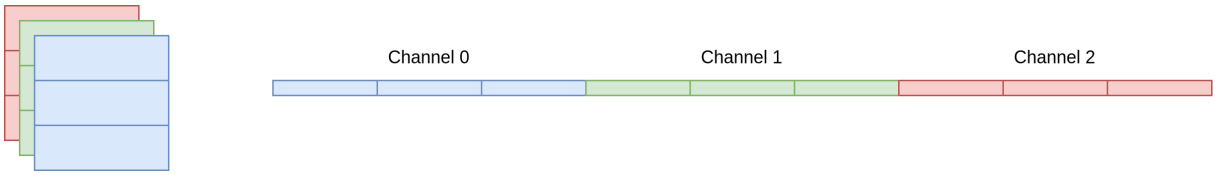


Figure 3.4: Layout of a tensor structure in memory

I have implemented operation fusion in the following cases:

- A GeMM node is followed exclusively by a ReLU node
- A GeMM node is followed exclusively by a Clip node
- A Conv node is followed exclusively by a ReLU node
- A Conv node is followed exclusively by a Clip node

3.4 Inference Library Implementation

3.4.1 Tensor memory model

The structure representing a tensor in my inference library is kept simple and lightweight. It is composed of a byte array to store data, an integer array of dimensions, and a small amount of metadata for type checking and error handling. Depending on the operation, elements may be read from a tensor consecutively row-wise or column-wise. I opted to store in a row-major format with the intention of pre-transposing tensors at the code-generation stage in order to ensure that elements being read consecutively would be contiguous in memory, however, this optimization is not yet implemented. Where a tensor has more than two dimensions, the 3rd dimension (usually the different channels of a tensor) is stored consecutively as shown in figure 3.4.

3.4.1.1 Indexing Tensors

My tensor structure stores all data as UINT8, with reading and writing being handled by copying memory to/from differently sized types given an offset calculated from a requested position and the size of a given data type. In order to index tensors to read/write values, I created a single function allowing for the reading and writing of any C datatype by making use of the C pre-processor. A type, for example, "`int8_t`", is provided when indexing a tensor, this is converted to "`sizeof(int8_t)`" by the C pre-processor. An offset in memory is then calculated and the appropriate number of bytes is copied to/from the tensor. This allowed me to create operators that are type-ambiguous, as a single interface can be used to index tensors storing any datatype.

In order to implement the stacking of kernels for operator operations such as Im2col convolution (section 2.6.6), I implemented array broadcasting similar to the functionality in Numpy[38]. When a tensor element is requested at a certain position, if that position would be outside the dimensions of the tensor then the index is adjusted in order to simulate the tensor being "tiled" in memory (figure 3.5). For example requesting index $(1, 5, 6)$ of a tensor with dimensions $(1, 3, 3)$ would read from the actual index $(0, 1, 2)$, accounting for zero indexing. The broadcasting behavior is implemented per dimension, so reading the same size-$(1, 3, 3)$ tensor at index $(0, 0, 9)$ would read from the actual index $(0, 0, 1)$, rather than reading from any other dimension.

I made this design decision based on the assumption that storing a single copy of a kernel and "broadcasting" it would achieve better performance than duplicating memory due to more optimal caching and model compression. However, the number of integer divisions and condition checks required to implement the broadcasting functionality on every read/write from a tensor may have large performance implications when scaled up to large kernels. Thankfully it would be a simple change to make the graph compiler choose a non-broadcasting variant of the indexing function to remove the additional overhead where it was not necessary.

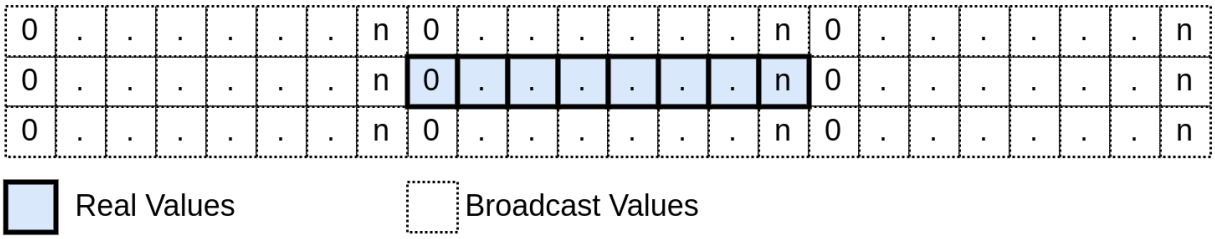


Figure 3.5: Example of the simulated tiling of arrays in memory

3.4.2 Operator Implementation

The ONNX specification changes quickly with new operators rapidly added to support the latest DNN architectures, even extremely universal operators such as GeMM are subject to slight changes between minor versions. Currently, there are well over one hundred operators in the ONNX specification. It would be infeasible and impractical to attempt to implement all of these operators, and many of them would be used extremely infrequently in an embedded application. For this reason, I have focused on implementing only a few core operators extremely common in CNNs, with the intention to expand the number of supported operations later.

To remain general-purpose, ONNX operators are defined at a high level, with large numbers of arguments creating an interface similar to the DNN design frameworks such as PyTorch. A direct translation of these operator variants from the high-level specification to a C implementation is not always simple, especially since the ONNX specification does not include precise definitions of operators, instead only providing example inputs and outputs. Thankfully most DNN frameworks do precisely define their operators, and the ONNX specifications intended behavior can be inferred by comparing the behavior of a function from a DNN framework to the ONNX operator it generates. For example, PyTorch's `nn.Conv2D` function, for which a precise definition is given, is generally converted to an ONNX `Conv` node with certain arguments.

All operators are tested by comparing the output to the PyTorch operation which generates the operator to the output of my implementation. Appendix A shows a comparison between the output of a full precision Conv2D operation in PyTorch and the de-quantized output of the model generated when the same PyTorch model is converted to C by my pipeline.

3.5 RISC-V Vectorization

The RISC-V vector (RVV) extension was ratified in November 2021[24], and despite the lack of availability of affordable development boards with RVV chips, a large amount of intellectual property has been developed, with SiFive already offering the intelligence X280 and performance P270 processors[39]. It is likely that many more small RVV cores for embedded systems will be developed, as this will enable greater competition with processors such as Arm's Cortex-M55, which supports vector instructions and is targeted at embedded ML. Vectorized code can be written using the RVV intrinsics[40], although the intrinsics themselves are not yet finalized. The Clang compiler has supported RVV intrinsics since 2019, also supporting auto-vectorisation[41]. GCC 13 has added RVV intrinsic support extremely recently[42] and GCC 14 is scheduled to support auto-vectorization for RISC-V.

In order to facilitate new RVV hardware I have implemented a vector version of the integer-only quantized GeMM operator using the RVV intrinsics. Currently, no public cycle-accurate modeling of an RVV core is available, and no affordable development boards supporting the RVV 1.0 specification are currently available to buy. This means I have been unable to verify an increase in performance compared to the non-vector version of my operators, however, they have been verified for functional correctness in both the Spike and Qemu instruction-level simulators.

Implementing the quantized, integer-only operators with vector intrinsics is complicated by the requirement for accumulating different data types, as described in section 2.6.5.1. Firstly the 8-bit elements must be loaded, then each one must be extended to 32-bits to fan out each value into vector lanes compatible with the accumulator vector. The RVV intrinsics do not allow support casting as in normal C,

instead, if UINT8 values were loaded and extended to 32 bits specific casting intrinsics must be used for the intrinsics API to allow them to be accumulated into an INT32-type vector. This is a curious feature of the RVV intrinsics, as there are no vector casting instructions in the RVV specification, so the generated assembly appears to ignore the casting intrinsics. An implementation of my vectorized integer-only GeMM algorithm can be seen in appendix B.

When compared to code auto-vectorized by Clang, my bespoke implementations generate more concise assembly using fewer non-vector instructions. A simple example of this for a multiply and accumulate operation over two arrays is shown in appendix C. For larger and more complex operators, this difference in generated assembly is even greater. My findings here align with other comparisons between the RVV auto-vectorization and hand-vectorized codes, and it has been found that RVV codes vectorized by hand can perform significantly better[43]. It is likely to be a long time before bespoke vectorization for RISC-V is offered in other libraries.

Chapter 4

Critical Evaluation

4.1 Test Hardware and Simulators

The SiFive HiFive1 Rev B (HiFive) is a development board based on the SiFive Freedom E310-G002 SoC. This core uses a 5-stage, in-order RV32IMAC pipeline running at up to 320MHz. It has 16KB of 2-way L1 instruction cache and 16KB of data SRAM scratchpad with a 2-cycle load latency. The micro-architecture supports hardware multiply and divide. The E310 is designed to compete with ARM Cortex-R4 series processors. On the HiFive1 Rev B board, there is 32MiB of external flash memory which can be used to store programs and data. Programs can be compiled and flashed onto the HiFive board using the SiFive Freedom E SDK[44].

Due to the limited SRAM on the development board, the size of a test model is generally limited by the size of activations. Along with a small amount of operational overhead, they must fit in the 16KB data scratchpad memory. Network weights and biases are all able to be stored in the read-only memory, so these are far less of a limiting factor for my testing. Throughout development, I also conducted functional testing using the SimEng simulator[45] developed by the University of Bristol. At the time of writing, SimEng supports the RV64IMA base ISA and extensions.

4.2 Test Models

Iris Dataset The Iris dataset[46, 47] is a multivariate dataset widely used in machine learning as an example of a classification task. It contains measurements of the petal width, petal length, sepal width, and sepal length of 150 iris flowers, and classifies them into three species.

MNIST Dataset The MNIST (Modified National Institute of Standards and Technology database) dataset[48] is widely used for image recognition tasks. It contains 70000 greyscale 28x28 pixel images of handwritten digits. Each image is normalized. The MNIST dataset is a common benchmark for machine learning models and new DNN architectures.

To evaluate the effectiveness of my optimization pipeline I developed several test models composed of the various operators I have implemented. These models were developed in PyTorch and then exported to ONNX. These ONNX models were processed by the pipeline. Some minor measures were taken to design the models in a way that would constrain activation ranges, resulting in higher precision of the quantized model. For example, the ReLU6 activation function clips values between 0 and 6, compared to standard ReLU which only clips values less than 0. A ReLU6 layer limits the magnitude of an activation entering succeeding layers, which makes the network less likely to have a large dynamic range in activation tensors. This is not universal, and the pipeline handles many models just as well using standard ReLU. Batch normalization and dropout layers were also used in training to reduce over-fitting, but these do not affect inference and were optimized out by PyTorch when exporting to ONNX. Each test model was trained for 500 epochs with an 80/20 train/test split on their respective datasets. All networks were trained using the Adam optimization algorithm[49]. Batch size and hyper-parameters varied for each model.

The test models are not intended to be examples of optimal networks for their respective classification problems, they merely provide a varied sequence of operations similar to what may be deployed on a

small embedded processor such as the E310-G002. Once trained, each model has a baseline accuracy to which I can compare the accuracy of the optimized model generated by my pipeline, as well as a baseline inference latency to which I can compare the effectiveness of certain optimizations.

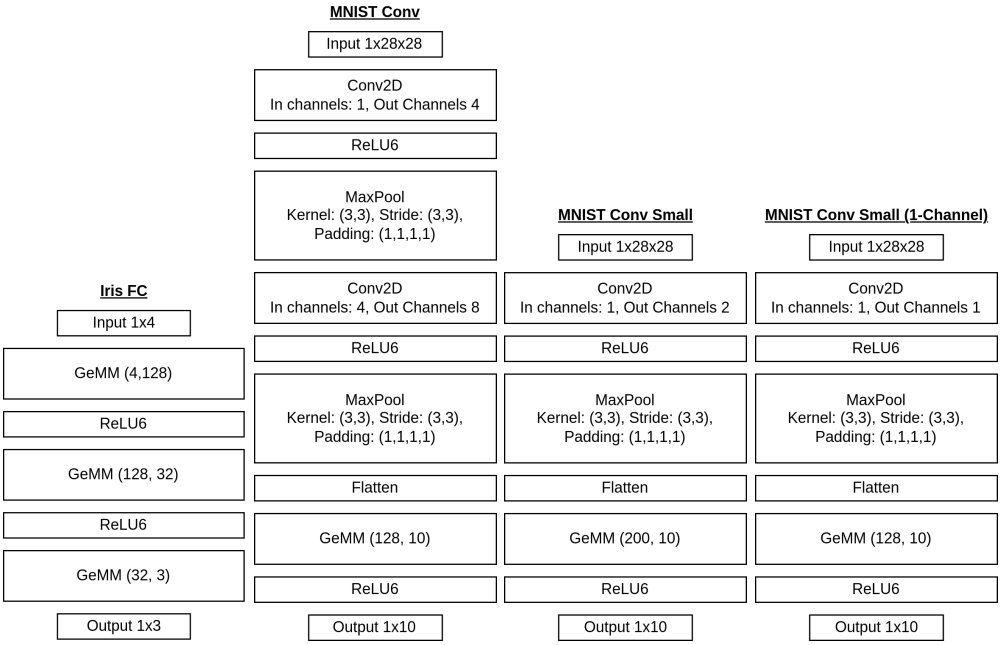


Figure 4.1: The topology of each of my test models

4.3 Network Precision Loss

Generally, my pipeline was able to generate an integer-only binary with low accuracy degradation. As shown in table 4.1, for all my test models I was able to achieve top-1 accuracy within 5% of the original network by either changing the options passed to the pipeline script, or (less frequently) by making a trivial change to the network architecture (such as swapping a ReLU activation function for ReLU6).

The effectiveness of the pipeline is strongly influenced by the dynamic range of every activation tensor and parameter in the input model. The models suffering more precision loss were always the ones containing a layer with a larger dynamic range. For example, when comparing results (Table 4.1) using the ReLU activation function and the `--weights-as-i8` pipeline flag, the model MNIST-Conv-Small-1Channel had a relatively bad accuracy degradation of 4.7%[†], whereas the larger, more complex MNIST-Conv model only suffered a 0.1% accuracy loss[††]. Table 4.2 shows a comparison between the dynamic ranges of weight tensors in these models, and the larger average range of weights in the MNIST-Conv-Small-1Channel model explains why it suffers more than MNIST-Conv once quantized. The 4.7% accuracy degradation is not catastrophic, however, when the `--weights-as-i8` flag is added the accuracy degradation is far worse at 23.2%[†††]. This is likely due to the forced expansion of the quantization range of the weight tensors caused by the symmetric quantization strategy. For example, the range of the tensor is from -0.934 to 1.893, but with symmetric quantization, the quantization range will be chosen as -1.893 to 1.893 (Section 2.6.2.3), losing a large amount of precision. Making things worse, the mean value of this tensor is 0.611 meaning it is skewed in the positive direction, so precision is lost from the area of the tensor containing the most values.

The amount of precision loss a network is able to suffer before accuracy degrades significantly is highly dependent on the network architecture and training. The same network with a different weight initialization or trained with different hyper-parameters may suffer very differently. I observed over-fitting and network generalization to be a reliable predictors of the quality of network quantization (Table 4.3). This follows intuitively from the underlying mathematics of a neural network since a generalized model is less reliant on precisely tuned parameters. During quantization precision is lost, so a better-generalized network will lose less performance once quantized as it was not using the precision to begin with. Dropout has been shown to be a simple method to counter over-fitting and improve generalization in DNNs[50].

I added a single dropout layer to the MNIST-Conv-Small-1Channel which was trained several times for 50 epochs with different dropout levels. The model variants with more dropout showed less accuracy degradation.

Even though most binaries generated by the pipeline achieved good performance, more static analysis could be performed on an input model to predict quantization error and recommend alternative quantization strategies. I include tools to quickly and easily evaluate the output network performance compared to the original model, however, it would add convenience to the tool if a choice of quantization strategy could be guided or even fully automated to prevent a user from having to compile and test a model in order to assess the resulting performance. It would be an easy addition to the pipeline scripts to provide a quantification of the error between the original and quantized parameters under different quantization strategies.

Test Model	**Activation Function**	**Full Precision Accuracy (Top-1 %)**	**Weight Datatype**	**Optimized Accuracy (Top-1 %)**
IRIS-FC	ReLU	97.3	int8	96.6
			uint8	96.6
	ReLU6	98.6	int8	98.6
			uint8	98.6
MNIST-Conv	ReLU	79.6	int8	79.6
			uint8	79.5 [††]
	ReLU6	90.9	int8	89.6
			uint8	89.9
MNIST-Conv-Small	ReLU	88.4	int8	85.9
			uint8	85.8
	ReLU6	89.9	int8	85.3
			uint8	86.3
MNIST-Conv-Small-1Channel	ReLU	68.5	int8	45.3 [†††]
			uint8	64.8 [†]
	ReLU6	90.2	int8	85.3
			uint8	86.3

Table 4.1: A comparison between the Top-1 accuracy of the test models at full precision, and after quantization, with different activation functions and weight-quantization strategy. Where the weight datatype is INT8, symmetric quantization of trained parameters was used. Where the weight datatype is UINT8, asymmetric quantization was used.

Model (ReLU)	**Parameter**	**Min**	**Max**	**Range**	**Tensor Mean**
MNIST-Conv	Conv1 Weight	-0.531	1.226	1.757	0.502
	Conv2 Weight	-0.694	0.455	1.150	-0.015
	FC1 Weight	-0.785	0.535	1.320	-0.045
Average:				**1.409**	**0.147**
MNIST-Conv-Small-1Channel	Conv1 Weight	-0.934	1.893	2.827	0.611
	FC1 Weight	-1.049	1.033	2.083	-0.057
Average:				**2.455**	**0.277**

Table 4.2: Comparison of the range of trained parameters in the MNIST-Conv and MNIST-Conv-Small-1Channel models

Dropout	Full precision Accuracy	Optimised Accuracy	Degradation
0.1	88.2	63.1	25.1
0.2	62.8	48.6	14.2
0.3	86.6	74.6	12
0.4	81.6	71.89	9.71
0.5	86.6	80.1	6.5

Table 4.3: Effect of over-fitting on the accuracy of the quantized MNIST-Conv-Small-1Channel model when trained with different levels of dropout

4.4 Model Performance on Hardware

There is no question of the effectiveness of converting floating point operations to integer operations for the RV32IMAC E310-G002 chip. For example, a simple multiply-and-accumulate loop over two 250-element arrays of numbers took 149486 cycles on FLOAT32 values, and only 31642 cycles on INT32 values. The MNIST-Conv and MNIST-Conv-Small test models would not be possible to deploy onto my test hardware without the compression enabled by quantization, so it would not be possible to compare the integer and floating point implementations of these networks on this hardware.

It is unfortunate that I am unable to offer a comparison between the performance of models compiled with my pipeline compared to tools such as TFLite. I was unable to create a working RISC-V port of TFLite as I found little documented precedence for the process, and manually porting the library proved to be infeasible during the project time frame. Due to the maturity of tools such as TFLite, it is likely that its operators would be more efficient than my own implementations. That being said, my inference library and optimization pipeline certainly provide sufficient performance for the deployment of small models to RISC-V targets, and offer a minimal-yet-effective, practical solution to the problem of DNN optimization for embedded RISC-V processors.

4.4.1 Effect of Quantization Strategy on Inference Latency

It is predicted in relevant literature[26, 51] (section 2.6.2.3) that using a symmetric quantization strategy for weights will improve performance due to the ability to represent the parameter value 0 with the integer 0, allowing for certain data to be skipped during common operations such as multiply-and-accumulate. This performance improvement was not observed in my test models. The optimization is only effective where there are high levels of sparsity - meaning a large proportion of the network parameters are zero. However, my test models do not exhibit much sparsity. Comparing asymmetric and symmetric quantization I observed a negligible effect on inference latency when testing on the E310-G002 (Table 4.4).

It is possible to induce sparsity into CNNs through various training strategies[51]. However, I induced sparsity on my model by forcing a certain percentage of network parameters to be zero. For these tests, I ignored the effect this had on network accuracy and only considered cycle count since the underlying operators are the same regardless of network accuracy. I measured inference latency on the Iris-FC at several levels of sparsity (Table 4.5) shows that again, no performance improvement was observed. The reason for this is inefficiencies in my inference library. GeMM is the most significant operation in this model, and every element must still be loaded, and accumulated into a sum before being re-scaled. For quantized GeMM the calculation of “Term2” and “Term4” (See section 2.6.5.1 equation 2.1) should ideally be entirely removed with symmetric quantization, as z_W is zero. I do not see a benefit of this in my implementation because ”Term2” is calculated in the same loop as the other terms, and so suffers from the same slowdowns caused by the rest of the memory-bound loop operation. The multiplication with z_W is also not performed until every element comprising “Term2” has been loaded and summed together. A code refactor as well as additional operator variants would allow my inference library to better exploit optimizations such as these.

Model	Weight Datatype	Inference Latency (cycles)
IRIS-FC	uint8	1116808
	int8	1123942
MNIST-Conv-Small	uint8	5787955
	int8	5779245
MNIST-Conv-Small-1Channel	uint8	3443926
	int8	3443971

Table 4.4: Comparison of average cycles taken to complete a single inference between asymmetric (weights UINT8) and symmetric (weights INT8) quantization

Induced Sparsity	Inference Cycles
0%	1123851
20%	1126006
40%	1123858
60%	1126007
80%	1123858
100%	1123804

Table 4.5: Effect of induced sparsity on the inference latency of the Iris-FC model

4.4.2 Effect of Operation Fusion on Inference Latency

To assess the effectiveness of operation fusion I created a test model using the iris data set. The model is a simple fully connected network with a single hidden layer. This results in a graph of two GeMM operations each followed by a ReLU6 operation. Once fused these become two GeMM+ReLU6 fused operations. I can resize the hidden layer to adjust the memory intensity of the network. I began with a small layer size of 128 and increased incrementally until the model would no longer fit into the memory of the HiFive development board. My hypothesis was that larger layer sizes would have bad caching performance as multiple large tensors would need to be moved in and out of a small cache, and that operation fusion would improve performance by reducing the total data movement during inference. However, as shown in table 4.6, only a minor performance increase (typically around a 1% reduction in inference latency) was observed on each layer size. In my inference library, ReLU and Clip operations are performed in place, and therefore I do not observe the compression benefits that might be seen by fusing other operators.

In a fused operation of GeMM and ReLU, the GeMM operator is far more expensive. ReLU is implemented as a simple loop over the data of a tensor, applying a clipping operation with a multiplication and bit shift for fixed point re-scaling. This loop is far easier to optimize and even on the largest hidden layer size (size 4800) the ReLU operation only iterates over 4800 array elements. This is in comparison to the 19200-element weight tensor which is continually read and operated on in the corresponding GeMM operation. Realizing this, it becomes clear that in these network architectures, my currently implemented fused operators are only serving to remove a relatively small and fast loop.

Even though the inference library is compiled separately, once the compiled model is compiled and linked to the inference library optimizations may be applied to the entire executable. I ensured there was no possibility of link-time optimization applying loop fusion to the models compiled without operation fusion enabled by re-testing with all compiler optimizations turned off. The results I observed showed a similar behavior to that seen in table 4.6.

Despite less of a performance boost than I had hoped, the implemented operation fusion still saves a measurable number of cycles, and the cycles saved increase with network size. Figure 4.3 shows clear - although sub-linear - scaling between layer size and cycles saved by inference.

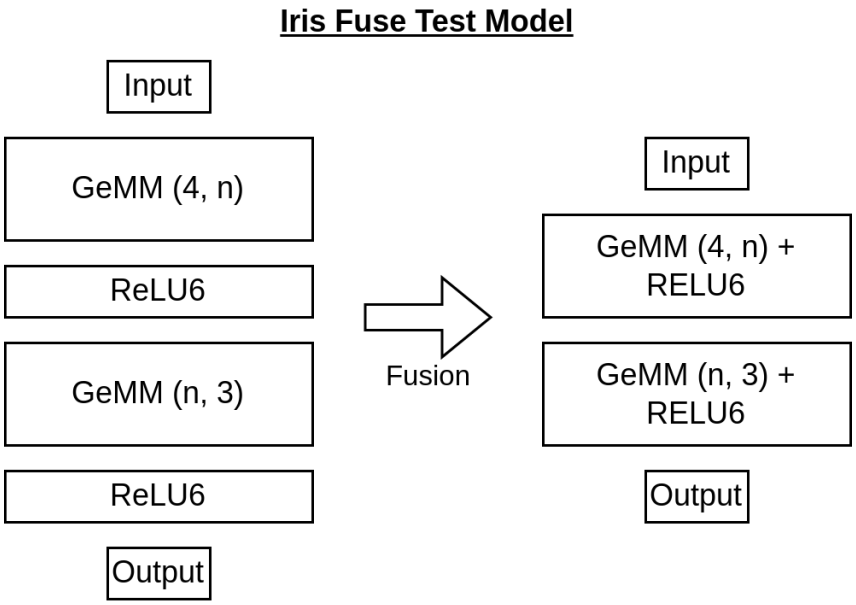


Figure 4.2: Operation fusion test model

Hidden Layer Size	**Inference Cycles (No Fusion)**	**Inference Cycles (Fusion)**	**Percent Improvement**
128	478330	468065	2.19%
500	1495026	1472555	1.53%
1000	2849050	2817698	1.11%
1500	4211491	4169625	1.00%
2000	5576660	5530606	0.83%
2500	6934949	6882607	0.76%
3000	8299336	8224003	0.92%
3500	9651701	9581364	0.73%
4096	11272679	11184307	0.79%
4800	13191576	13092931	0.75%

Table 4.6: Effect of induced sparsity on inference latency

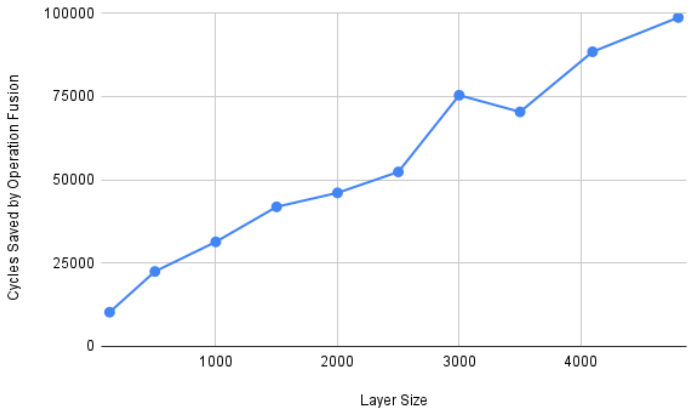


Figure 4.3: Cycles saved by operation fusion on different layer sizes of the fusion test model

4.5 MISRA Compliance

Due to my software architecture decisions throughout the development of the inference library, only minor MISRA code violations remain.

The remaining rule violations are roughly as severe as:

- MISRA 2012 Rule 2.7: There should be no unused parameters in functions
- MISRA 2012 Rule 7.3: The lowercase character “l” shall not be used in a literal suffix.
- MISRA 2012 Rule 12.1: The precedence of operators within expressions should be made explicit.
- MISRA 2012 Rule 15.7: All if ... else if constructs shall be terminated with an else statement.

These are all extremely minor fixes or at worst call for a small refactor of a function’s logic.

Chapter 5

Conclusion

5.1 Summary of Achievements

I have implemented a minimal version of a DNN optimization pipeline and inference library. The pipeline takes a pre-trained FLOAT32 input model in ONNX format and performs several steps to make the model suitable for deployment on a resource-constrained RISC-V target. First, the model is simulated to gather data used for quantization, next this data is used to quantize the model to an integer datatype. After static analysis of the input model, compression and graph optimizations are applied, and a C implementation of the model is generated from calls to an inference library I have developed.

The inference library implements an integer-only set of core operators commonly used in DNNs, as well as fused variants of some operators. Aside from inside functions to ingest data to a quantized format, no floating point instructions are used at all. Where arithmetic on real numbers is required, fixed point arithmetic using values generated by static analysis of the input model provides a faster solution to software floating point. To preserve precision on quantized data, operator implementations are based on rigorous mathematical definitions. The inference library implements efficient algorithms such as Im2col convolution to improve the performance of expensive operators. The inference library is designed to be extensible, implementing mechanisms such as array broadcasting to enable the easy translation of high-level operators to efficient C code. All implemented operators are tested against reference operators from a library such as PyTorch. Although it is not yet completely MISRA-C compliant, the software architecture of the inference library is designed to avoid violations that would require extensive refactoring to remove from an established code base, such as the use of dynamic memory allocation.

The pipeline has been tested on a set of differently-sized test models and, as evidenced by table 4.1, has shown to be capable of implementing a quantized, integer-only version of each model with minimal precision loss. I have tested these models on the SiFive HiFive1 Rev B development board and shown the effect of various quantization strategies and optimizations on the performance of a binary generated with the current version of my tool. Models which would otherwise have been unable to fit into the memory of the test board were able to be compressed, optimized, and successfully tested, increasing the potential of the limited hardware capabilities of this development board. Model compression was further aided by a custom static memory allocation algorithm.

The pipeline is able to be used from a simple command line interface and integrates well with various build systems and existing embedded machine learning development environments. Whilst performance can certainly be improved, the pipeline offers a simple and effective solution for embedded machine learning engineers targeting RISC-V, providing an easy alternative to the painful experience of porting other optimization tools to RISC-V. To further differentiate from existing tooling, I have also provided an implementation of the quantized GeMM operator using the RISC-V vector intrinsics.

5.2 Current Project Status

Overall, the project has achieved its goal of presenting a minimal yet practical solution to a problem that currently remains unsolved by major organizations. A broad range of problems have been tackled

to enable the development of a complete optimization pipeline, and I feel that the resulting tool offers significant utility to an embedded ML engineer targeting RISC-V processors.

5.2.1 Summary of Features

Currently, the graph compiler and inference library support the following operations on data quantized to 8-bit integer types:

- Generalised Matrix Multiplication (GeMM) with support for transposing input tensors
- Vectorised GeMM
- Two-dimensional convolution using the Im2col algorithm, with support for varying kernel sizes, strides, and zero padding
- Maximum Pooling with support for varying kernel sizes, strides, and zero padding.
- Average Pooling with support for varying kernel sizes, strides, and zero padding
- The Rectified Linear Unit activation (ReLU) function
- Value clipping between two values (used to implement ReLU6
- Tensor flattening
- Fused GeMM and ReLU
- Fused GeMM and value clipping

Aside from the fused operators, all of these operators correspond directly to operators in the ONNX spec. As long as an ONNX graph does not contain branches and only contains these operators, my pipeline should be able to process the model.

The currently supported quantization strategies are:

- Asymmetric quantization (UINT8) of activations and weights, with symmetric quantization (INT32) of biases
- Asymmetric quantization (UINT8) of activations, symmetric quantization (INT8) of weights, and symmetric quantization (INT32) of biases

The simulation stage of the pipeline currently has three methods of choosing quantization parameters as mentioned in section 3.2. Currently, an option for choosing between these is not exposed to a user and only the simple histogram-based range selection is used. A small number of graph optimizations are applied as mentioned in section 3.3.4.1.

5.3 Future Work

I feel that I have successfully achieved the goals of the project and provided a functional optimization toolkit. However, due to the broad range of problems I have tackled, the pipeline tool has a large amount of scope for improvement. Each core component of the optimization pipeline could be extended with dozens of features. The inference library can also be considerably expanded. Below I have highlighted some changes and improvements which I think would be most effective in the short to medium term.

5.3.1 Additions and Changes to Inference Library

In the immediate future, there are a number of changes and minor optimizations that would greatly benefit the performance and reliability of the inference library. Firstly, as network weights can either be quantized as INT8 or UINT8, certain operators such as GeMM contain large amounts of condition checking. To avoid this it would be preferable for the library to contain several variants of these operators, with each one optimized for its specific context. When generating the model implementation, instead of using a call to a general implementation of quantized GeMM, the graph compiler could generate a call to the appropriate GeMM variant. This would improve the efficiency of inference in general, and also allow the inference library to better exploit optimizations such as sparsity - an optimization the inference library currently fails to take advantage of as shown in section 4.4.1. As more RISC-V cores are developed, additional operator variants could be implemented to target specific microarchitectures.

Another example of an un-exploited optimization is the ability to pre-compute certain parts of some operators before inference. For example, as mentioned in section 2.6.3.1 certain terms of equation 2.1 could be calculated by the graph compiler and hard-coded into the model, saving computation during inference. The scope for this type of optimization will grow as more operators and their vectorized variants are added.

As demonstrated in table 4.1, certain models lose very little accuracy even after quantization. This implies that more precision could be taken from the model whilst preserving acceptable model accuracy. In order to do this the optimization pipeline could be adapted to support mixed-precision operations, for example, using int4 parameters. This could potentially enable up to 8x model compression compared to FLOAT32, as well as inference latency improvements. It would also be beneficial to allow quantization strategies to be applied per tensor, per channel, or even per row. With sufficient extensions to the graph compiler, this would allow the optimization pipeline to waste far less dynamic range and preserve even more precision.

As mentioned in section 4.5, the inference library still falls short of a number of minor MISRA C programming guidelines. Fixing this is certainly achievable in the short term, and would greatly improve the credibility of the code base.

5.3.2 Upgrades to the Graph Compiler

In order to support more network architectures, it is extremely important to upgrade the graph compiler to be able to process graphs containing branches, and potentially even cycles for supporting ResNet[52] architectures. As well as support for more complex graph structures, more quantized operators need to be derived mathematically, implemented in the inference library, and supported by the graph compiler. The most important operators to add in the short term would be tensor reshaping utilities, as well as batch normalization. Adding these features would allow the optimization pipeline to work with large and complex models on the scale of AlexNet[53] and various MobileNets[54].

Improving the graph compiler's model compression and static memory algorithm as described in section 3.3.3 would allow larger models to be deployed in limited memory than is possible with the current algorithm. Supporting larger models would vastly expand the utility of the pipeline, but better compression of small models would increase the feasibility of integrating them into embedded applications which might themselves occupy a large proportion of the memory available to a hardware target. Model compression would also be greatly improved by the implementation of more memory-efficient convolution operators in the inference library such as the algorithms mentioned in section 2.6.6[37].

The graph compiler could also be upgraded to improve inference latency by analyzing the access patterns of tensors and ensuring they were stored optimally in memory. For example, currently, the quantized weights for a GeMM operation would be stored in the same orientation as they appear in the ONNX model data, and the call to the GeMM would be generated with the arguments set to read the weights as if they were transposed. Caching performance could be greatly improved by ensuring that weights are stored in a manner favorable to the way they are read during inference.

5.3.3 Other Improvements

Usability Improvements As the graph compiler and inference library mature, there will naturally be added complexity for a user due to additional choices of quantization strategy, options for enabling/disabling certain optimizations, and potentially even options to tune a model to utilize specific computational resources more or less - for example, adjusting a trade-off between inference latency, memory usage, or power usage. Abstracting away complex choices from the user via pre-configured profiles designed for specific microarchitectures may be a helpful way to mitigate these problems.

Potential For Integration With Other Tools Once the inference library is mature enough to compare with advanced inference libraries such as those powering tools like TFLite, there is scope to consider additional utility as a back end for tools such as Apache TVM. This would join together any RISC-V-specific optimization contained in my inference library - for instance the bespoke vectorization - with the advanced cache tuning and other optimization features of TVM.

Bibliography

[1] "ONNX." [Online]. Available: https://onnx.ai/

[2] "PyTorch." [Online]. Available: https://pytorch.org/

[3] "TensorFlow." [Online]. Available: https://www.tensorflow.org

[4] "Protobuf." [Online]. Available: https://protobuf.dev/

[5] "NNEF." [Online]. Available: https://www.khronos.org/nnef

[6] "Onnx runtime." [Online]. Available: https://onnxruntime.ai/

[7] "Arm market share statistics." [Online]. Available: https://www.statista.com/statistics/1132112/arm-market-share-targets/

[8] "Analyzing the risc-v cpu market for sip, socs, ai and design starts." [Online]. Available: https://semico.com/content/analyzing-risc-v-cpu-market-sip-socs-ai-and-design-starts

[9] "TensorFlow Lite." [Online]. Available: https://www.tensorflow.org/lite

[10] M. S. Louis, Z. Azad, L. Delshadtehrani, S. Gupta, P. Warden, V. J. Reddi, and A. Joshi, "Towards deep learning using tensorflow lite on risc-v," in *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*, vol. 1, 2019, p. 6.

[11] Y.-R. Chen, H.-H. Liao, C.-H. Chang, C.-C. Lin, C.-L. Lee, Y.-M. Chang, C.-C. Yang, and J.-K. Lee, "Experiments and optimizations for tvm on risc-v architectures with p extension," in *2020 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2020, pp. 1–4.

[12] "TVM BYOC." [Online]. Available: https://tvm.apache.org/2020/07/15/how-to-bring-your-own-codegen-to-tvm

[13] L. Calicchia, V. Ciotoli, G. C. Cardarilli, L. di Nunzio, R. Fazzolari, A. Nannarelli, and M. Re, "Digital signal processing accelerator for risc-v," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2019, pp. 703–706.

[14] "ONNX Optimizer." [Online]. Available: https://github.com/onnx/optimizer

[15] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. Ma, Q. Xu, H. Liu, P. M. Phothilimthana, S. Wang, A. Goldie, A. Mirhoseini, and J. Laudon, "Transferable graph optimizers for ml compilers," 2021.

[16] "TensorFlow Grappler." [Online]. Available: https://www.tensorflow.org/guide/graph_optimization

[17] S. Kalapothas, M. Galetakis, G. Flamis, F. Plessas, and P. Kitsos, "A survey on risc-v-based machine learning ecosystem," *Information*, vol. 14, no. 2, p. 64, 2023.

[18] "MISRA." [Online]. Available: https://www.misra.org.uk/

[19] R. Bagnara, A. Bagnara, and P. M. Hill, "The misra c coding standard and its role in the development and analysis of safety- and security-critical embedded software," 2018.

[20] H. Carvalho, P. Zaykov, and A. Ukaye, "Leveraging the hw/sw optimizations and ecosystems that drive the ai revolution," 2022. [Online]. Available: https://arxiv.org/abs/2208.02808

[21] "Embedded Systems with Jetson." [Online]. Available: https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/

[22] "Risc-v cores list." [Online]. Available: https://github.com/riscvarchive/riscv-cores-list

[23] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, "An empirical survey-based study into industry practice in real-time systems," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020, pp. 3–11.

[24] "Risc-v recently ratified extensions." [Online]. Available: https://wiki.riscv.org/display/HOME/Recently+Ratified+Extensions

[25] I. A. Assir, M. E. Iskandarani, H. R. A. Sandid, and M. A. R. Saghir, "Arrow: A risc-v vector accelerator for machine learning inference," 2021.

[26] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," 2017.

[27] "Achieving fp32 accuracy for int8 inference using quantization aware training with nvidia tensorrt." [Online]. Available: https://developer.nvidia.com/blog/achieving-fp32-accuracy-for-int8-inference-using-quantization-aware-training-with-tensorrt/

[28] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," 2021.

[29] L. Mao, "Quantization for neural networks." [Online]. Available: https://leimao.github.io/article/Neural-Networks-Quantization

[30] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," 2013.

[31] F. Li, B. Liu, X. Wang, B. Zhang, and J. Yan, "Ternary weight networks," 2022.

[32] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," 2016.

[33] M. Al-Hami, M. Pietroń, R. Casas, and M. Wielgosz, "Methodologies of compressing a stable performance convolutional neural networks in image classification," *Neural Processing Letters*, vol. 51, 02 2020.

[34] R. Zhao, Y. Hu, J. Dotzel, C. D. Sa, and Z. Zhang, "Improving neural network quantization without retraining using outlier channel splitting," *CoRR*, vol. abs/1901.09504, 2019. [Online]. Available: http://arxiv.org/abs/1901.09504

[35] "Google GeMMlowp." [Online]. Available: https://github.com/google/gemmlowp

[36] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*, G. Lorette, Ed., Université de Rennes 1. La Baule (France): Suvisoft, Oct. 2006, http://www.suvisoft.com. [Online]. Available: https://inria.hal.science/inria-00112631

[37] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, "High-performance low-memory lowering: Gemm-based algorithms for dnn convolution," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 99–106.

[38] "Numpy broadcasting." [Online]. Available: https://numpy.org/doc/stable/user/basics.broadcasting.html

[39] "Sifive: RISC-V vector processing is taking off." [Online]. Available: https://riscv.org/blog/2022/06/risc-v-vector-processing-is-taking-off-sifive/

[40] "RISC-V vector intrinsic documentation." [Online]. Available: https://github.com/riscv-non-isa/rvv-intrinsic-doc

[41] J. K. L. Lee, M. Jamieson, N. Brown, and R. Jesus, "Test-driving risc-v vector hardware for hpc," *ArXiv*, vol. abs/2304.10319, 2023.

[42] "GCC13 changelog." [Online]. Available: https://gcc.gnu.org/gcc-13/changes.html

[43] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Sergio, "Vectorizing posit operations on risc-v for faster deep neural networks: Experiments and comparison with arm sve," *Neural Computing and Applications*, 08 2021.

[44] "SiFive Freedom E SDK." [Online]. Available: https://github.com/sifive/freedom-e-sdk

[45] S. McIntosh-Smith, "Enabling processor design space exploration with simeng," in *ModSim: Workshop on Modeling and Simulation of Systems and Applications*, 2019.

[46] R. A. Fisher and M. Marshall, "Iris data set," *RA Fisher, UC Irvine Machine Learning Repository*, vol. 440, p. 87, 1936.

[47] E. Anderson, "The species problem in iris," *Annals of the Missouri Botanical Garden*, vol. 23, no. 3, pp. 457–509, 1936. [Online]. Available: http://www.jstor.org/stable/2394164

[48] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[49] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[50] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[51] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, "Sparse convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[52] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[53] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 1097–1105.

[54] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017.

Appendix A

Precision Loss Of Operations

This appendix contains an example of the precision loss between a full-precision (Float32) operator in PyTorch, and the integer only implementation of Conv2D using the parameters generated by my pipeline.

Output of PyTorch Conv2D Layer:

```
1.8125 2.5625 2.7500 2.7500 2.0000
2.5625 3.6875 3.8750 3.8750 2.7500
2.7500 3.8750 3.8750 3.8750 2.7500
2.7500 3.8750 3.8750 3.6875 2.5625
2.0000 2.7500 2.7500 2.5625 1.8125

2.0625 2.8125 3.0000 3.0000 2.2500
2.8125 3.9375 4.1250 4.1250 3.0000
3.0000 4.1250 4.1250 4.1250 3.0000
3.0000 4.1250 4.1250 3.9375 2.8125
2.2500 3.0000 3.0000 2.8125 2.0625

2.3125 3.0625 3.2500 3.2500 2.5000
3.0625 4.1875 4.3750 4.3750 3.2500
3.2500 4.3750 4.3750 4.3750 3.2500
3.2500 4.3750 4.3750 4.1875 3.0625
2.5000 3.2500 3.2500 3.0625 2.3125

2.5625 3.3125 3.5000 3.5000 2.7500
3.3125 4.4375 4.6250 4.6250 3.5000
3.5000 4.6250 4.6250 4.6250 3.5000
3.5000 4.6250 4.6250 4.4375 3.3125
2.7500 3.5000 3.5000 3.3125 2.5625
```

Output of generated model Integer-Only Conv operator:

```
1.813726 2.557353 2.756863 2.756863 1.995098
2.557353 3.700000 3.881373 3.881373 2.756863
2.756863 3.881373 3.881373 3.881373 2.756863
2.756863 3.881373 3.881373 3.700000 2.557353
1.995098 2.756863 2.756863 2.557353 1.813726

2.067647 2.811275 3.010784 3.010784 2.249020
2.811275 3.953922 4.135294 4.135294 3.010784
3.010784 4.135294 4.135294 4.135294 3.010784
3.010784 4.135294 4.135294 3.953922 2.811275
2.249020 3.010784 3.010784 2.811275 2.067647
```

```
2.303432 3.065196 3.246569 3.246569 2.502941
3.065196 4.189706 4.389216 4.389216 3.246569
3.246569 4.389216 4.389216 4.389216 3.246569
3.246569 4.389216 4.389216 4.189706 3.065196
2.502941 3.246569 3.246569 3.065196 2.303432

2.557353 3.319118 3.500490 3.500490 2.756863
3.319118 4.443628 4.625000 4.625000 3.500490
3.500490 4.625000 4.625000 4.625000 3.500490
3.500490 4.625000 4.625000 4.443628 3.319118
2.756863 3.500490 3.500490 3.319118 2.557353
```

Average Error is: 0.00412269

Appendix B

RISC-V Vector GeMM

```
enum ERVIL_ERR_T operator_GEMM_vec(tensor *Y, tensor *C, tensor *A, tensor *B,
                                   int64_t fixed_point_rescaler,
                                   uint8_t right_shift) {

  // multiply mxn by nxp to get mxp
  // c_i,j = sum from k = 1..n of a_i,k * b_k,j
  // i = 1..m, j = 1..p
  uint32_t m = 0;
  uint32_t p = 0;
  uint32_t n = 0;
  uint32_t n2 = 0;

  // For tensors with different numbers of dimensions
  // indicies of row/col dimentsion will be different
  // n_dims == 2  --> dims == [R, C]
  // n_dims == 3  --> dims == [Ch, R, C]
  // n_dims == 4  --> dims == [Mb, Ch, R, C]
  // m,p,n are derived from R and C
  uint8_t A_R_index = A->n_dims - 2;
  uint8_t A_C_index = A->n_dims - 1;
  uint8_t B_R_index = B->n_dims - 2;
  uint8_t B_C_index = B->n_dims - 1;

  m = A->dims[A_R_index];
  p = B->dims[B_C_index];
  n = A->dims[A_C_index];

  const int64_t rounding = (right_shift < 1) ? 0 : (1 << (right_shift - 1));
  const int32_t TERM4 = p * A->q_zp * B->q_zp;

  for (uint32_t i = 0; i < m; i++) {
    for (uint32_t j = 0; j < p; j++) {
      size_t vlmax = __riscv_vsetvlmax_e8m1();
      vint32m1_t vec_zero = __riscv_vmv_v_x_i32m1(0, vlmax);

      int32_t TERM1 = 0;
      int32_t TERM2 = 0;
      int32_t TERM3 = 0;
      vint32m4_t TERM1_VEC = __riscv_vmv_v_x_i32m4(0, vlmax);
      vint32m4_t TERM2_VEC = __riscv_vmv_v_x_i32m4(0, vlmax);
      vint32m4_t TERM3_VEC = __riscv_vmv_v_x_i32m4(0, vlmax);
```

```
      uint32_t k = n;
      uint8_t *ptr_a = ((uint8_t *)A->data) + (i * m);
      uint8_t *ptr_b = ((uint8_t *)B->data) + j;

      for (size_t vl; k > 0; k -= vl, ptr_a += vl, ptr_b += vl) {
        vl = __riscv_vsetvl_e8m1(k);

        vuint8m1_t vec_a = __riscv_vle8_v_u8m1(ptr_a, vl);
        vuint8m1_t vec_b = __riscv_vlse8_v_u8m1(
            ptr_b, n,
            vl); // strided load with stride of row-size - this reads columns
        vuint32m4_t vec_a_extended = __riscv_vzext_vf4_u32m4(vec_a, vl);
        vuint32m4_t vec_b_extended = __riscv_vzext_vf4_u32m4(vec_b, vl);
        vint32m4_t vec_a_as_i32 =
            __riscv_vreinterpret_v_u32m4_i32m4(vec_a_extended);
        vint32m4_t vec_b_as_i32 =
            __riscv_vreinterpret_v_u32m4_i32m4(vec_b_extended);

        TERM1_VEC = __riscv_vmacc_vv_i32m4_tu(TERM1_VEC, vec_a_as_i32,
                                              vec_b_as_i32, vl);
        TERM2_VEC = __riscv_vadd_vv_i32m4(TERM2_VEC, vec_a_as_i32, vl);
        TERM3_VEC = __riscv_vadd_vv_i32m4(TERM3_VEC, vec_b_as_i32, vl);
      }

      vint32m1_t TERM1_SUM = __riscv_vredsum_vs_i32m4_i32m1(
          TERM1_VEC, vec_zero, __riscv_vsetvl_e32m4(n));
      vint32m1_t TERM2_SUM = __riscv_vredsum_vs_i32m4_i32m1(
          TERM2_VEC, vec_zero, __riscv_vsetvl_e32m4(n));
      vint32m1_t TERM3_SUM = __riscv_vredsum_vs_i32m4_i32m1(
          TERM3_VEC, vec_zero, __riscv_vsetvl_e32m4(n));

      TERM1 = __riscv_vmv_x_s_i32m1_i32(TERM1_SUM);
      TERM2 = __riscv_vmv_x_s_i32m1_i32(TERM2_SUM) * B->q_zp;
      TERM3 = __riscv_vmv_x_s_i32m1_i32(TERM3_SUM) * A->q_zp;

      int32_t ACC = (TERM1 - TERM2 - TERM3 + TERM4);
      int32_t BIAS_TERM = 0;

      // Will not always have a bias vector, so check if C is null before trying
      // to read it
      if (C) {
        tensor_at(&BIAS_TERM, C, 0, 0, j, i, int32_t);
      }

      int64_t rescaled = (int64_t)((fixed_point_rescaler *
                                    ((int64_t)ACC + (int64_t)BIAS_TERM)));
      int64_t y_q_64 = (int64_t)Y->q_zp +
                       ((int64_t)(rescaled + rounding) >> (right_shift + 31));
      uint8_t y_q = saturating_cast_i64_to_u8(y_q_64);
      tensor_set_at(&y_q, Y, uint8_t, 0, 0, j, i);
    }
  }

  return SUCCESS;
}
```

Appendix C

RISC-V Vector Assembly Comparison

A simple multiply-and-accumulate loop, accumulating products of UINT8 values into a INT32 value:

```
int byte_mac(unsigned char a[], unsigned char b[], int len) {
    int32_t sum = 0;
    for (int i = 0; i < len; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

Auto-vectorised assembly generated by clang 15 rv64gc (-O3 -march=rv64gv1p0):

```
byte_mac(unsigned char*, unsigned char*, int): # @byte_mac(unsigned char*, unsigned
blez a2, .LBB0_3
csrr a3, vlenb
srli a3, a3, 1
bgeu a2, a3, .LBB0_4
li a4, 0
li a3, 0
j .LBB0_7
.LBB0_3:
li a0, 0
ret
.LBB0_4:
addi a5, a3, -1
and a5, a2, a5
sub a4, a2, a5
vsetvli a6, zero, e32, m2, ta, ma
vmv.v.i v8, 0
mv a6, a4
mv a7, a1
mv t0, a0
.LBB0_5:
vle8.v v10, (t0)
vle8.v v11, (a7)
vzext.vf4 v12, v10
vzext.vf4 v14, v11
vmacc.vv v8, v14, v12
add t0, t0, a3
sub a6, a6, a3
add a7, a7, a3
```

```
bnez a6, .LBB0_5
vmv.s.x v10, zero
vredsum.vs v8, v8, v10
vmv.x.s a3, v8
beqz a5, .LBB0_9
.LBB0_7:
add a1, a1, a4
add a0, a0, a4
sub a2, a2, a4
.LBB0_8:
lbu a4, 0(a0)
lbu a5, 0(a1)
mul a4, a5, a4
addw a3, a4, a3
addi a1, a1, 1
addi a2, a2, -1
addi a0, a0, 1
bnez a2, .LBB0_8
.LBB0_9:
mv a0, a3
ret
```

Version of the code vectorised by hand using the RVV intrinsics:

```
int byte_mac_vec(unsigned char *a, unsigned char *b, int len) {
    size_t vlmax = __riscv_vsetvlmax_e8m1();
    vint32m4_t vec_s = __riscv_vmv_v_x_i32m4(0, vlmax);
    vint32m1_t vec_zero = __riscv_vmv_v_x_i32m1(0, vlmax);

    int k = len;
    for (size_t vl; k > 0; k -= vl, a += vl, b += vl) {
        vl = __riscv_vsetvl_e8m1(k);
        vuint8m1_t a8s = __riscv_vle8_v_u8m1(a, vl);
        vuint8m1_t b8s = __riscv_vle8_v_u8m1(b, vl);
        vuint32m4_t a8s_extended = __riscv_vzext_vf4_u32m4(a8s, vl);
        vuint32m4_t b8s_extended = __riscv_vzext_vf4_u32m4(b8s, vl);
        vint32m4_t a8s_as_i32 =
            __riscv_vreinterpret_v_u32m4_i32m4(a8s_extended);
        vint32m4_t b8s_as_i32 =
            __riscv_vreinterpret_v_u32m4_i32m4(b8s_extended);
        vec_s = __riscv_vmacc_vv_i32m4_tu(vec_s, a8s_as_i32, b8s_as_i32, vl);
    }

    vint32m1_t vec_sum =
        __riscv_vredsum_vs_i32m4_i32m1(vec_s,
        vec_zero, __riscv_vsetvl_e32m4(len));

    int sum = __riscv_vmv_x_s_i32m1_i32(vec_sum);
    return sum;
}
```

Assembly generated by clang 15 rv64gcv (-O3 -march=rv64gv1p0):

```
byte_mac_vec(unsigned char*, unsigned char*, int): # @byte_mac_vec(unsigned char*, u
vsetvli a3, zero, e32, m4, ta, ma
vmv.v.i v8, 0
blez a2, .LBB0_3
mv a4, a2
.LBB0_2:
slli a5, a4, 32
```

```
srli a5, a5, 32
vsetvli a5, a5, e32, m4, ta, ma
vle8.v v12, (a0)
vle8.v v13, (a1)
vzext.vf4 v16, v12
vzext.vf4 v20, v13
vsetvli zero, zero, e32, m4, tu, ma
vmacc.vv v8, v16, v20
subw a4, a4, a5
add a0, a0, a5
add a1, a1, a5
bgtz a4, .LBB0_2
.LBB0_3:
vsetvli zero, a3, e32, m1, ta, ma
vmv.v.i v12, 0
vsetvli zero, a2, e32, m4, ta, ma
vredsum.vs v8, v8, v12
vmv.x.s a0, v8
ret
```