Auto-tuning RISC-V Vectorized convolutions in oneDNN

Master's thesis in Computer science and engineering

SAMUEL JÖNSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Auto-tuning RISC-V Vectorized convolutions in oneDNN

SAMUEL JÖNSSON

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Auto-tuning RISC-V Vectorized convolutions in oneDNN
SAMUEL JÖNSSON

Supervisor: Miquel Pericàs, Computer Science and Engineering
Examiner: Miquel Pericàs, Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Auto-tuning RISC-V Vectorized convolutions in oneDNN
SAMUEL JÖNSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Convolutional Neural Networks are an essential part in deep learning tasks such as image recognition and object detection. The open-source RISC-V ISA with its vector extension offers a great opportunity for optimizing convolution algorithms on high-performance systems. The objective of this thesis was to implement previously vectorized convolution algorithms in a deep learning library called oneDNN and develop a state-of-the-art auto-tuner to further optimize the performance. The algorithms im2col+GEMM were executed and tested on the QEMU emulator running RISC-V Vector Extension 1.0. Four different machine learning models were evaluated for their accuracy and predictive power to auto-tune the algorithms. The results show significant improvement in both execution time and instruction efficiency compared to naive implementations and implementations that didn't use auto-tuning. The thesis concludes that using a random forest model to auto-tune the convolution algorithm configurations generates the most accurate predictions. Furthermore, the thesis demonstrates the effectiveness of utilizing RISC-V Vector instructions and auto-tuning to optimize im2col+GEMM in oneDNN.

Keywords: Convolutions, Neural Networks, Auto-tuning, Vectorized instructions, QEMU, RISC-V, Thesis, Computer Engineering, Computer Science

Acknowledgements

I would like to thank my supervisor and examiner Miquel Pericàs for the continuous support and guidance during the project. I would also like to thank Nikela Papadopoulou and Sonia Rani Gupta for their work on vectorizing the algorithms used in this thesis and giving the project a good starting point. Finally, I would like to thank Emelie Blade and Samuel Kontola for their aid in improving this thesis report through their constructive criticism and comments.

Samuel Jönsson, Gothenburg, 2024-06-19

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Context . . . 1
1.2 Problem Statement . . . 3
1.3 Related Work . . . 4

2 Background **7**
2.1 Machine Learning and algorithms . . . 7
2.1.1 im2col+GEMM . . . 7
2.1.2 Winograd . . . 8
2.1.3 Neural Networks . . . 11
2.1.4 Artificial Neural Networks (ANN) . . . 12
2.1.5 Random Forest Classification . . . 12
2.1.6 Linear Regression . . . 12
2.1.7 Support Vector Machine (SVM) . . . 13
2.1.8 VGG16 . . . 13
2.2 RISC-V . . . 15
2.2.1 Vector Extension . . . 15
2.2.2 RISC-V JIT . . . 15
2.2.3 QEMU Emulator . . . 17
2.3 RISC-V For Machine Learning . . . 18

3 Methods **19**
3.1 Working Environment . . . 19
3.1.1 Compiler . . . 19
3.1.2 QEMU . . . 19
3.1.3 oneDNN . . . 20
3.2 Network Model . . . 21
3.3 RISC-V JIT . . . 21
3.3.1 JIT versus compiler intrinsics . . . 21
3.3.2 Converting previous work to oneDNN RISC-V JIT . . . 23
3.3.3 New instructions . . . 23
3.4 Auto-tuning . . . 24

3.4.1 im2col+GEMM for VGG16 24
3.4.2 Network agnostic auto-tuning 25
3.4.3 Models 26

4 Results 27
4.1 Auto-tuning VGG16 27
4.1.1 ANN model 28
4.1.2 Linear Regression model 29
4.1.3 SVM model 31
4.1.4 Random Forest model 33
4.2 Network agnostic auto-tuner 36
4.2.1 Switching N to actual values 36
4.2.1.1 ANN-N model 36
4.2.1.2 Linear Regression-N model 38
4.2.1.3 SVM-N model 40
4.2.1.4 Random Forest-N model 42
4.2.2 Modeling all 39 parameters 44
4.2.2.1 ANN-39 model 44
4.2.2.2 Linear Regression-39 model 46
4.2.2.3 SVM-39 model 48
4.2.2.4 Random Forest-39 model 50

5 Discussion 53
5.1 Auto-tuning models 53
5.1.1 ANN 53
5.1.2 Linear Regression & SVM 54
5.1.3 Random Forest 55
5.2 Comparing with non auto-tuned executions 55
5.2.1 Network-agnostic models 55
5.2.2 Blocking factor vs actual N value 56
5.2.3 Blocking factor vs 39 additional parameters 57
5.2.4 Network-agnostic models summary 58
5.3 Winograd 59

6 Conclusion 61
6.1 Continuing with oneDNN 61
6.2 Auto-tuning model success 61
6.3 Future Work 62

Bibliography 63

A Appendix 1 I

List of Figures

2.1 Image demonstrating im2col 7
2.2 Image showing the structure of a neural network 11
2.3 Image showing the structure of VGG16 14
2.4 RVJIT implementation of vslideup_vx 17
2.5 Instruction alias for use by the programmer 17

4.1 Results from ANN model prediction 28
4.2 Results from Linear Regression model prediction 30
4.3 Results from SVM model prediction 32
4.4 Results from Random Forest model prediction 34
4.5 Results from ANN-N prediction 37
4.6 Results from Linear Regression-N prediction 39
4.7 Results from SVM-N prediction 41
4.8 Results from Random Forest-N prediction 43
4.9 Results from ANN-39 prediction 45
4.10 Results from Linear Regression-39 model prediction 47
4.11 Results from SVM-39 model prediction 49
4.12 Results from Random Forest-39 model prediction 51

List of Tables

2.1 Table 1 from Simonyan and Zisserman showing the different configurations the researchers implemented [28]. 14

3.1 First attempt at defining a parameter space 24
3.2 Second attempt at defining a parameter space 24

4.1 ANN model configuration predictions 28
4.2 ANN model layer instruction count predictions ratios (L0 to L5) . . . 29
4.3 ANN model layer instruction count predictions ratios (L6 to L12) . . 29
4.4 Linear Regression configuration predictions 30
4.5 Linear Regression model layer instruction count predictions ratios (L0 to L5) 31
4.6 Linear Regression model layer instruction count predictions ratios (L6 to L12) 31
4.7 SVM configuration predictions 32
4.8 SVM model layer instruction count predictions ratios (L0 to L5) . . . 33
4.9 SVM model layer instruction count predictions ratios (L6 to L12) . . 33
4.10 Random Forest configuration predictions 34
4.11 Random Forest layer instruction count predictions ratios (L0 to L5) . 35
4.12 Random Forest layer instruction count predictions ratios (L6 to L12) 35
4.13 Instruction count ratios between no unrolling or blocking and best predicted configuration (L0 to L5) 35
4.14 Instruction count ratios between no unrolling or blocking and best predicted configuration (L6 to L12) 36
4.15 ANN-N configuration predictions 36
4.16 ANN-N layer instruction count predictions ratios (L0 to L5) 37
4.17 ANN-N layer instruction count predictions ratios (L6 to L12) 38
4.18 Linear Regression-N configuration predictions 38
4.19 Linear Regression-N layer instruction count predictions ratios (L0 to L5) 39
4.20 Linear Regression-N layer instruction count predictions ratios (L6 to L12) 40
4.21 SVM-N configuration predictions 40
4.22 SVM-N layer instruction count predictions ratios (L0 to L5) 41
4.23 SVM-N layer instruction count predictions ratios (L6 to L12) 42
4.24 Random Forest-N configuration predictions 42

4.25 Random Forest-N layer instruction count predictions ratios (L0 to L5) 43
4.26 Random Forest-N layer instruction count predictions ratios (L6 to L12) 44
4.27 ANN-39 configuration predictions 44
4.28 ANN-39 layer instruction count predictions ratios (L0 to L5) 45
4.29 ANN-39 layer instruction count predictions ratios (L6 to L12) 46
4.30 Linear Regression-39 configuration predictions 46
4.31 Linear Regression-39 model layer instruction count predictions ratios (L0 to L5) 47
4.32 Linear Regression-39 model layer instruction count predictions ratios (L6 to L12) 48
4.33 SVM-39 configuration predictions 48
4.34 SVM-39 model layer instruction count predictions ratios (L0 to L5) . 49
4.35 SVM-39 model layer instruction count predictions ratios (L6 to L12) . 50
4.36 Random Forest-39 configuration predictions 50
4.37 Random Forest-39 layer instruction count predictions ratios (L0 to L5) 51
4.38 Random Forest-39 layer instruction count predictions ratios (L6 to L12) 52

5.1 Average ratio between prediction and actual instruction count for blocking factor auto-tuners, N value auto-tuners, and auto-tuners using the 39 additional layer parameters (L0 to L4) 56

1
Introduction

Convolutional Neural Networks (CNNs) are one of the most used types of neural networks (NNs) in the field of Deep Learning (DL), often used for object detection or speech processing. The main benefit of using CNNs over other types of NNs is their weight sharing, which substantially decreases the amount of neurons needed in the network [1]. This reduces the computational load of the network, decreases the memory size, and makes the network less likely to overfit to noise in the data. The architecture of CNNs lends them to be great at assigning importance to various features in input images through the convolutional layers [2]. Furthermore, by connecting multiple convolutional layers more complex features can be detected.

Additionally, the parallel nature of the convolutional kernels make them attractive to try to adapt for use in processors capable of Single Instruction Multiple Data (SIMD) parallelism from an initial Single Instruction Single Data format (SISD) [3]. In combination with the fact that many CNN-based models have an increasing computation cost while requiring low-latency and portability or ability to be used on embedded systems means that running them on processors using the open source Instruction Set Architecture (ISA) RISC-V Vector Extensions (RISC-V "V") have great potential for high-performance convolutions [4], [5]. The RISC-V ISA's research potential is also increased along with its adoption and growing usage as can be seen by for example the adoption of RISC-V by the European Processor Initiative and the European PILOT project [6], [7].

1.1 Context

CNNs and other types of neural networks are often implemented in optimized libraries. The library that this thesis will use and work with is oneDNN [8]. It is an open-source Deep Neural Network (DNN) library that is used as the building block for many DL applications, such as PyTorch and Tensorflow. OneDNN builds on a concept called primitives. These primitives can store states, for example a convolution primitive can store tensor shapes which enables precomputing parameters and pre-generating code to be tailored for a specific operation. It also has just-in-time (JIT) compilation to optimize code based on input parameters, such as optimizing tile size and loop reordering. While oneDNN has been both manually optimized and has JIT optimization, it is still challenging to fully adapt to every given tensor shape [9].

The problem of auto-tuning these parameters has been and still is the focus of research and development. Auto-tuning being the act of selecting and optimizing, or tuning, parameters and settings to make a program work more efficiently on a piece of hardware. For example AutoTVM developed for NVIDIA GPUs and ARM CPUs using the TVM domain-specific compiler [10]. AutoTVM is used to generate optimized code and it eventually finds the optimal combination of parameters by brute forcing the solution. Although the optimal solution is guaranteed to be found, the number of tested configurations grow exponentially with the size of the convolutions [11].

Furthermore, linear algebra libraries have been developed for various architectures such as GPUs, multicore processors with SIMD, vector processors. This lends to optimized routines for use in CNNs however these suffer from being hardware-specific. Meaning they have great performance on certain pieces of hardware but if they were to be run on any other type of hardware their performance would degrade considerably. RISC-V, as an open standard, offers an alternative to proprietary architectures. Which is a concern of for example the European PILOT project which aims to have an open source software and hardware HPC system [7].

Previously, there has been research done at Chalmers on the implementation of convolutions on vector architecture, both on RISC-V "V" and ARM-SVE using a framework called Darknet [12], [4]. They showed that Winograd convolutions, a convolution method to decrease the amount of multiplication operations in return of increasing the amount of addition operations, benefited from using longer vectors. To do this, they utilized the Vector extension of RISC-V and testing different vector sizes [12]. For the paper concerning im2col+GEMM (Image to column + General Matrix Multiply), it was improved by vectorizing the kernels of the convolutional layer in Darknet as well as doing manual optimizations. Examples of implemented optimizations are using intrinsic instructions to vectorize, using vector memory registers, loop reordering and unrolling, block tiling, and memory prefetching [4]. However, continuing working with Darknet is not as beneficial as working with oneDNN due to the low impact the research would have in comparison as frameworks such as PyTorch and Tensorflow gain popularity and optimize their backends using oneDNN. For instance, the European PILOT project has adopted oneDNN as one of the components in its software stack [7].

In addition to this, a paper on current state-of-the-art proposals for computing direct convolutions on SIMD capable CPUs shows how frequent cache misses lower performance [3]. In comparison to the previous mentioned project, direct convolution avoids the memory overhead of im2col transformations. Their goal is to optimize cache reuse by reordering loops and utilize different parallelization strategies to make better use of the caches [3]. The researchers propose two algorithmic approaches: *Bounded Direct Convolution* throttles the amount of computations exposed to mitigate cache misses, *Multi-Block Direct Convolution* which focuses on improving the memory access pattern [3]. In comparison to this direct convolution research, this project will continue working on im2col+GEMM and Winograd.

1.2 Problem Statement

The main problems in converting the already developed functions will be:

- How to utilize oneDNN? What are the best ways to work with the library?
- How to use the RISC-V JIT that has been developed for oneDNN?
- How to test and evaluate the performances of the auto-tuned algorithms in comparison to naive implementations?

For this thesis a neural network called VGG16 will be used to test the implementation's performance. VGG16 has been used previously and is a well-known network model which will simplify the development process. VGG16, like many neural networks, has a number of layers and changes the amount of inputs and outputs in those layers to be able to recognize more complex features. For the purposes of convolution, this changes the input and output sizes of the data that is to be convolved.

Different input dimensions, i.e., different layers in the network with different widths, heights, and channels, may perform better or worse than others due to better memory utilization or better throughput. Optimizations like loop unrolling, i.e., executing more work per iteration and reducing the total amount of iterations, may lead to faster execution due to reducing the amount of branching. There are multiple factors that may impact the performance of the convolution operation on a certain layer. Thus, the goal of the auto-tuner is to find the best configuration for a specific layer so that the whole network is executed in a fast and efficient manner.

A problem that stems from this is the strategy of how to auto-tune. In general there are two ways to auto-tune, offline auto-tuning or online auto-tuning. This thesis will use offline auto-tuning, meaning utilizing historical data to train an analytical model to optimize the variables, to reduce the development complexity in comparison to creating an online auto-tuner. Theoretically, an offline auto-tuner could do a full exhaustive search of the parameter space to find the optimal solution. The feasibility of such a search depends on the number of parameters that are used and on possible future proofing or extendability for the auto-tuner. In other words, adding an extra parameter to take into consideration when auto-tuning increases the search space exponentially. This can lead to an unfeasible amount of combinations and configurations to test and therefore it would be preferable if it is possible to use another strategy.

Given this information, the goals of this thesis are to implement vectorized im2col +GEMM and Winograd convolution algorithms in oneDNN for RISC-V processors and to develop a state-of-the-art method to auto-tune the given parameters that gives consistent and accurate predictions of performance without having to do an exhaustive search.

1.3 Related Work

As mentioned, auto-tuning is not a new strategy to improve performance. In this section three state-of-the-art approaches are presented.

Researchers at the Norwegian University of Science and Technology used autotuning to improve the performance of OpenCL [13]. OpenCL is a framework and open standard for parallel programming, one of it's most important features being that it enables cross-platform applications [14]. However, the performance is not necessarily the same on all platforms the code can run on. To showcase this the researchers ran a benchmark on three different devices, an Intel i7 3770 CPU, an Nvidia K40 GPU and an AMD Radeon HD 7970 GPU [13]. In their example they showcased that using the best Nvidia configuration on the Intel CPU caused a slowdown by a factor of 17.1 . As seen from this, tuning programs to the hardware they will run on can have a great impact on their performances.

To auto-tune their benchmarks they first picked random configurations of a space of possible implementations. They then measured the performance of these random configurations which gave them a model to train on. This model was then used to predict new configurations and the best of these predictions were then measured to find which configuration was actually the best [13]. To predict and determine this the researchers used artificial neural networks (ANN). They state that the reasons for choosing ANNs were that they have good predictive power and can handle arbitrary functions. However, they are difficult to interpret and it is hard to gain a deeper insight into how the parameters interact [13].

Another related work in the domain of auto-tuning is the Kernel Tuning Toolkit (KTT) by HiPerCoRe [15], [16]. It is a framework to tune OpenCL, CUDA kernels and GLSL compute shaders. It can do both offline auto-tuning and dynamic auto-tuning [16]. Offline tuning is done outside of the application, it can find the most demanding parameters and export tuned values to the build system. Dynamic tuning is performed during runtime and can be executed at different times in the program. For example, if the program is being executed on a new hardware device or if certain performance characteristics are changing. The auto-tuner tests different configurations and sends the best kernel it finds to the application to be executed. It does not return the kernels it tests. It avoids side effects by replicating input and output arrays so that it avoids modifying the data that will be used after tuning. Some parameters KTT tunes are tile size, local memory caching, memory tiling, and loop unrolling [16]. KTTs offline tuning uses an exhaustive search to find the best performance for the devices they tested. This stands in contrast to the auto-tuning mentioned first which gather a few random samples to then predict the rest of the search space.

Another auto-tuning framework, called GPTune, is built on multitask and transfer learnings to tune using Bayesian optimization [17], [18]. Their method defines a space class which has three different spaces. The input space consists of the application targets and each point in the input space represents one instance of a problem. The parameter space defines the parameters that can and will be tuned. In this

space, each point represents a combination of parameters and the goal of the autotuner is to find the point that minimizes the objective function of the application in question. Finally there is the output space which defines the results or the objective of the application. Some examples the authors give here are runtime and energy consumption [17]. To build their model they utilize Bayesian optimization as mentioned. This means that they have a surrogate model of the real objective function which is cheaper to evaluate and therefore faster to predict. Especially in comparison to an exhaustive search that goes through the whole search space, the Bayesian optimization can go through the search space more efficiently [17].

2

Background

2.1 Machine Learning and algorithms

This section begins with breaking down some of the algorithms that are used in the thesis and continues with an overview of the machine learning models that are utilized throughout the thesis.

2.1.1 im2col+GEMM

Im2col+GEMM are well-known methods used in convolutional networks. Im2col takes a three dimensional image (or multidimensional tensor) and transforms it into a two dimensional matrix, flattening the image into a column vector - hence the name. This flattened matrix can then be used in GEMM [19]. A complex multidimensional problem of convolution then becomes a more simple matrix multiplication. This technique is commonly used in Basic Linear Algebra Subprograms (BLAS) to efficiently handle complex convolutions. The downside of this is that the process of flattening the input image increases memory overhead and calculations done, especially if the input images are large. RISC-V Vectorized versions of im2col+GEMM have been developed in previous work and were used in this project [4].

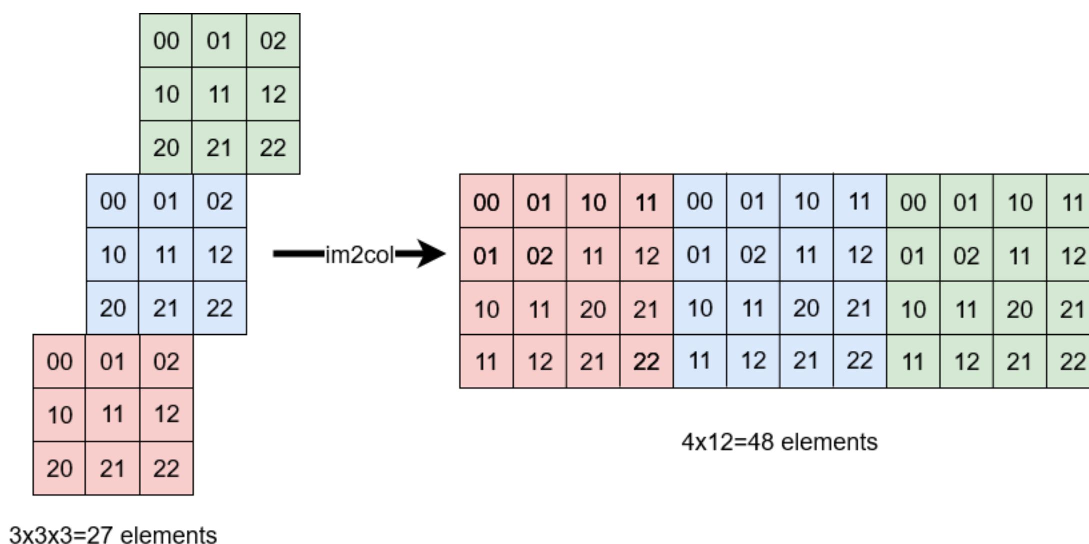

Figure 2.1: Image demonstrating im2col

Given a convolutional layer with a kernel size of $k \times k$, input of dimensions $h \times w \times c$ GEMM multiplies an input weight matrix and input matrix of sizes $M \times K$ and $K \times N$ respectively. Where $M = n$, $K = k \times k \times c$, $N = h \times w$ (n is the number of filters, h is the input height, w is the input width, c is the amount of input channels). Figure 2.1 demonstrates a three-dimensional input that is flattened to a two-dimensional matrix. Algorithm 1, 2, 3, and 4 show pseudo-code for im2col+GEMM.

Algorithm 1 im2col helper function

```
1: Function im2col_get_pixel(im, height, width, channels, row, col, channel,
   pad)
2:    row ← row − pad
3:    col ← col − pad
4: if row < 0 or col < 0 or row ≥ height or col ≥ width then
5:    return 0
6: end if
7: return im[col + width * (row + height * channel)]
```

Algorithm 2 Naive implementation of im2col

```
1: height_col ← (height + 2 * pad - ksize) / stride + 1
2: width_col ← (width + 2 * pad - ksize) / stride + 1
3: channels_col ← channels * ksize * ksize
4: size ← channels_col * height_col * width_col
5: for c ← 0 to channels_col - 1 do
6:    w_offset ← c % ksize
7:    h_offset ← (c / ksize) % ksize
8:    c_im ← c / ksize / ksize
9:    for h ← 0 to height_col - 1 do
10:      for w ← 0 to width_col - 1 do
11:         im_row ← h_offset + h * stride
12:         im_col ← w_offset + w * stride
13:         col_index ← (c * height_col + h) * width_col + w
14:         data_col[col_index] ← im2col_get_pixel(data_im, height,
            width, channels, im_row, im_col, c_im, pad)
15:      end for
16:   end for
17: end for
```

2.1.2 Winograd

Winograd convolution is an algorithm that reduces the amount of multiplications and increases the amount of additions done for convolution [20].

In an example using a filter of size 3 and output of size 2, denoted F(2,3), Winograd transformed the standard algorithm

$$Y = A^T[(Gg) \odot (B^T d)]$$

Algorithm 3 Naive implementation of GEMM

```
1: for i ← 0 to M − 1 do
2:   for k ← 0 to K − 1 do
3:     tmp ← alpha × A[i, k]
4:     for j ← 0 to N − 1 do
5:       C[i][j] ← C[i][j] + tmp × B[k][j]
6:     end for
7:   end for
8: end for
```

Algorithm 4 Vectorized 3-loop implementation of GEMM

```
1: for j ← 0 to N − 1 do
2:   gvl ← vsetvl(N − j) // 'granted vector length'
3:   for i ← 0 to M − 1 by U do
4:     VC[i : i + U] ← C[i : i + U, j : j + gvl]
5:     for k ← 0 to K − 1 do
6:       VB ← B[k, j : j + gvl]
7:       for it ← 0 to U − 1 do
8:         tmp ← alpha × A[it, k]
9:         Vtmp ← tmp
10:        VC[it] ← vfmacc(VC[it], Vtmp, VB)
11:      end for
12:    end for
13:    C[i : i + U, j : j + gvl] ← VC[i : i + U]
14:  end for
15:  j ← j + gvl
16: end for
```

to

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

where

$$m_1 = (d_0 - d_2)g_0,\ m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$

$$m_4 = (d_1 - d_3)g_2,\ m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

What Winograd documented was a method that uses 16 multiplications compared to the standard of 36 for that scenario [20]. The benefit of this is that multiplications are often more computationally intensive than additions and thus by converting the convolution to use more additions than multiplications the computational cost would be lowered. However, to convert the convolution to use the Winograd convolution requires transforming the data which does increase the memory usage of the program.

Similarly to im2col+GEMM, RISC-V Vectorized versions of Winograd were implemented in previous work [12]. In algorithm 5 a code snippet of tuple multiplication utilizing the slideup RISC-V V instruction developed previously is showcased.

Algorithm 5 Code snippet from Winograd tuple multiplication using slideup [12]

```
1: ind ← 0
2: VL ← get vector length
3: iteration ← num/64
4: for itr ← 0 to num − 1 do
5:    gvl ← vsetvl(num - itr) // 'granted vector length'
6:    index_vec ← index[0:gvl]
7:    for k ← 0 to iteration-1 do
8:       for j ← 0 to inputchannels-1 do
9:          b0_vec ← B[itr + (j × VL)]
10:         for i ← 0 to 63 by 4 do
11:            a0_vec ← load(A[i + (64 × k) + (j × VL)])
12:            for ind ← 0 to gvl/8-1 do
13:               a0_vec ← slideup(a0_vec, 4 × ind)
14:            end for
15:            acc_vec[i] ← vfmacc(acc_vec[i], a0_vec, b0_vec)
16:         end for
17:      end for
18:      // Store results in the resultant matrix in a contiguous way
19:   end for
20:   itr ← itr + gvl
21: end for
```

2.1.3 Neural Networks

Neural Networks are a programming paradigm inspired by the way the brain works. Traditional software can be seen as rule-based where the programmer sets the logic and rules for how the program should run given certain inputs or environments [21]. Algorithms such as im2col mentioned earlier is an example of this. Given an input the program follows a set path that has a deterministic output.

A neural network, as the name suggests, consists of neurons that are connected and to each other in multiple layers to form a complex structure, showcased in Figure 2.2. There is an input layer, multiple hidden layers, and an output layer [21]. The input and output layers are exposed to the outside, e.g., the input layer takes an input from the user and the output layer gives the user a result. The hidden layers, give and receive information and signals to and from other neurons in the network, not the user. These neurons and layers are interconnected and weighted to find more complex patterns and connections in the input data [22].

In a similar manner to how we can receive and learn new information and with that new information take other more informed decisions, a neural network can adapt to the information it receives and adapts its decision making to better reflect the reality it is given. This is an important point, a neural network may be able to find complex structures in the data it receives, structures and relations that a person may not be able to find. However, if the data itself is biased the network will not magically find an objective truth. Furthermore, if the training data set is too small, a neural network may overfit and generalize patterns that are not representative for the whole problem [21]. It is up to the programmer to make sure that the training data has enough depth and breadth to represent the problem in question.

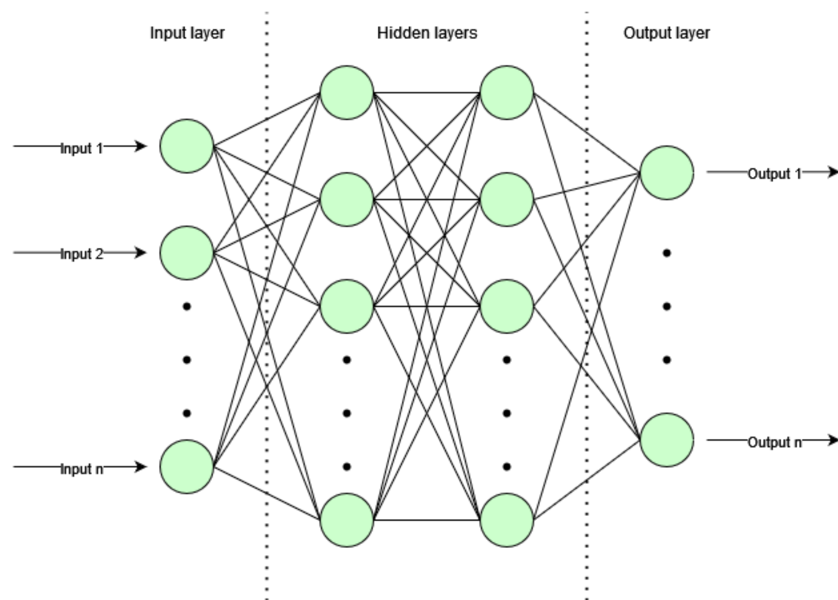

Figure 2.2: Image showing the structure of a neural network

Some use-cases for neural networks include image recognition, targeted advertise-

ments, financial predictions. For image recognition, convolutional neural networks are often used. These are neural networks that extract features in an input image by performing filtering, or convolutions. One convolutional neural network that is used in this thesis is described in 2.1.8.

2.1.4 Artificial Neural Networks (ANN)

Oftentimes, neural networks refers to artificial ANNs [21]. This thesis utilizes supervised ANNs to auto-tune the convolutional layers. This means that the input data used to train the model is labeled and the goal of the network is to match and predict the input data and the output data. The neurons in the network are simplified models of real neurons that activate an output signal when a strong enough input signal is given to it [21]. One key aspect of ANNs is what activation functions to use. Activation functions are used to introduce non-linearity and enable finding more complex features. For example, one may use a sigmoid function, ReLU, and tanh [23], [24]. Without the use of activation functions the ANN would only be able to find linear patterns. Essentially removing the benefit of using multiple hidden layers. A benefit of ANNs is that they can achieve good predictions and handle arbitrary functions. A drawback is their black box nature making it difficult to decipher causalities [13].

2.1.5 Random Forest Classification

Random forest classifications are, like neural networks, a type of machine learning. They are however very different in their structure and methodology. Random forests combine multiple smaller decision trees and takes the average of them to get a prediction. They are easy to understand and in comparison to neural networks, they avoid having a black box nature which gives the random forest a more transparent classification.

A single decision tree may overfit to the training data but by combining the results of multiple trees the random forest mitigates the risk of overfitting. In fact, as the number of trees used increases the confidence of the classification increases and the generalization error decreases [25]. In this thesis, random forest classification is used to both have a comparison method to compare the results of the ANN with and to investigate if this less complex method can be used to auto-tune the convolutional layers.

2.1.6 Linear Regression

Linear regressions are a common method to predict a variable's values that depend on other variables' values. The method estimates a linear equation to the data collected and fits a line or surface to predict data.

In linear regression the variable that we predict is called the dependent variable and the other values that are used for the prediction are called independent variables [26]. An example of this is predicting housing prices. We can have the independent

variables of the age of the house, the size of the house, and the distance to the city center. Then based on these three we could predict the dependent variable of the house price. Linear regression sets up an equation and predicts the coefficients for:

$$\text{Price} = \beta_0 + \beta_1\text{age} + \beta_2\text{area} + \beta_3\text{distance}$$

We can now use linear regression to predict the price for a new house that we haven't seen before by plugging in the data for that house in the equation. For the auto-tuner in this thesis, if the data has a linear relation then linear regression may work well.

2.1.7 Support Vector Machine (SVM)

If the data lacks a linear relation, or even if it is linear, a SVM may perform better than linear regression. The idea behind SVM is to find a hyperplane in an N-dimensional space. N being the total amount of features we are using for classification. SVMs are very effective in high-dimensional spaces meaning when we have lots of features we are looking at in the dataset.

One of the key advantages of SVMs is their robustness to overfitting, especially in high-dimensional space. This is achieved through the regularization parameter, which controls the trade-off between achieving a low training error and a low testing error, thus ensuring better generalization to unseen data.

- **Linear SVM**: This is used when the data is linearly separable, meaning it can be separated into categories or classes with a single straight line or plane [27].
- **Non-Linear SVM**: When the data is not linearly separable, kernel functions such as RBF or polynomial are used to transform the data into a higher dimension where a hyperplane can separate the classes [27].

2.1.8 VGG16

VGG16 is a convolutional network model that is relatively simple in its implementation with a high accuracy proposed by Karen Simonyan and Andrew Zisserman [28]. They devised multiple configurations that all share the same building blocks but have different depths.

This thesis implements the D variation. It consists of 13 convolutional layers, 5 max-pooling layers, 3 fully connected layers. Every layer except the pooling layers also has a ReLu activation function. The ReLu layers introduce non-linearity to allow the model to capture more complex patterns. Each convolutional layer uses a 3x3 kernel size and a stride of 1 pixel. The pooling layers use a 2x2 kernel size and a stride of 2 pixels [28]. The convolutional layers are the core of the network and perform the convolution operations that gather features of the input. They are also the focus of this thesis. The 3x3 kernel is the minimal size to capture the directions of up, down, left, right, and the center [28]. The max-pooling layers are used to reduce the dimensions for the next convolutional layer. The 2x2 kernel

and stride of 2 for the max pooling layers halves the width and height. The fully connected layers are used to classify the results. They take the features extracted by the convolutional layers and output the probabilities of the 1000 classification labels that the input belongs to.

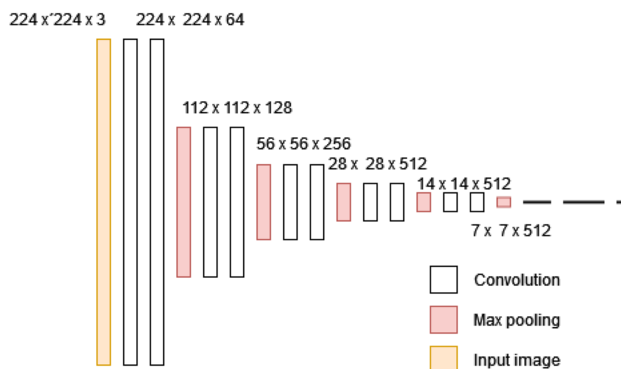

Figure 2.3: Image showing the structure of VGG16

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2.1: Table 1 from Simonyan and Zisserman showing the different configurations the researchers implemented [28].

2.2 RISC-V

RISC-V ("risk five") is an open Instruction Set Architecture (ISA) pioneered by Krste Asanoviü, Andrew Waterman, and Yunsup Lee in 2011 [29]. In 2015 the RISC-V foundation was created whose purpose is to manage and organize the community. The goal of RISC-V is to have a free and open ISA that is suitable for any application. It is a modular ISA meaning that you can add or remove instruction extensions that aren't necessary for your application, reducing the footprint. The base ISA has support for 32-,64-, and 128-bit address space and integer instructions [30]. Furthermore, the base extensions M, A, D, F (also known together as G) provide the general purpose instruction set. The G set of instructions provide integer multiplication and division, single- and double-precision floating point instructions, and atomic memory operations.

2.2.1 Vector Extension

The RISC-V Vector Extension, RISC-V "V", adds 32 vector registers and vectorized SIMD instructions. The element length is required to be greater or equal to 8 bits and a power of 2. The vector length is required to be at least equal to the element length and no greater than 2^{16}, and also a power of 2 [31]. This means that the maximum vector length able to be configured is 65,536 elements wide. The Vector Extension version 1.0 has been frozen since September 2021 and is undergoing review. It is considered stable enough to develop implementations with, as done in this thesis, and when it is ratified it will become Vector Extension version 2.0 [31].

2.2.2 RISC-V JIT

The key development done in previous work concerning this thesis is the implementation of a RISC-V JIT in oneDNN. The main usage of the JIT in this project has been to enable the use of RISC-V instructions. It consists of six files integrated into oneDNN, jit_assembler.hpp, instruction_types.h, mnemonic.c, rvjit.c, rvjit.h, and rvjit.hpp. The following section contains a description of these files and how they work together. instruction_types.h contains instruction definitions for the RISC-V ISAs different instruction types, the file acts as a header file for mnemonic.c where the instructions are implemented. The purpose of these are to populate the bitfields to correspond to the correct OP-codes that are defined in the RISC-V ISA specifications. The code below showcases the Vector Extension bitfield generators.

```
/// @brief Creates a V-type instruction from its bitfields
rvj_instr opV(int vd, int vs1, int vs2, int f6,int width, int vm) {
    return 0x57 | ((vd & 0x1F) << 7) | ((width & 0x7) << 12)
        | ((vs1 & 0x1F) << 15) | ((vs2 & 0x1F) << 20) | ((vm & 1) << 25)
        | ((f6 & 0x3F) << 26);
}
/// @brief Creates a int vector-vector V-type instruction from its bitfields
rvj_instr opIVV(int vd, int vs1, int vs2, int f6, int vm) {
    return opV(vd, vs1, vs2, f6, 0, vm);
```

```
}
/// @brief Creates a float vector-vector V-type instruction from its bitfields
rvj_instr opFVV(int vd, int vs1, int vs2, int f6, int vm) {
    return opV(vd, vs1, vs2, f6, 1, vm);
}
/// @brief Creates a M vector-vector V-type instruction from its bitfields
rvj_instr opMVV(int vd, int vs1, int vs2, int f6, int vm) {
    return opV(vd, vs1, vs2, f6, 2, vm);
}
/// @brief Creates a vector-immediate V-type instruction from its bitfields
rvj_instr opIVI(int vd, int simm5, int vs2, int f6, int vm) {
    return opV(vd, (simm5 & 0x1F), vs2, f6, 3, vm);
}
/// @brief Creates a vector-scalar (GPR) V-type instruction from its bitfields
rvj_instr opIVX(int vd, int rs1, int vs2, int f6, int vm) {
    return opV(vd, rs1, vs2, f6, 4, vm);
}
/// @brief Creates a vector-scalar (FP) V-type instruction from its bitfields
rvj_instr opFVF(int vd, int rs1, int vs2, int f6, int vm) {
    return opV(vd, rs1, vs2, f6, 5, vm);
}
/// @brief Creates a vector-scalar (GPR) V-type instruction from its bitfields
rvj_instr opMVX(int vd, int rs1, int vs2, int f6, int vm) {
    return opV(vd, rs1, vs2, f6, 6, vm);
}
```

Furthermore, in the file mnemonic.c the RISC-V instructions that make use of these OP-code generators are also defined. Such as this implementation of vslideup_vx. This function is defined in the RISC-V specifications as [31]:

```
vslideup behavior for destination elements
OFFSET is amount to slideup, either from x register or a 5-bit immediate
    0 <=  i < max(vstart, OFFSET)   Unchanged
    max(vstart, OFFSET) <= i < vl   vd[i] = vs2[i-OFFSET] if v0.mask[i] enabled
    vl <= i < VLMAX                 Follow tail policy
```

In the document containing the OP-codes for RISC-V Vector extension it is located under OPIVX and has the following definition [32]:

```
# OPIVX
vslideup.vx    31..26=0x0e vm vs2 rs1 14..12=0x4 vd 6..0=0x57
```

From this we can gather that the correct OP-code generator to use is opIVX and the corresponding OP-code is 0x0e.

```
rvj_instr rvj_vslideup_vx(REGV vd, REGV vs2, REGX rs1) {
    return opIVX(vd, vs2, rs1, 0x0e, rvj_unmasked);
}
```

Figure 2.4: RVJIT implementation of vslideup_vx

The file rvjit.hpp contains the function that is used by the programmer when implementing the convolution layer.

```
void vslideup_vx(vr_t vd, vr_t vs2, gpr_t rs1, vmask_t vm = vmask::unmasked) {
    push(rvj_vslideup_vx(vd, vs2, rs1));
}
```

Figure 2.5: Instruction alias for use by the programmer

The file rvjit.c contains the core JIT functions. These push the instructions, manage memory for the generated binary code, manages instruction labels and ensures that the labels are unique. It memory maps the instructions using mmap and sets the generated code to be executable by using mprotect. From the programmers point of view the JIT handles the code generation and memory management in the background and leaves only the function logic for e.g., GEMM to be implemented by the programmer. However, while it defines the registers, such as general purpose registers and vector registers, which makes it so that the programmer can use the registers in their code. It leaves the management of those register up to the programmer. Meaning the user has to see to it that the correct registers are used in the correct scope and that registers are not overwritten.

2.2.3 QEMU Emulator

To run the code that is created using the RISC-V ISA instructions the project uses the QEMU Emulator [33]. In general, an emulator is a device or application that behaves like another system. The purpose of this is to run programs made for this other system. For example, emulating a video game console to play an old game on your modern computer. Two important terms here are host and guest system. The host system is the actual device or hardware that is running the emulator. The guest system is what is to be emulated and is a virtual version of the real hardware. QEMU is an emulator created to run code and programs that are made for another operating system than your host operating system.

E.g., if your host system is running Windows then you can use QEMU to emulate a Linux system and run applications made for Linux or vice versa. Additionally, it can also emulate other CPUs, such as ARM, MIPS, and RISC-V. This means that we can use an x86 based system and run code compiled for RISC-V without needing a real RISC-V CPU.

QEMU has a dynamic translator. This means that it converts code from the target CPU instruction to the host instructions during runtime. It does this dividing the target instructions to *micro operations* [33]. These micro operations are small pieces

of C code that are chained together to then execute on the host system. In our case that means that during the runtime of the program, QEMU converts RISC-V ISA instructions to x86 ISA instructions.

QEMU also has support for plugins. These plugins are able to monitor the system state [34]. These plugins can enable the user to receive an executed instructions count, counting amount of times that different instructions are used, and tracing instructions with memory access for security purposes. Users are also encouraged to contribute to the QEMU project with their own plugins.

2.3 RISC-V For Machine Learning

The RISC-V ISA and the RISC-V Vector extension provide useful instructions and benefits when it comes to the field of machine learning. As mentioned, vectorized instructions and algorithms enable faster computations of different layers in machine learning models thanks to being able to handle more data at once. This is further increased by the ability to change the vector length which could enable processing up to 65,536 elements in parallel. This scalability means that RISC-V is useful for both low-power systems and high-performance systems.

Furthermore, since the RISC-V ISA is open-source it is available to a wider market for research, both in academia and the private sector. The RISC-V ISA being open-source also pushes for collaboration and knowledge-sharing which increases the speed of innovation. A goal of using RISC ISAs compared to Complex Instruction Set Computers (CISC) such as one based on the x86 ISA is the reduced amount of instructions which could lead to a reduced instruction overhead and make a RISC device more energy efficient than CISC device. RISC-V with its modularity may lead to an even more energy efficient device by removing instructions that aren't necessary. This aspect of energy efficiency may be a concern to large-scale applications such as in data centers or battery-powered devices. When it comes to this thesis however, energy efficiency is not taken into consideration but this aspect as well as those mentioned shows why RISC-V is of interest for this thesis and future research.

3

Methods

3.1 Working Environment

3.1.1 Compiler

This project uses an LLVM based compiler developed by the Barcelona Supercomputing Center (BSC) [35]. The thesis uses a pre-built binary that is made for RISC-V Vector Extension 1.0 [36]. It is part of the European Processor Initiative project and implements a set of vector intrinisics. These instrinsics were not used as part of this project and instead the RISC-V JIT developed for oneDNN mentioned earlier is used. In other words, the compiler is used to be able to compile the project to RISC-V ISA executables, not for its vector intrinsics or EPI specific features.

A common method to compile RISC-V programs, which was used at the start of the project, is the use of the RISC-V GNU Toolchain [37]. It is an open-source toolchain and compiler that generates executable files from C or C++ code. The reason the LLVM compiler is used over the GCC compiler is that it was used in the previous projects at Chalmers and eases the continuity in adapting the previously developed algorithms. Furthermore, the RISC-V JIT was also developed at BSC and keeping the same tools as the previous projects reduces unnecessary complexity.

3.1.2 QEMU

The QEMU setup used in the project is version 8.2.2 which can be found on the QEMU website [38]. It is important to download the version from the repository instead of downloading it from a package manager. This is due to the fact that the version that the package managers may not have the latest version.

After downloading QEMU it was build using these configurations:

```
sudo ./configure --enable-slirp --enable-virtfs --enable-plugins
```

These are used to enable file transfers between the host system and QEMU, sharing folders, and enabling the use of plugins for benchmarking.

QEMU has two modes, system-mode and user-mode. System mode emulates the whole system including system calls and signals. User-mode emulates just the program running on the system without the overhead of system signals [38]. The following flags are used when running QEMU. It is running Vector Extension 1.0

and if vector length and element length aren't specified their default values are 128 and 64 respectively.

```
qemu-system-riscv64 \
    -nographic \
    -machine virt \
    -cpu rv64,v=true \
    -m 6G \
    -smp 4 \
    -virtfs local,path=shared_folder,mount_tag=host0,
        security_model=passthrough,id=host0 \
    -device virtio-blk-device,drive=hd \
    -drive file=overlay.qcow2,if=none,id=hd \
    -device virtio-net-device,netdev=net \
    -netdev user,id=net,hostfwd=tcp::2222-:22 \
    -bios /usr/lib/riscv64-linux-gnu/opensbi/generic/fw_jump.elf \
    -kernel /usr/lib/u-boot/qemu-riscv64_smode/uboot.elf \
    -object rng-random,filename=/dev/urandom,id=rng \
    -device virtio-rng-device,rng=rng \
    -append "root=LABEL=rootfs console=ttyS0"
```

The following flags are used to run QEMU in user-mode. It is running Vector Extension 1.0 and using a vector length of 1024 and element length of 64. Here a plugin is enabled, which can be done for the system-mode as well, to count the number of instructions executed [34]. Since it is in user-mode the instruction count is more representative of the amount the program will use. Using these plugins in system-mode includes instructions that are executed to emulate the whole Linux system which is not useful to evaluate the the performance of the convolutional network.

```
qemu-riscv64 \
    -cpu rv64,v=true,vlen=1024,elen=64,vext_spec=v1.0 \
    -plugin /home/samjons/thesis/qemu/build/tests/plugin/libinsn.so \
    -d plugin \
```

3.1.3 oneDNN

The following flags are used to compile oneDNN.

```
-DDNNL_TARGET_ARCH=RV64 -DDNNL_CPU_RUNTIME=SEQ -DDNNL_LIBRARY_TYPE=STATIC \
-DCMAKE_C_COMPILER=/llvm-EPI-development-toolchain-cross/bin/clang \
-DCMAKE_CXX_COMPILER=/llvm-EPI-development-toolchain-cross/bin/clang++ \
-DCMAKE_SYSROOT=/llvm-EPI-development-toolchain-cross/
                            riscv64-unknown-linux-gnu/sysroot \
-DCMAKE_LIBRARY_PATH=/llvm-EPI-development-toolchain-cross/
                            riscv64-unknown-linux-gnu/sysroot/usr/lib64/lp64d \
-DCMAKE_C_FLAGS="-march=rv64gcv" \
-DCMAKE_CXX_FLAGS="-march=rv64gcv" \
```

3.2 Network Model

As stated in 2.1.8 the D configuration of VGG is used. It is implemented using oneDNN's forward inference property since the goal of the project is not to train the network but to improve the efficiency of the convolutional layers.

VGG16 is implemented using the oneDNN API and are in large parts modified versions of the cnn inference example in the oneDNN documentation. VGG16 is used to compare different convolutional layer configurations. I.e., the network model is static while the specific convolutional layer implementations it uses changes.

The networks are created by having each layer (convolution, ReLu, max pool, fully connected) be added to the network. After the layers are added with their specific configuration, such as amount of input channels and kernel size etc., they are executed sequentially.

3.3 RISC-V JIT

The JIT is called during the creation of the network and when the convolution layers are added. The JIT's instructions push the code and assembles it to a function pointer. The JIT sets the appropriate permissions to allow the generated code to be executed. then the code is executed as a function which means that the code can alternate between C++ code and vector instructions during execution.

3.3.1 JIT versus compiler intrinsics

In the previous work RISC-V vector intrinsics were used instead of a JIT [12]. Compiler intrinsics means that the compiler has instructions that are aliases for RISC-V ISA instructions. For example the instruction to do vector floating point addition is:

```
__epi_2xf32 __builtin_epi_vfadd_2xf32_mask(__epi_2xf32 merge, __epi_2xf32 a,
                                           __epi_2xf32 b, __epi_2xi1 mask,
                                           unsigned long int gvl);
```

In the JIT however, the instructions are implemented as assembly-like instructions that are pushed by the JIT.

```
void vfadd_vv(vr_t vd, vr_t vs1, vr_t vs2, vmask_t vm = vmask::unmasked) {
        push(rvj_vfadd_vv(vd, vs1, vs2, vm));
}
```

Comparing two code blocks, one using the vector intrinsics and one using the JIT we can notice some differences:

```
//Index calculation using intrinsics
__epi_2xi32 wcol = __builtin_epi_vload_2xi32(&w_col[w+0], gvl);
__epi_2xi32 OFFSET = __builtin_epi_vmv_v_x_2xi32(w_offset, gvl);
```

```
__epi_2xi32 PAD = __builtin_epi_vmv_v_x_2xi32(pad, gvl);
__epi_2xi32 STRIDE = __builtin_epi_vmv_v_x_2xi32(stride, gvl);
__epi_2xi32 intermediate1 = __builtin_epi_vmul_2xi32(STRIDE, wcol, gvl);
__epi_2xi32 imcol = __builtin_epi_vadd_2xi32(intermediate1, OFFSET, gvl);
__epi_2xi32 WIDTHCOL = __builtin_epi_vmv_v_x_2xi32(width_col, gvl);
__epi_2xi32 INTER = __builtin_epi_vmv_v_x_2xi32(intermediate, gvl);
__epi_2xi32 intermediate2 = __builtin_epi_vmul_2xi32(INTER, WIDTHCOL, gvl);
__epi_2xi32 colindex = __builtin_epi_vadd_2xi32(intermediate2, wcol, gvl);
imcol = __builtin_epi_vsub_2xi32(imcol, PAD, gvl);
```

```
//Index calculation using JIT instructions
const vr_t wcol = vout[0], OFFSET = vout[1], PAD = vout[2];
const vr_t STRIDE = vout[3], intermediate1 = vout[4], IMCOL = vout[5];
const vr_t WIDTHCOL = vout[6], INTER = vout[7], intermediate2 = vout[8];
const vr_t colindex = vout[9], WIDTH = vout[10], HEIGHT = vout[11];
const vr_t CIM = vout[12], IMROW = vout[13];

add(index, index, width_i);
vl(wcol, index, src_sew);
vmv_sx(OFFSET, w_offset);
vmv_sx(PAD, pad_reg);
vmv_sx(STRIDE, stride_reg);
vmul_vv(intermediate1, STRIDE, wcol);
vadd_vv(IMCOL, intermediate1, OFFSET);
vmv_sx(WIDTHCOL, width_end);
vmv_sx(INTER, intermediate);
vmul_vv(intermediate2, INTER, WIDTHCOL);
vadd_vv(colindex, intermediate2, wcol);
vsub_vv(IMCOL, IMCOL, PAD);
```

The most notable difference is how variables are handled. With the vector intrincics variables are managed by the compiler, meaning the programmer does not have to manually set specific registers and locations for the data to use. To use the JIT the programmer has to set which vector register to use manually and has to track which are used where and when in the program to ensure that the data is not overwritten or used at the wrong time.

Standard programming functions such as a for loop has to be converted to an assembly-like instruction block which makes the code longer and harder to read:

```
const gpr_t channel = tmp.pick();
const gpr_t channel_end = tmp.pick();
load_constant(channel, 0);
load_constant(channel_end, channels_col);
L("channels");
```

```
//for (c = 0; c < channels_col; ++c)
// Channel loop
addi(channel, channel, 1);
blt(channel, channel_end, "channels");
```

3.3.2 Converting previous work to oneDNN RISC-V JIT

Converting functions such as im2col+GEMM has consisted of finding which RISC-V ISA instructions correspond to which compiler intrinsics and converting the code to a more assembly-like format. In general, the benefits of the RISC-V JIT implemented in oneDNN is that it enables using RISC-V Vector instructions in your code. The main downside is the amount of manual work and manual checking the programmer has to do.

Another downside of the JIT, caused by this manual checking, and difficulty in converting the previous work to oneDNN is that it makes debugging more complicated. For example, if the data in a certain register is overwritten during the runtime of the program it might not be noticed until much later in the process. The program will run and execute properly from the outside but the data used could be from the wrong matrix and the result would be wrong. The challenge caused by this is that the problem is only noticed during execution and not during compilation of the program. Pinpointing the location of the bug is therefore more challenging in some cases.

3.3.3 New instructions

The process of adding instructions is quite simple, if tedious. As described briefly in 2.2.2, it is basically a three step process. Find the instruction you want to add, find the OP-codes that represent it and which instruction type it is, create the instruction to push the corresponding OP-code bitfield generator. Due to the fact that the implementations of im2col+GEMM and Winograd were created previously, finding the instruction to add involved finding what RISC-V ISA instruction was implemented by which intrinsic. In other words, except for a few cases there was no need to look into the RISC-V ISA description to find the relevant instructions to get the functionality that was needed. The exceptions to this involved looking into some more complex instructions functionality such as vslideup for example. Also during debugging after an instruction was added it was sometimes necessary to check that the registers were correctly formatted in the instruction inputs.

Some instructions that had to be added for this thesis were vector masking instructions. To ensure that they are correct the opcodes used for them come from the RISCV repository [32]. Another instruction that was added is the vslideup that was showcased in 2.2.2.

```
rvj_instr rvj_vmand_mm(rvj_vmask vd, rvj_vmask vs1, rvj_vmask vs2);
rvj_instr rvj_vmor_mm(rvj_vmask vd, rvj_vmask vs1, rvj_vmask vs2);
rvj_instr rvj_vmslt_vx(rvj_vmask vd, REGV vs1, REGV vs2);
```

```
rvj_instr rvj_vmseq_vv(rvj_vmask vd, REGV vs1, REGV vs2);
rvj_instr rvj_vmsgt_vx(rvj_vmask vd, REGV vs1, REGV vs2);
```

3.4 Auto-tuning

3.4.1 im2col+GEMM for VGG16

To auto-tune the im2col+GEMM implementation we first create the parameter space that will create the different configurations. To enable unrolling for GEMM each of the different unrolling factors had to be manually implemented and then depending on which unrolling factor is used a different part of the code is picked. An unroll factor of 1 means that one vector instruction is executed per iteration and an unroll factor of 8 means that 8 vector instructions are executed per iteration etc.

At first, a parameter space with three different parameters was defined, each of which had six possible states. The parameter space is shown in 3.1 and gives a possible of $6^3 = 216$ different configurations for each layer. This means that with 13 convolutional layers in VGG16 we get a total of $216^{13} = 2.2e30$ unique combinations. Considering the time and resources available this is not a feasible amount of configurations. Using the same unroll factor for each layer reduces the amount of configurations to $36^{13} \times 6 = 1e21$. Furthermore, to get a more reasonable amount of configurations given the amount of time, we limit the checked configurations to set K to 1. This further reduces the amount of configurations to $6^{13} \times 6 = 7.8e10$.

Blocks N dimension	1,2,4,8,16,32
Blocks K dimension	1,2,4,8,16,32
Unroll	1,8,12,16,24,32

Table 3.1: First attempt at defining a parameter space

However, to achieve an even more reasonable amount of configurations we can look at the structure of VGG16. As shown in 2.3 multiple layers share the same height and width dimensions. This means that they will also share the same dimensions for N, which is the parameter we are looking at. Since $N = h \times w$. We reduce the amount of configurations considerably due to only having six different values for N per configuration. In total we get a total amount of configurations of $6^5 \times 6 = 46654$.

Blocks N dimension	1,2,4,8,16,32
Unroll	1,8,12,16,24,32

Table 3.2: Second attempt at defining a parameter space

Since QEMU is able to be extended with plugins as mentioned in 3 we can receive the executed instruction count of the program. The time to both create the network and the time to fully execute the program is also measured. We only care about the actual time it takes to execute the program and not create the network

since theoretically we could have pre-generated the kernels, thus we take the total execution time and subtract the creation time of the network. We can therefore create an estimate of the performance of a configuration by comparing execution times and instruction counts executed. An important note here is that QEMU is an emulator and is therefore not completely representative of real world performance. The execution times and instruction counts are therefore used as estimations of how the program would run on real hardware. Regardless, the methodology used and the strategy to auto-tune the convolutional layers would still be the same. Through testing we can get the standard deviation and variation in performances to both see how consistent the runtime performance is and get a more well informed image of the accuracy of predictions.

To generate the data that the model uses to train on, random configurations of blocking are selected and then tested with each unroll factor. Each configuration executes a forward inference pass of VGG16 to get execution time and instruction count that can be used for prediction purposes. Since this is an offline auto-tuner, theoretically a full exhaustive search to explore the entire search space could guarantee finding the best configuration. However, for future work more parameters could be added, such as changing vector length. With each new parameter added the search space would increase exponentially. As such, developing a working method to avoid doing a full exhaustive search would be preferable. Therefore this project uses the state-of-the-art approach that the researchers at the Norwegian University of Science and Technology used which was mentioned in section 1.3. In their work they used ANNs to classify and predict the performances of their configurations. This project implements one model using ANNs and also implements a model using a random forest classifier. Both models are implemented using the tensorflow libary for Python.

In other words we define the search space above, we generate multiple different configurations and do performance tests on them, predict new configurations and their performances and finally test the best these predictions to find the best configuration given our system.

3.4.2 Network agnostic auto-tuning

In addition to the VGG16 specific auto-tuner two approaches to creating a more network agnostic auto-tuner will be attempted. Network agnostic meaning that we could switch out layers or add layers to transfer the knowledge from training on VGG16 to other networks. The approach for creating the network agnostic models will be the same as for the VGG16 specific auto-tuner. The difference will be in the training data. Instead of training on the blocking factors we will switch to use the network characteristics. I.e., the values for M, N, K that are specific to each layer.

The reason we are using M, N, K instead of the layer sizes such as input height and width, kernel size, filters/channels etc. is because we reduce the amount of parameters from five per layer to three per layer. This might not seem like much but consider VGG16 with 13 convolutional layers. We reduce the amount of possible parameters from $13 \times 5 = 65$ to $13 \times 3 = 39$.

The first approach will be to switch out the blocking factors to use the actual values for N instead. Meaning the network will learn explicitly which values are used instead of with the blocking factors where it learns implicitly what values are used in the network. The second approach will be one using all 39 parameters in addition to the blocking factors and unrolling factor.

The goal of using these explicitly stated values is to enable switching to another layer with other values. For example, when using blocking factors the model learns implicitly that dividing layer 5 in 8 blocks is good but if we were to switch it to a layer with a different M, N, K values dividing by 8 might not work at all. By using the actual values the model may be able to learn more explicitly how the values for M, N, K interact and affect the resulting execution times and instruction counts.

3.4.3 Models

The different models used for auto-tuning predictions are: ANN, Random Forests, Linear Regression, and SVM. These are described in chapter 2. These were all implemented in Python 3.9 using scikit-learn, except for the ANN which was implemented in Python 3.9 using tensorflow. The same dataset, classes, and prediction parameters were used for all meaning we can compare the quality of the predictions between the models.

The ANN is implemented following the previous research at the Norwegian University of Science and Technology [13]. I.e., one single hidden layer with 30 neurons using a sigmoid activation function. The reasoning behind choosing the same model is to give a baseline and a reference point from previous work.

For the random forest, 100 trees are used which is a commonly used amount. It gives a good balance between accurate results and performance. A random forest with 200 trees was also tested but gave a negligible difference in the results. Therefore, in the result section the random forest with 100 trees is used.

The linear regression model is using the model directly imported from scikit. Two SVM models were implemented, one using the radial basis function (rbf) as kernel, meaning it transforms the data to a higher dimensional space to find non-linear relations. The other used a linear kernel. The one using rbf performed better relative to the linear and thus is used in the result chapter.

4

Results

4.1 Auto-tuning VGG16

The following figures show the results from four different models used to make predictions for auto-tuning im2col+GEMM. Each model's ten best predictions were used to compare with real executions to verify their accuracy and capabilities. Also presented is the ratios of the instruction count predictions for each layer in a configuration. These show the deviation from 1. Meaning a ratio of 0.5 means a wrong prediction by 50%.

The following sections have visualized the top ten predictions from the different models. For comparison. but not visualized. executing VGG16 using naive im2col+GEMM implementations has the following results:

- Creation time: 34.676 s
- Execution time: 731.251 s
- Instruction count: 26394774869

Naive meaning that the implementation is not using any RISC-V JIT instructions and operating on one element at a time instead of a vector.

Executing VGG16 with the RISC-V JIT im2col+GEMM without unrolling or blocking has the following results:

- Creation time: 43.217 s
- Execution time: 52.197 s
- Instruction count: 37818226595

As a reminder. in table 3.2 the parameter space is defined as six different values for blocking the N parameter where the value is how many blocks we divide the N dimension into. All layers use the same unrolling factor. Five different values for N in VGG16 means and six unrolling factors per configuration means that we have 46654 total configurations.

The y-axis scales are not the same between models. instead they are made to have the columns be most visible and readable sizes.

4.1.1 ANN model

Table 4.1 shows the ten best configurations suggested by the ANN model:

Configuration	N1	N2	N3	N4	N5	Unroll
1	1	1	32	1	32	1
2	2	1	32	1	32	1
3	1	1	32	1	1	1
4	1	1	32	1	2	1
5	1	1	32	1	4	1
6	1	1	32	1	16	1
7	1	1	32	1	8	1
8	1	1	32	1	32	1
9	1	1	32	2	32	1
10	2	1	32	1	1	1

Table 4.1: ANN model configuration predictions

Important to note for the ANN model is that the execution time y-axis in figure 4.1 is between 30 and 90 seconds while it is between 30 and 50 for the others.

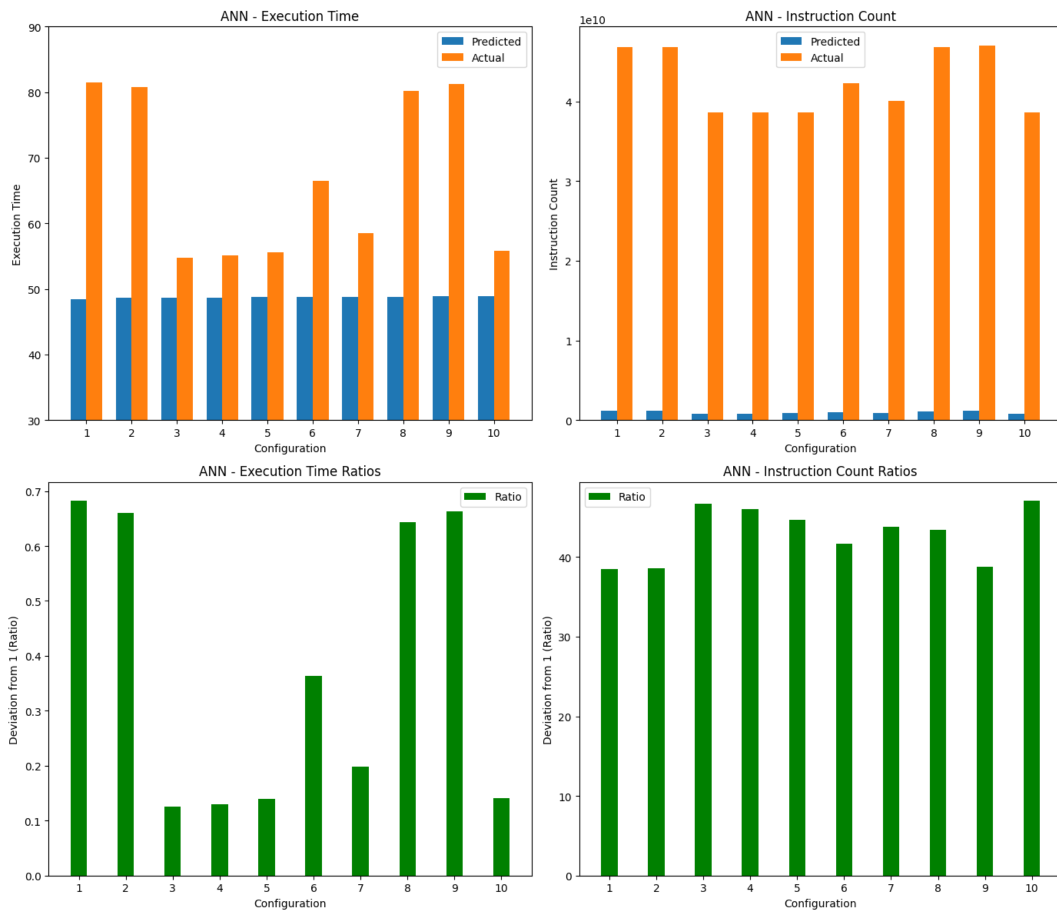

Figure 4.1: Results from ANN model prediction

Here is a breakdown of the individual layers for the ANN model predictions presented as the ratio between prediction and actual value:

Config	L0	L1	L2	L3	L4	L5
1	39.49925	12.13436	6.64428	9.44795	12.05852	15.92881
2	39.60258	11.92213	6.66701	9.54237	12.28824	16.15547
3	47.74927	18.00278	9.57948	13.35904	16.31229	23.81524
4	47.05593	17.80156	9.25102	13.17418	16.52670	23.89964
5	45.71911	17.16136	8.94258	12.94025	15.82377	22.79462
6	42.66019	14.44958	7.86498	11.12910	13.86080	19.32016
7	44.82042	15.58634	8.48207	12.02498	14.88162	21.21385
8	44.39647	13.01141	7.17122	10.44108	13.23531	17.78692
9	39.80746	11.96085	6.66076	9.43151	11.96878	15.83827
10	48.06756	19.29978	9.63486	13.83746	17.02497	24.39709
Avg	4.39E+01	1.51E+01	8.09E+00	1.15E+01	1.44E+01	2.01E+01

Table 4.2: ANN model layer instruction count predictions ratios (L0 to L5)

Config	L6	L7	L8	L9	L10	L11	L12
1	8.70892	8.21686	12.97720	18.98925	34.13941	46.82095	76.34222
2	8.66442	8.26688	12.80373	19.00791	33.95538	44.81757	75.93022
3	14.59537	12.68064	21.63606	26.97795	6.20495	8.52490	14.16318
4	14.22306	12.47540	21.30948	26.05839	6.41631	8.49050	14.12455
5	13.74180	12.35800	20.56084	25.88426	5.99204	8.49736	13.81835
6	11.10172	10.36504	16.40949	22.27524	21.33652	28.40472	47.65591
7	12.51471	11.12185	19.02634	24.78895	12.83541	16.56991	27.96048
8	9.74316	9.12925	14.74065	21.77775	37.42111	49.33783	82.87887
9	8.72939	8.58002	13.79331	20.21976	33.73695	45.09932	76.29218
10	14.72734	63.35190	5.61447	26.92918	6.26001	8.54580	14.24142
Avg	1.17E+01	1.57E+01	1.59E+01	2.33E+01	1.98E+01	2.65E+01	4.43E+01

Table 4.3: ANN model layer instruction count predictions ratios (L6 to L12)

4.1.2 Linear Regression model

Table 4.4 shows the ten best configurations suggested by the Linear Regression model:

Configuration	N1	N2	N3	N4	N5	Unroll
1	1	1	1	1	1	32
2	2	1	1	1	1	32
3	1	2	1	1	1	32
4	4	1	1	1	1	32
5	2	2	1	1	1	32
6	1	1	2	1	1	32
7	2	1	2	1	1	32
8	4	2	1	1	1	32
9	1	2	2	1	1	32
10	1	4	1	1	1	32

Table 4.4: Linear Regression configuration predictions

Figure 4.2 shows the resulting execution times and instruction counts for the configurations suggested by the Linear Regression model.

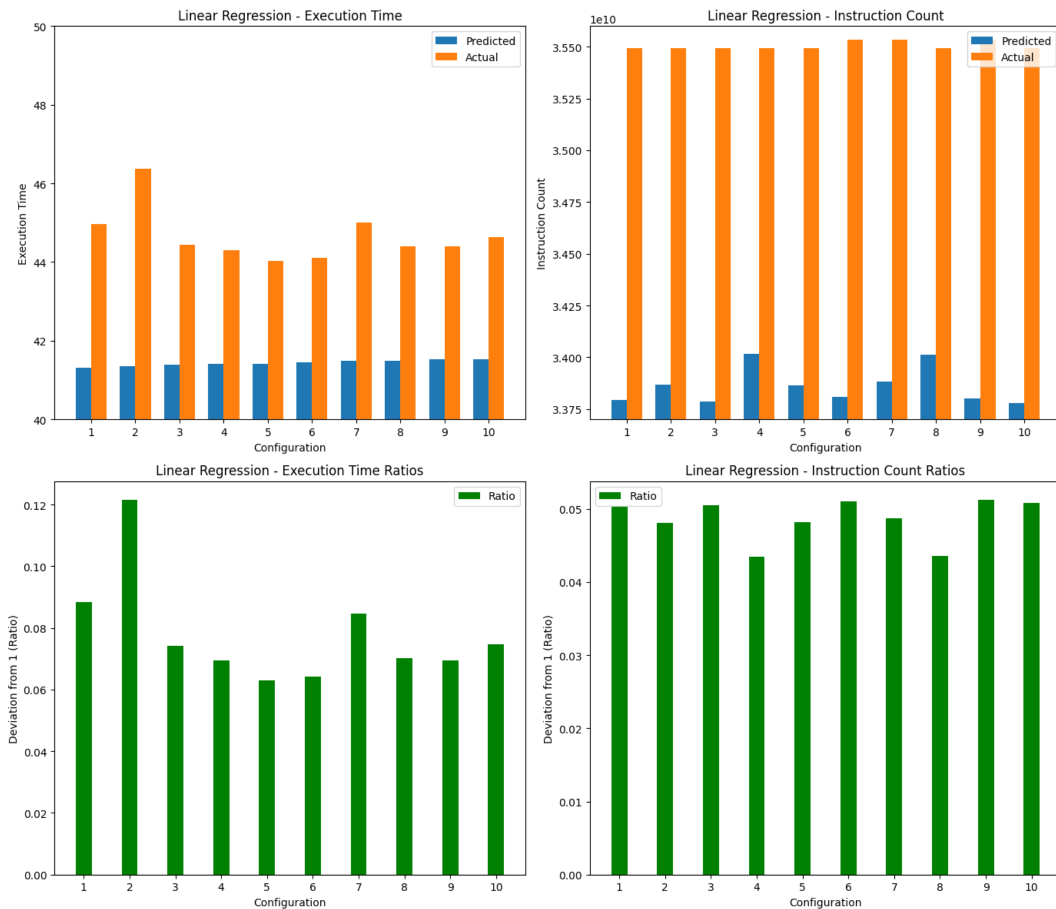

Figure 4.2: Results from Linear Regression model prediction

Here is a breakdown of the individual layers for the Linear Regression model predictions presented as the ratio between prediction and actual value:

Config	L0	L1	L2	L3	L4	L5
1	1.05038	1.14436	1.17274	1.12767	1.09528	1.11395
2	1.04806	1.18036	1.10502	1.10832	1.09618	1.10743
3	1.05053	1.14789	1.13133	1.12423	1.09767	1.14553
4	1.04343	1.10939	1.11377	1.11526	1.09417	1.12250
5	1.04821	1.10561	1.11296	1.11931	1.07932	1.09445
6	1.05106	1.12127	1.10580	1.10458	1.10312	1.11933
7	1.04874	1.13441	1.13003	1.11184	1.09784	1.13037
8	1.04358	1.13433	1.10897	1.09940	1.09438	1.11198
9	1.05121	1.11021	1.11145	1.11699	1.15058	1.14021
10	1.05084	1.12846	1.13049	1.10123	1.09002	1.13029
Avg	1.05E+00	1.13E+00	1.12E+00	1.11E+00	1.10E+00	1.12E+00

Table 4.5: Linear Regression model layer instruction count predictions ratios (L0 to L5)

Config	L6	L7	L8	L9	L10	L11	L12
1	1.12880	1.08312	1.08578	1.08184	1.03340	1.04128	1.04024
2	1.14016	1.08792	1.07570	1.08064	1.03752	1.03568	1.04129
3	1.11482	1.06198	1.05066	1.06527	1.02677	1.03006	1.03567
4	1.13013	1.07140	1.06187	1.07510	1.04187	1.05498	1.04526
5	1.11915	1.06825	1.07120	1.08098	1.02777	1.03139	1.02801
6	1.13743	1.07247	1.06125	1.06860	1.02602	1.02622	1.02780
7	1.12139	1.07184	1.06959	1.08257	1.07576	1.05212	1.22774
8	1.11532	1.07788	1.07016	1.07662	1.04338	1.04844	1.04825
9	1.13024	1.06255	1.05364	1.06558	1.05325	1.05775	1.07184
10	1.14245	1.07414	1.07004	1.07186	1.02275	1.02207	1.02607
Avg	1.13E+00	1.07E+00	1.07E+00	1.07E+00	1.04E+00	1.04E+00	1.06E+00

Table 4.6: Linear Regression model layer instruction count predictions ratios (L6 to L12)

4.1.3 SVM model

Table 4.7 shows the ten best configurations suggested by the SVM model:

Configuration	N1	N2	N3	N4	N5	Unroll
1	8	4	2	1	1	16
2	8	4	2	2	1	16
3	8	4	1	1	1	16
4	8	4	4	1	1	16
5	8	4	1	2	1	16
6	8	4	4	2	1	16
7	8	4	2	1	2	16
8	8	4	2	2	2	16
9	8	4	1	1	2	16
10	8	4	4	1	2	16

Table 4.7: SVM configuration predictions

Figure 4.3 shows the resulting execution times and instruction counts for the configurations suggested by the SVM model.

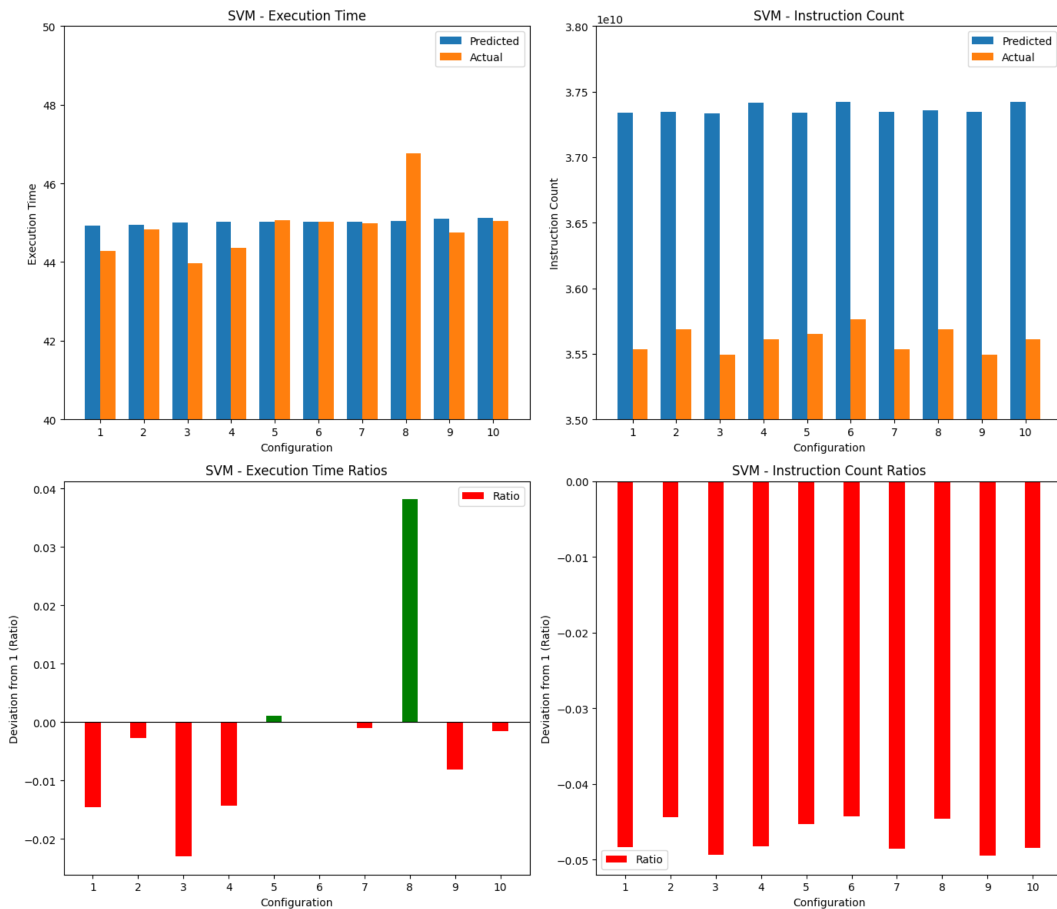

Figure 4.3: Results from SVM model prediction

Here is a breakdown of the individual layers for the SVM model predictions presented as the ratio between prediction and actual value:

Config	L0	L1	L2	L3	L4	L5
1	0.95161	1.01539	0.97614	1.01992	0.95726	1.01521
2	0.95559	1.00760	0.98635	0.99731	0.97838	1.05490
3	0.95070	1.01778	1.00011	0.99740	0.97171	1.02964
4	0.95174	1.00305	0.97721	0.99580	0.96823	1.05851
5	0.95468	1.00562	1.02884	1.03657	0.94975	1.01653
6	0.95571	1.02526	0.99046	0.99957	0.97403	1.04734
7	0.95143	1.01179	0.98590	1.02003	1.00462	1.03239
8	0.95540	1.02982	1.00116	1.02720	0.97834	1.04793
9	0.95021	1.01923	0.99094	1.01925	0.94120	1.03169
10	0.95361	1.02297	0.98248	1.01043	1.01816	1.09305
Avg	9.53E-01	1.02E+00	9.92E-01	1.01E+00	9.74E-01	1.04E+00

Table 4.8: SVM model layer instruction count predictions ratios (L0 to L5)

Config	L6	L7	L8	L9	L10	L11	L12
1	0.97955	0.95529	0.95276	0.94197	1.17070	1.15153	1.17092
2	0.98428	0.98950	0.99410	0.98287	1.17098	1.16991	1.15594
3	0.99872	0.95839	0.95351	0.93911	1.12362	1.12502	1.14049
4	0.99858	0.99791	0.96223	0.94755	1.18724	1.17998	1.18130
5	0.98472	1.00280	1.00249	0.98733	1.15018	1.13917	1.15605
6	1.00400	0.99525	1.00373	1.00132	1.20309	1.19132	1.20053
7	0.97874	1.00203	0.97662	0.99380	1.09732	1.08145	1.09335
8	0.99231	1.02270	1.01405	1.00370	1.09359	1.06881	1.08414
9	1.00130	0.94180	0.96753	0.92334	1.07460	1.07172	1.08659
10	1.04367	1.00275	0.98342	0.95547	1.03738	1.02166	1.03139
Avg	9.97E-01	9.87E-01	9.81E-01	9.68E-01	1.13E+00	1.12E+00	1.13E+00

Table 4.9: SVM model layer instruction count predictions ratios (L6 to L12)

4.1.4 Random Forest model

Table 4.10 shows the ten best configurations suggested by the Random Forest model:

Configuration	N1	N2	N3	N4	N5	Unroll
1	4	2	1	1	1	24
2	4	1	1	1	2	24
3	4	1	1	1	4	24
4	4	1	1	1	1	24
5	4	2	1	1	2	24
6	4	2	1	1	4	24
7	16	2	1	1	1	24
8	2	1	1	1	2	24
9	2	1	1	1	4	24
10	2	2	1	1	1	24

Table 4.10: Random Forest configuration predictions

Figure 4.4 shows the resulting execution times and instruction counts for the configurations suggested by the Random Forest model.

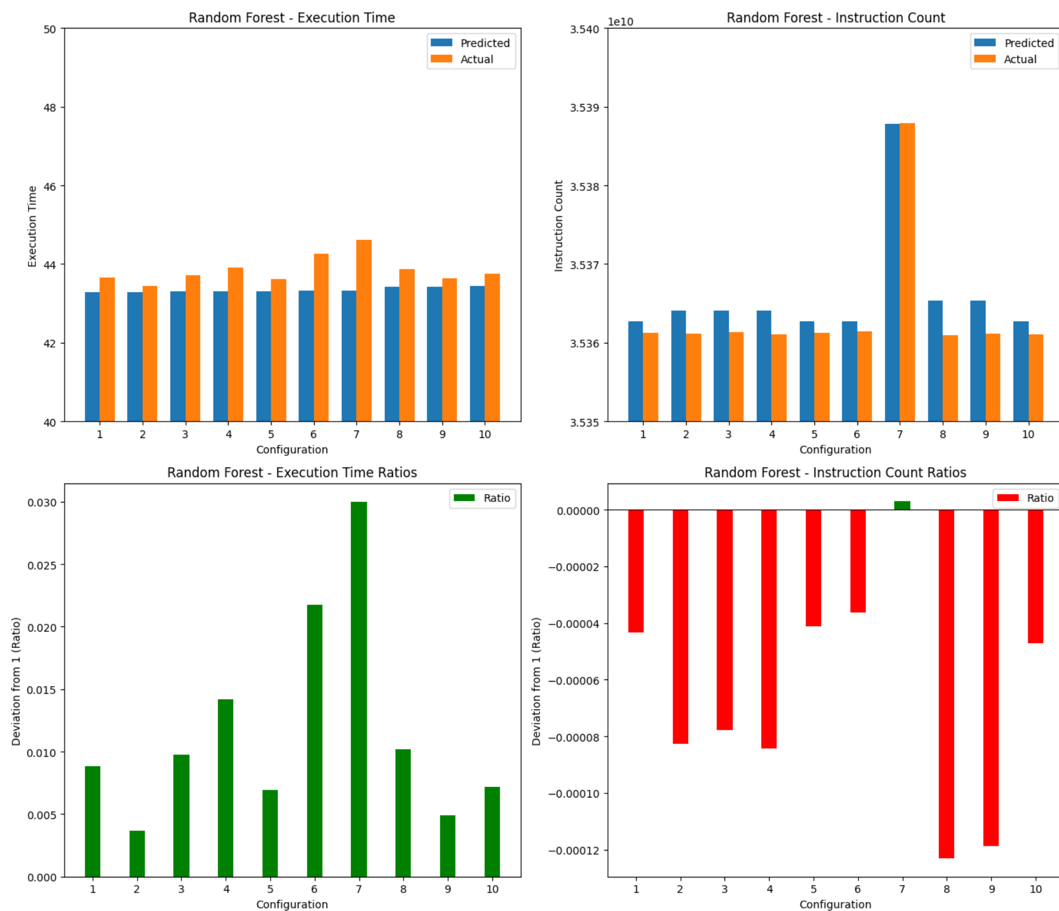

Figure 4.4: Results from Random Forest model prediction

Here is a breakdown of the individual layers for the Random Forest predictions presented as the ratio between prediction and actual value:

Config	L0	L1	L2	L3	L4	L5
1	0.99995	1.00321	1.02353	0.97385	0.98632	0.98780
2	0.99991	1.01050	1.00815	1.02241	1.02091	1.01000
3	0.99992	0.99520	1.03755	1.02112	0.99918	1.00297
4	0.99991	1.00843	1.03969	1.00808	1.01641	0.99367
5	0.99995	1.03134	1.00696	0.97319	0.99723	0.98836
6	0.99996	1.03576	1.00832	1.00052	0.99752	0.98732
7	1.00000	1.00954	0.99546	0.99350	1.00607	0.99449
8	0.99987	0.99970	1.05256	1.01622	1.03141	0.98786
9	0.99988	1.00636	1.01616	1.00106	0.99716	1.00048
10	0.99987	1.00334	1.03509	0.99774	1.01602	1.02311
Avg	1.00E+00	1.01E+00	1.02E+00	1.00E+00	1.01E+00	9.98E-01

Table 4.11: Random Forest layer instruction count predictions ratios (L0 to L5)

Config	L6	L7	L8	L9	L10	L11	L12
1	0.98431	1.00309	1.02133	1.00061	1.00351	1.01201	0.99942
2	1.02643	1.02993	1.03291	1.01772	1.01165	1.01267	1.00190
3	0.99736	1.02114	1.00978	1.00883	1.01006	1.00674	1.00771
4	1.02311	0.99316	1.01537	1.01821	1.03523	1.01716	1.00225
5	0.98943	1.02036	0.99971	0.98970	0.99099	0.99681	0.99282
6	0.98165	0.98497	1.03502	1.06274	1.07927	1.03478	1.02776
7	0.99773	0.97548	0.98826	1.00752	1.00025	0.99848	0.99874
8	1.02226	1.00585	0.99626	0.98261	0.99010	0.98633	0.99039
9	1.00306	0.98853	1.00492	1.02125	1.00402	1.00120	1.00274
10	1.02608	1.00182	0.99582	0.98899	0.99461	0.99147	0.99395
Avg	1.01E+00	1.00E+00	1.01E+00	1.01E+00	1.01E+00	1.01E+00	1.00E+00

Table 4.12: Random Forest layer instruction count predictions ratios (L6 to L12)

The following tables compare the instruction counts per layer for the non auto-tuned RISC-V JIT implementation and the configuration with the lowest instruction count layer per layer. Note. the instruction count in the graph includes the creation time which is why even though configuration 7 has the highest count on the graph. the individual layers' instruction counts are the lowest.

Config	L0	L1	L2	L3	L4	L5
7	0.93574	0.77064	0.76924	0.75212	0.75635	0.74097

Table 4.13: Instruction count ratios between no unrolling or blocking and best predicted configuration (L0 to L5)

Config	L6	L7	L8	L9	L10	L11	L12
7	0.74673	0.73576	0.73769	0.75717	0.74240	0.73960	0.73637

Table 4.14: Instruction count ratios between no unrolling or blocking and best predicted configuration (L6 to L12)

4.2 Network agnostic auto-tuner

4.2.1 Switching N to actual values

The following figures show the results for the models trained on the actual N values rather than the blocking factors.

4.2.1.1 ANN-N model

Table 4.15 shows the ten best configurations suggested by the ANN-N model. In Figure 4.5 we see the resulting execution times and instruction counts for the configurations suggested by the ANN-N model. Finally, Tables 4.16 and 4.17 show a breakdown for the individual layers' instruction count ratios.

Configuration	N1	N2	N3	N4	N5	Unroll
1	50176	392	98	784	196	1
2	50176	392	196	784	196	1
3	50176	784	98	784	196	1
4	50176	784	196	784	196	1
5	50176	392	392	784	196	1
6	50176	1568	98	784	196	1
7	50176	784	392	784	196	1
8	50176	1568	196	784	196	1
9	50176	392	98	784	196	1
10	50176	392	196	784	196	1

Table 4.15: ANN-N configuration predictions

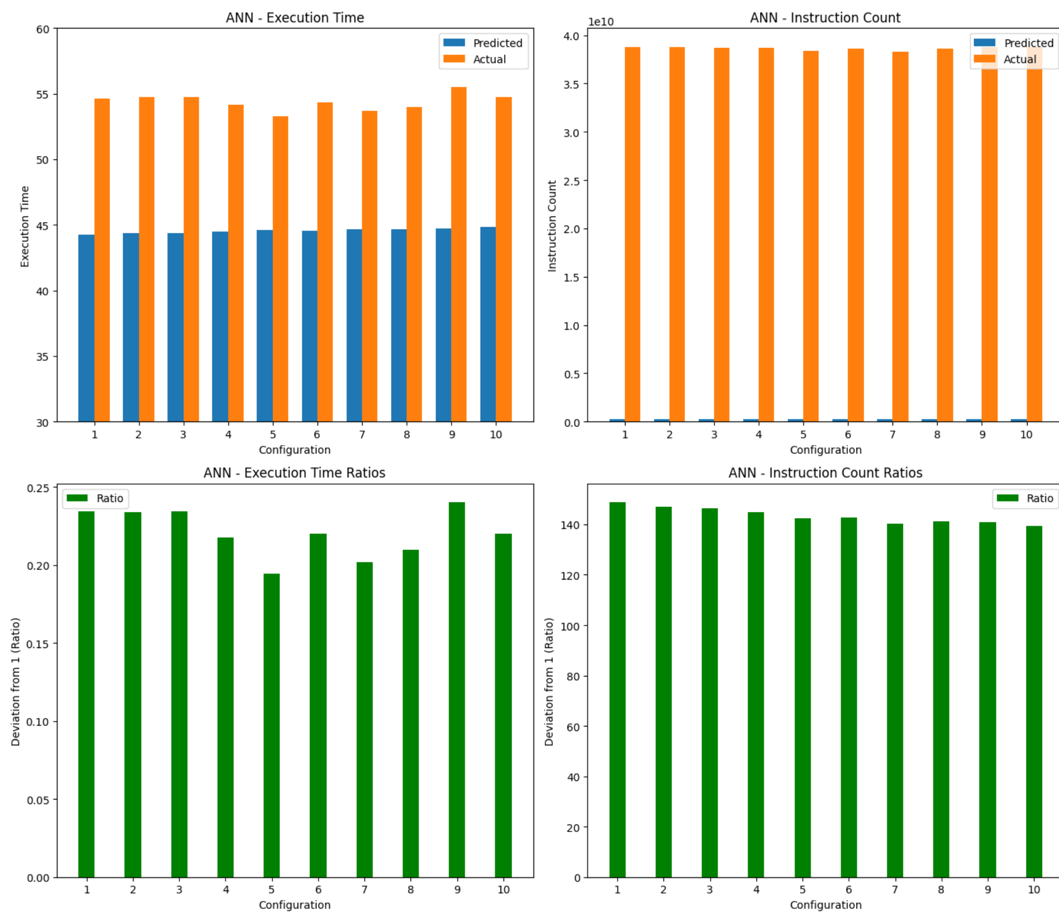

Figure 4.5: Results from ANN-N prediction

Config	L0	L1	L2	L3	L4	L5
1	149.76828	56.02920	28.93484	65.09417	36.08063	64.03918
2	148.13873	54.58199	29.24152	64.94810	35.28996	63.19611
3	147.46857	55.91829	27.11022	59.91627	35.61338	62.33061
4	145.86207	53.87062	26.55175	59.13030	34.97884	61.64842
5	143.44160	53.50559	28.34164	63.05823	30.27614	53.63133
6	143.63547	53.83608	25.14565	56.51021	35.18167	60.80592
7	141.22896	53.26190	25.95029	57.85992	31.02014	60.03355
8	142.06720	52.44339	24.77380	55.25099	34.16189	60.79915
9	141.93440	52.59369	27.44287	61.00457	34.45864	60.51207
10	140.43857	52.37267	27.05370	60.29888	33.31084	59.19108
Avg	1.44E+02	5.38E+01	2.71E+01	6.03E+01	3.40E+01	6.06E+01

Table 4.16: ANN-N layer instruction count predictions ratios (L0 to L5)

Config	L6	L7	L8	L9	L10	L11	L12
1	70.85483	28.80005	61.84872	49.68472	22.21368	27.95474	23.00676
2	70.40972	28.69557	60.43076	49.28519	22.02635	28.13318	22.79075
3	70.05656	28.51834	61.94731	49.81100	21.77399	28.45943	22.97698
4	68.98928	28.84093	59.93690	48.31480	21.98598	27.00229	22.98581
5	59.63075	28.14344	60.63785	49.16011	21.64713	27.01327	22.02942
6	68.06959	28.46764	60.19871	47.31660	21.68767	27.57043	22.10612
7	58.96785	28.01884	59.49504	50.06234	21.31531	26.71740	22.21258
8	67.19769	27.97631	58.31156	47.17172	21.01008	26.41014	21.91142
9	67.34666	27.46696	58.59278	47.10762	21.95601	27.88889	22.73363
10	66.45500	27.47258	57.11763	46.52348	21.60157	26.50039	21.74612
Avg	6.68E+01	2.82E+01	5.99E+01	4.84E+01	2.17E+01	2.74E+01	2.24E+01

Table 4.17: ANN-N layer instruction count predictions ratios (L6 to L12)

4.2.1.2 Linear Regression-N model

Table 4.18 shows the ten best configurations suggested by the Linear Regression-N model. In Figure 4.6 we see the resulting execution times and instruction counts for the configurations suggested by the Linear Regression-N model. Finally, Tables 4.19 and 4.20 show a breakdown for the individual layers' instruction count ratios.

Configuration	N1	N2	N3	N4	N5	Unroll
1	50176	12544	3136	784	196	32
2	25088	12544	3136	784	196	32
3	12544	12544	3136	784	196	32
4	6272	12544	3136	784	196	32
5	3136	12544	3136	784	196	32
6	1568	12544	3136	784	196	32
7	50176	12544	1568	784	196	32
8	25088	12544	1568	784	196	32
9	12544	12544	1568	784	196	32
10	6272	12544	1568	784	196	32

Table 4.18: Linear Regression-N configuration predictions

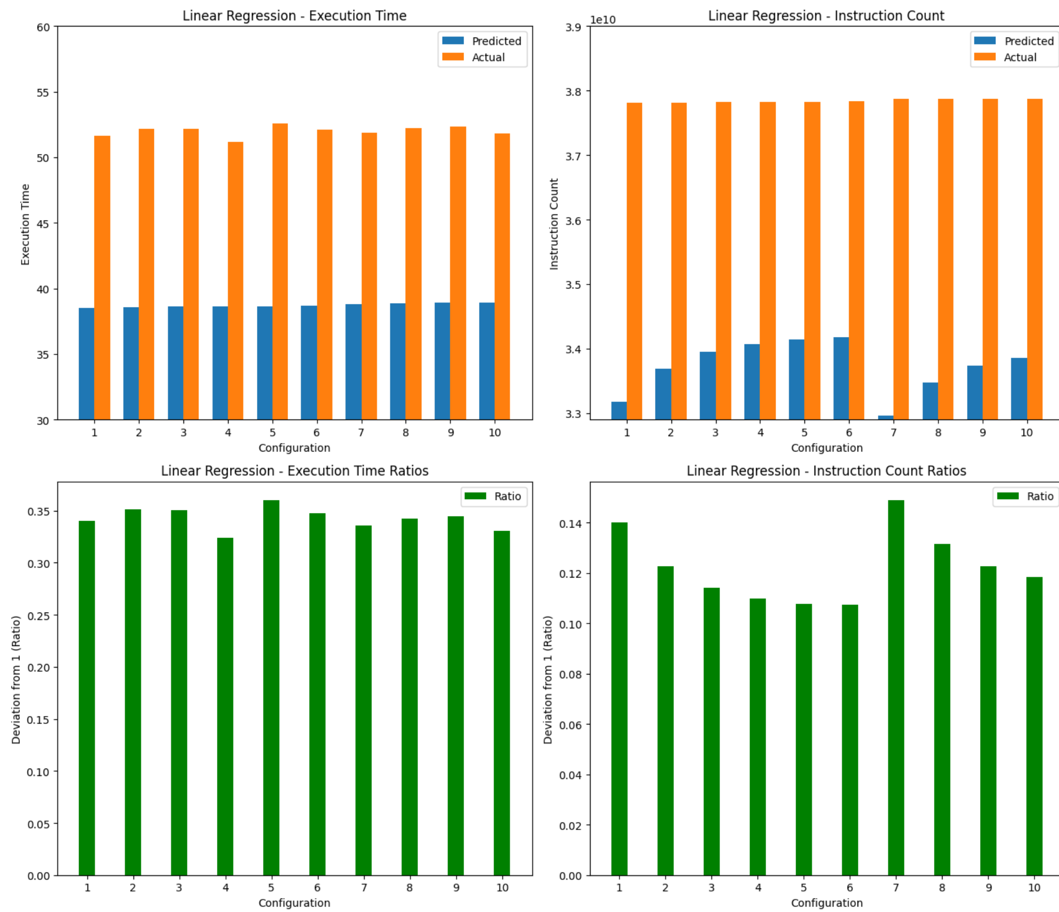

Figure 4.6: Results from Linear Regression-N prediction

Config	L0	L1	L2	L3	L4	L5
1	1.14001	1.43067	1.45271	1.44221	1.45223	1.51724
2	1.12265	1.46016	1.43360	1.46331	1.45130	1.55386
3	1.11408	1.45622	1.40945	1.46170	1.46371	1.54505
4	1.10982	1.43171	1.44479	1.45825	1.46104	1.53086
5	1.10770	1.48980	1.41497	1.45366	1.47335	1.52221
6	1.10729	1.48206	1.47646	1.46219	1.46079	1.54428
7	1.14885	1.46963	1.52722	1.44998	1.36224	1.41120
8	1.13136	1.50649	1.40753	1.44250	1.35859	1.41246
9	1.12272	1.42278	1.41211	1.45794	1.35413	1.49750
10	1.11842	1.43235	1.42539	1.45295	1.39704	1.42705
Avg	1.12E+00	1.46E+00	1.44E+00	1.45E+00	1.42E+00	1.50E+00

Table 4.19: Linear Regression-N layer instruction count predictions ratios (L0 to L5)

Config	L6	L7	L8	L9	L10	L11	L12
1	1.54688	1.57772	1.58284	1.57801	1.92751	1.98852	1.99116
2	1.55787	1.56528	1.58385	1.61971	1.94489	1.96579	1.98465
3	1.56154	1.60662	1.61830	1.61911	1.99850	2.25370	2.04426
4	1.52877	1.57570	1.57522	1.61438	1.96896	1.94150	1.94693
5	1.52689	1.69935	1.57985	1.61722	1.96894	1.95953	1.95252
6	1.54522	1.58962	1.59296	1.71217	1.97542	1.97956	2.08423
7	1.40432	1.54330	1.54923	1.54441	2.07401	2.10591	2.14124
8	1.41707	1.55226	1.60475	1.56089	2.11542	2.11451	2.12899
9	1.47223	1.60218	1.57157	1.65626	2.37760	2.30074	2.30941
10	1.47668	1.58938	1.55161	1.57127	2.07566	2.07632	2.08605
Avg	1.50E+00	1.59E+00	1.58E+00	1.61E+00	2.04E+00	2.07E+00	2.07E+00

Table 4.20: Linear Regression-N layer instruction count predictions ratios (L6 to L12)

4.2.1.3 SVM-N model

Table 4.21 shows the ten best configurations suggested by the SVM-N model. In Figure 4.7 we see the resulting execution times and instruction counts for the configurations suggested by the SVM-N model. Finally, Tables 4.22 and 4.21 show a breakdown for the individual layers' instruction count ratios.

Configuration	N1	N2	N3	N4	N5	Unroll
1	25088	12544	3136	784	196	32
2	25088	12544	3136	784	196	24
3	25088	12544	3136	784	196	16
4	25088	12544	3136	784	196	12
5	25088	12544	3136	784	196	8
6	25088	12544	3136	784	196	1
7	25088	12544	3136	784	98	32
8	25088	12544	3136	784	98	24
9	25088	12544	3136	784	98	16
10	25088	12544	3136	784	98	12

Table 4.21: SVM-N configuration predictions

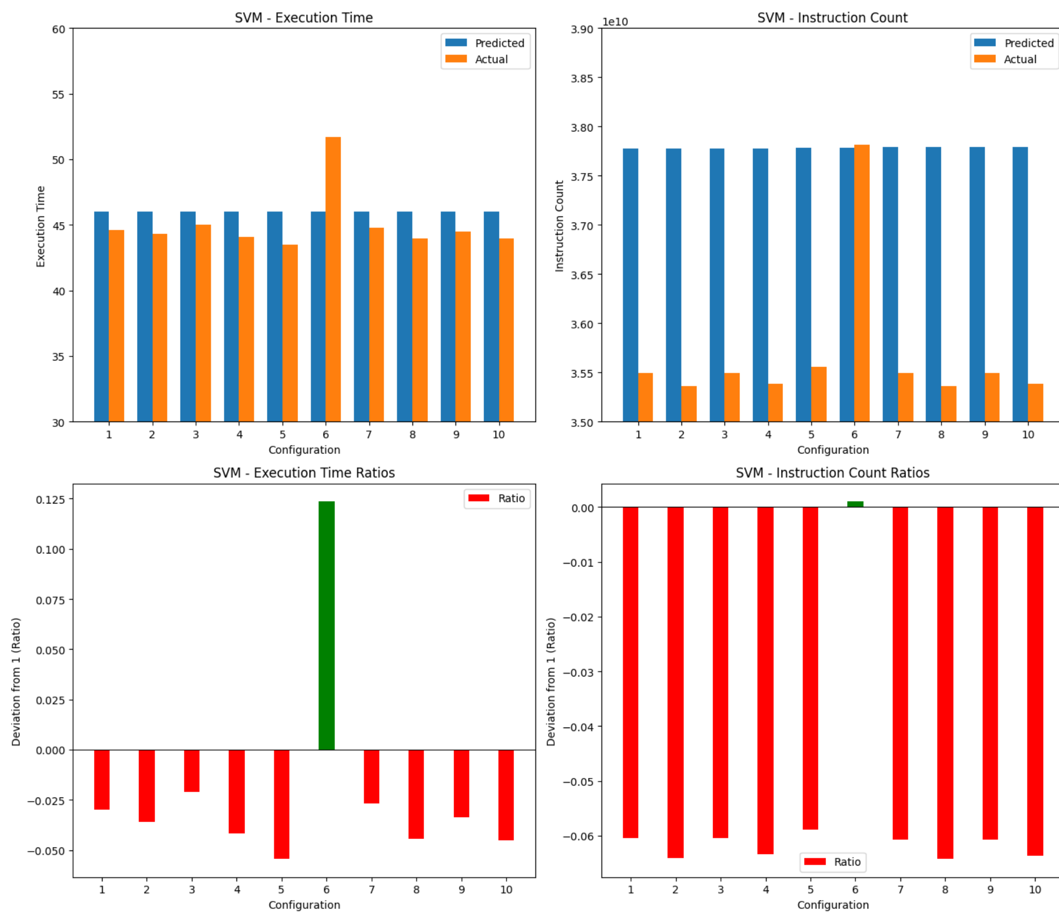

Figure 4.7: Results from SVM-N prediction

Config	L0	L1	L2	L3	L4	L5
1	0.93953	0.98821	0.97273	0.95732	0.96739	0.96065
2	0.93598	0.93874	0.92675	0.93300	0.95372	0.95229
3	0.93951	1.00699	1.01834	0.97942	1.16109	0.96472
4	0.93667	0.98046	0.95590	0.92215	0.91779	0.94434
5	0.94115	0.96377	0.93521	0.91234	0.91646	0.92573
6	1.00102	1.25732	1.21880	1.21071	1.21610	1.26015
7	0.93926	0.99820	0.99193	0.95487	0.95689	0.97703
8	0.93572	0.93066	1.01951	0.93391	0.93587	0.93984
9	0.93925	0.98895	0.98374	0.94422	0.95362	0.99704
10	0.93640	0.95565	0.95277	0.92897	0.93453	0.95160
Avg	9.44E-01	1.00E+00	9.98E-01	9.68E-01	9.91E-01	9.87E-01

Table 4.22: SVM-N layer instruction count predictions ratios (L0 to L5)

Config	L6	L7	L8	L9	L10	L11	L12
1	0.95511	0.96739	0.98043	0.97033	0.93337	0.93774	0.93524
2	0.97065	0.93314	0.95811	0.96382	0.93139	0.92789	0.94169
3	0.95826	0.97320	0.96957	0.98478	0.97737	0.96208	0.98385
4	0.94133	0.94269	0.96617	0.96637	0.90990	0.91059	0.90915
5	0.95426	0.92661	0.94829	0.92216	0.92111	0.92603	0.92567
6	1.26452	1.24363	1.27705	1.27465	1.25705	1.25433	1.25361
7	0.99844	0.97106	0.98372	0.97131	0.94639	0.94313	0.96471
8	0.93598	0.93119	0.95520	0.94590	0.96093	0.95598	0.96303
9	0.98723	0.95287	0.96438	0.95578	0.94687	0.95116	0.95242
10	0.94211	0.92824	1.55161	0.94143	0.91806	0.91407	0.91004
Avg	9.91E-01	9.77E-01	1.58E+00	9.90E-01	9.70E-01	9.68E-01	9.74E-01

Table 4.23: SVM-N layer instruction count predictions ratios (L6 to L12)

4.2.1.4 Random Forest-N model

Table 4.24 shows the ten best configurations suggested by the Random Forest-N model. In Figure 4.8 we see the resulting execution times and instruction counts for the configurations suggested by the Random Forest-N model. Finally, Tables 4.25 and 4.26 show a breakdown for the individual layers' instruction count ratios.

Configuration	N1	N2	N3	N4	N5	Unroll
1	6272	12544	3136	784	49	24
2	6272	12544	3136	784	196	24
3	6272	12544	3136	784	98	24
4	6272	6272	3136	784	49	24
5	6272	6272	3136	784	98	24
6	25088	6272	3136	784	196	24
7	25088	12544	3136	784	196	24
8	25088	12544	3136	784	49	24
9	25088	6272	3136	784	49	24
10	25088	6272	3136	784	98	24

Table 4.24: Random Forest-N configuration predictions

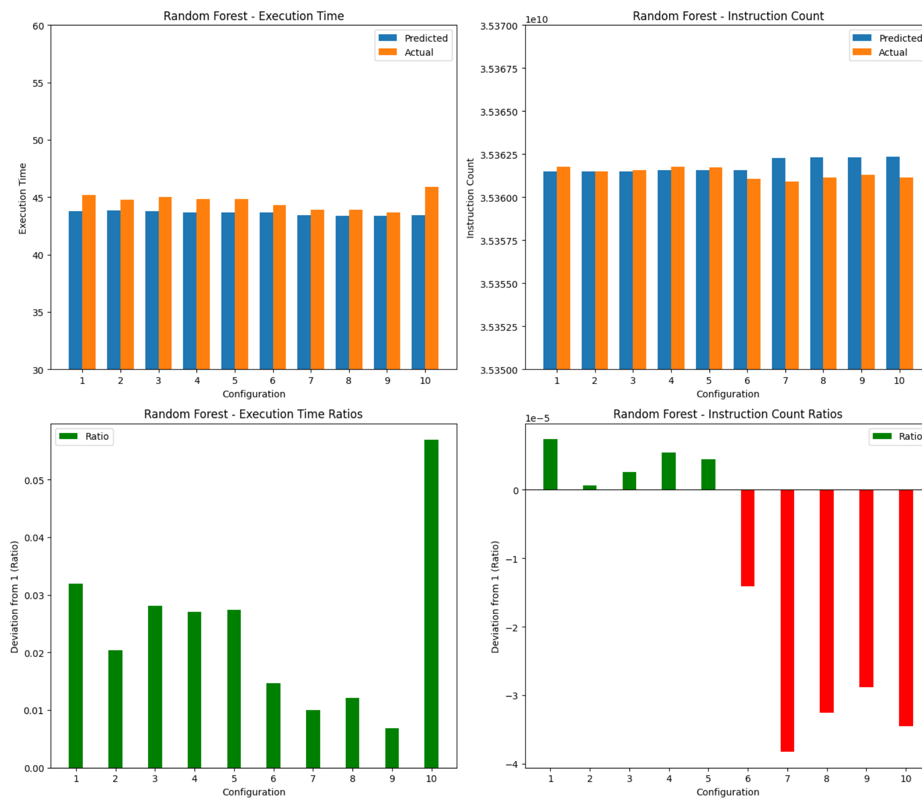

Figure 4.8: Results from Random Forest-N prediction

Config	L0	L1	L2	L3	L4	L5
1	1.00000	1.03040	1.15119	1.02898	0.99409	1.02745
2	1.00000	1.03899	1.02968	1.02799	0.98492	1.00568
3	1.00000	1.02343	1.03652	1.01512	1.00056	1.00889
4	1.00001	1.00200	1.00941	1.01370	1.00142	0.99990
5	1.00000	0.99445	1.02330	1.01783	0.96656	0.99238
6	0.99998	0.99269	0.99006	0.98615	1.00027	0.99681
7	0.99997	1.00306	0.99127	1.00207	1.00471	0.97596
8	0.99997	1.00933	1.00620	1.00198	1.01388	0.99524
9	0.99997	1.00507	1.00645	1.01151	1.02757	0.99449
10	0.99997	1.03349	1.01381	1.02387	0.99780	1.09305
Avg	1.00E+00	1.01E+00	1.03E+00	1.01E+00	9.99E-01	1.01E+00

Table 4.25: Random Forest-N layer instruction count predictions ratios (L0 to L5)

Config	L6	L7	L8	L9	L10	L11	L12
1	1.03061	0.98557	0.98390	1.00160	1.07387	1.03247	1.03805
2	1.01692	1.02296	1.01848	1.00352	1.01324	1.05168	1.01295
3	1.00970	1.02915	1.04170	1.01285	1.06603	1.06603	1.03645
4	1.01509	0.97981	0.99015	0.96637	1.02852	1.03642	1.02957
5	1.01414	0.97892	0.99382	0.92216	1.02113	1.00864	1.00445
6	1.01831	0.99816	0.99475	1.27465	1.06755	1.03566	1.00804
7	0.99554	0.99341	0.99440	0.97131	1.00325	1.00013	1.01584
8	1.00128	0.98697	1.00560	0.94590	1.00507	1.00222	1.01499
9	0.99898	0.99315	0.99897	0.95578	1.00210	1.00256	0.99910
10	1.06899	0.99055	0.98215	0.94143	1.01595	1.01342	1.01359
Avg	1.02E+00	9.96E-01	1.00E+00	9.99E-01	1.02E+00	1.02E+00	1.02E+00

Table 4.26: Random Forest-N layer instruction count predictions ratios (L6 to L12)

4.2.2 Modeling all 39 parameters

4.2.2.1 ANN-39 model

Table 4.27 shows the ten best configurations suggested by the ANN-39 model. In Figure 4.9 we see the resulting execution times and instruction counts for the configurations suggested by the ANN-39 model. Finally, Tables 4.28 and 4.29 show a breakdown for the individual layers' instruction count ratios.

Configuration	N1	N2	N3	N4	N5	Unroll
1	1	32	1	32	1	1
2	1	32	1	32	2	1
3	1	32	1	32	4	1
4	2	32	2	32	1	1
5	1	32	1	32	8	1
6	1	32	2	32	2	1
7	2	32	2	32	2	1
8	1	32	1	32	4	1
9	2	32	2	32	4	1
10	1	32	2	32	4	1

Table 4.27: ANN-39 configuration predictions

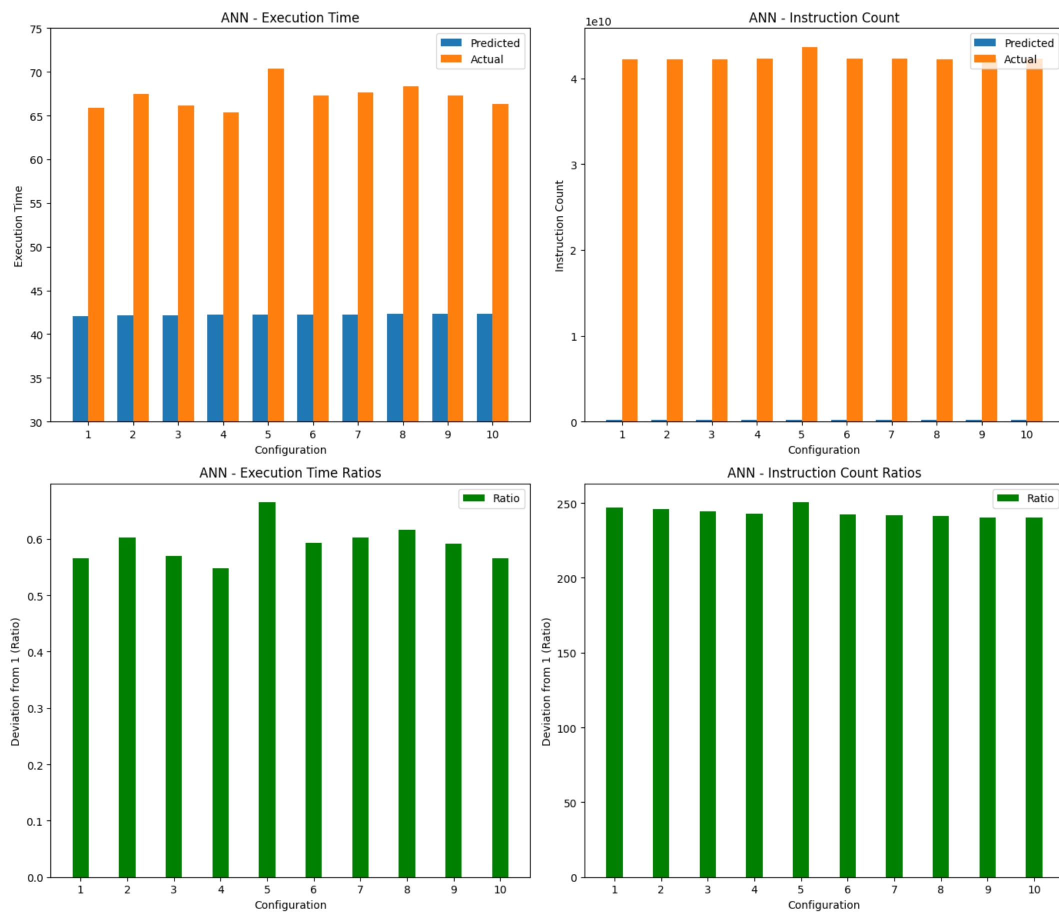

Figure 4.9: Results from ANN-39 prediction

Config	L0	L1	L2	L3	L4	L5
1	248.00065	88.24028	46.54587	92.84624	42.14797	82.81346
2	247.17769	91.45804	48.35592	95.56825	41.98263	82.40043
3	245.61113	87.50492	45.86855	92.54452	41.63171	80.60050
4	243.74054	84.87257	45.13195	91.62394	41.28968	83.44079
5	251.44437	86.29793	45.82176	92.16636	41.36907	82.09223
6	243.08689	88.47006	46.18090	93.29124	44.75248	82.50508
7	242.93634	89.55523	46.14128	94.19410	41.89936	81.61063
8	242.48775	86.44984	46.11955	93.29615	41.32558	81.09064
9	241.40358	91.34049	46.22033	92.55358	41.47835	81.83567
10	241.25666	86.67472	45.53634	91.80631	41.39983	80.70455
Avg	2.45E+02	8.81E+01	4.62E+01	9.30E+01	4.19E+01	8.19E+01

Table 4.28: ANN-39 layer instruction count predictions ratios (L0 to L5)

Config	L6	L7	L8	L9	L10	L11	L12
1	97.46037	96.48565	190.27250	181.63697	37.47465	41.75913	33.73647
2	105.98651	100.43824	195.35433	190.86880	37.51238	42.01405	33.60486
3	99.24445	98.77171	189.24859	178.27829	37.26781	41.76301	33.58788
4	100.75072	94.06497	183.99458	175.46346	37.74347	42.15132	33.46984
5	100.89711	94.98899	185.02912	177.98161	81.91812	91.27483	73.47952
6	100.00207	96.13391	187.24468	177.49263	37.10930	41.75993	33.57646
7	99.18812	95.51195	185.04865	176.67056	37.33905	41.16766	33.32482
8	97.92799	96.70640	186.25908	183.02879	37.23646	41.30073	33.32364
9	98.78616	94.49297	185.45701	177.70250	37.54091	43.32380	34.00597
10	98.68361	95.72661	186.21508	174.64802	36.63750	41.09191	33.16947
Avg	9.99E+01	9.63E+01	1.87E+02	1.79E+02	4.18E+01	4.68E+01	3.75E+01

Table 4.29: ANN-39 layer instruction count predictions ratios (L6 to L12)

4.2.2.2 Linear Regression-39 model

Table 4.30 shows the ten best configurations suggested by the Linear Regression-39 model. In Figure 4.10 we see the resulting execution times and instruction counts for the configurations suggested by the Linear Regression-39 model. Finally, Tables 4.31 and 4.32 show a breakdown for the individual layers' instruction count ratios.

Configuration	N1	N2	N3	N4	N5	Unroll
1	1	1	1	1	1	32
2	2	1	1	1	1	32
3	1	2	1	1	1	32
4	4	1	1	1	1	32
5	2	2	1	1	1	32
6	1	1	2	1	1	32
7	2	1	2	1	1	32
8	4	2	1	1	1	32
9	1	2	2	1	1	32
10	1	4	1	1	1	32

Table 4.30: Linear Regression-39 configuration predictions

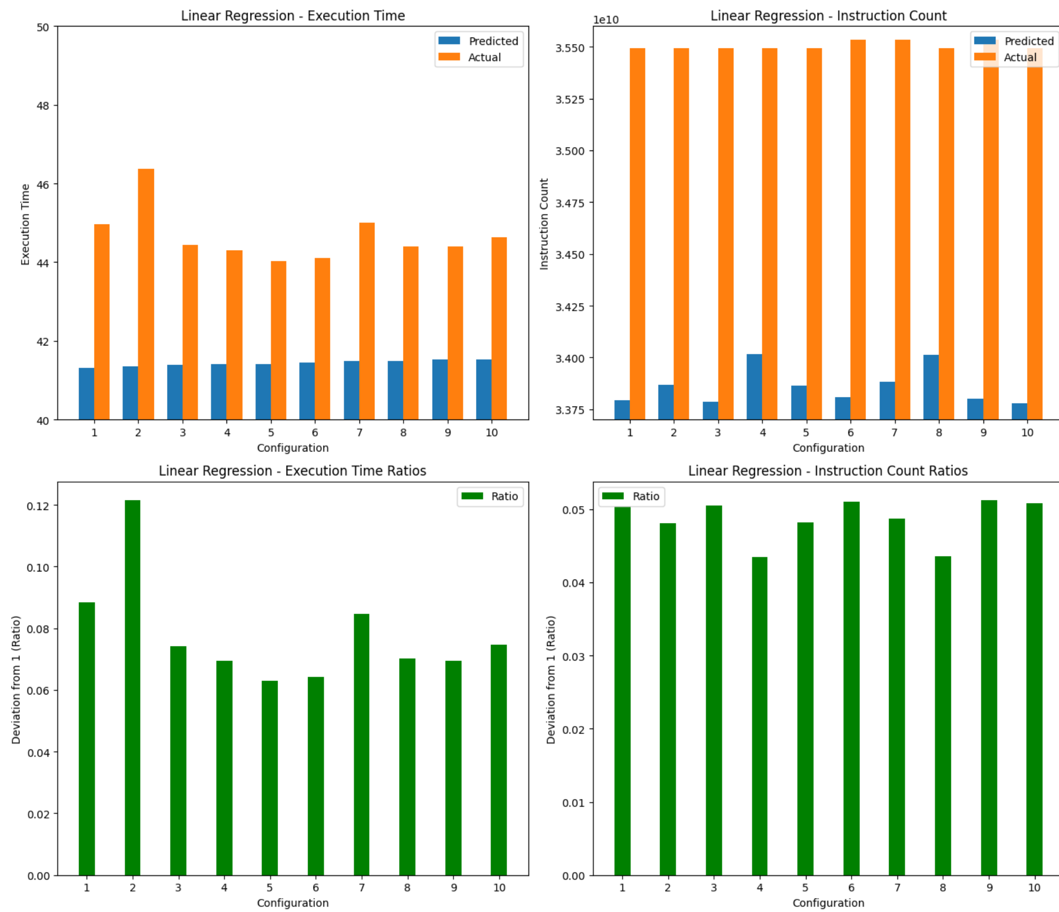

Figure 4.10: Results from Linear Regression-39 model prediction

Config	L0	L1	L2	L3	L4	L5
1	1.05038	1.14436	1.17274	1.12767	1.09528	1.11395
2	1.04806	1.18036	1.10502	1.10832	1.09618	1.10743
3	1.05053	1.14789	1.13133	1.12423	1.09767	1.14553
4	1.04343	1.10939	1.11377	1.11526	1.09417	1.12250
5	1.04821	1.10561	1.11296	1.11931	1.07932	1.09445
6	1.05106	1.12127	1.10580	1.10458	1.10312	1.11933
7	1.04874	1.13441	1.13003	1.11184	1.09784	1.13037
8	1.04358	1.13433	1.10897	1.09940	1.09438	1.11198
9	1.05121	1.11021	1.11145	1.11699	1.15058	1.14021
10	1.05084	1.12846	1.13049	1.10123	1.09002	1.13029
Avg	1.05E+00	1.13E+00	1.12E+00	1.11E+00	1.10E+00	1.12E+00

Table 4.31: Linear Regression-39 model layer instruction count predictions ratios (L0 to L5)

Config	L6	L7	L8	L9	L10	L11	L12
1	1.12880	1.08312	1.08578	1.08184	1.03340	1.04128	1.04024
2	1.14016	1.08792	1.07570	1.08064	1.03752	1.03568	1.04129
3	1.11482	1.06198	1.05066	1.06527	1.02677	1.03006	1.03567
4	1.13013	1.07140	1.06187	1.07510	1.04187	1.05498	1.04526
5	1.11915	1.06825	1.07120	1.08098	1.02777	1.03139	1.02801
6	1.13743	1.07247	1.06125	1.06860	1.02602	1.02622	1.02780
7	1.12139	1.07184	1.06959	1.08257	1.07576	1.05212	1.22774
8	1.11532	1.07788	1.07016	1.07662	1.04338	1.04844	1.04825
9	1.13024	1.06255	1.05364	1.06558	1.05325	1.05775	1.07184
10	1.14245	1.07414	1.07004	1.07186	1.02275	1.02207	1.02607
Avg	1.13E+00	1.07E+00	1.07E+00	1.07E+00	1.04E+00	1.04E+00	1.06E+00

Table 4.32: Linear Regression-39 model layer instruction count predictions ratios (L6 to L12)

4.2.2.3 SVM-39 model

Table 4.33 shows the ten best configurations suggested by the SVM-39 model. In Figure 4.11 we see the resulting execution times and instruction counts for the configurations suggested by the SVM-39 model. Finally, Tables 4.34 and 4.35 show a breakdown for the individual layers' instruction count ratios.

Configuration	N1	N2	N3	N4	N5	Unroll
1	1	1	1	1	1	32
2	2	1	1	1	1	32
3	1	2	1	1	1	32
4	1	1	2	1	1	32
5	2	2	1	1	1	32
6	2	1	2	1	1	32
7	1	2	2	1	1	32
8	4	1	1	1	1	32
9	1	1	1	2	1	32
10	1	4	1	1	1	32

Table 4.33: SVM-39 configuration predictions

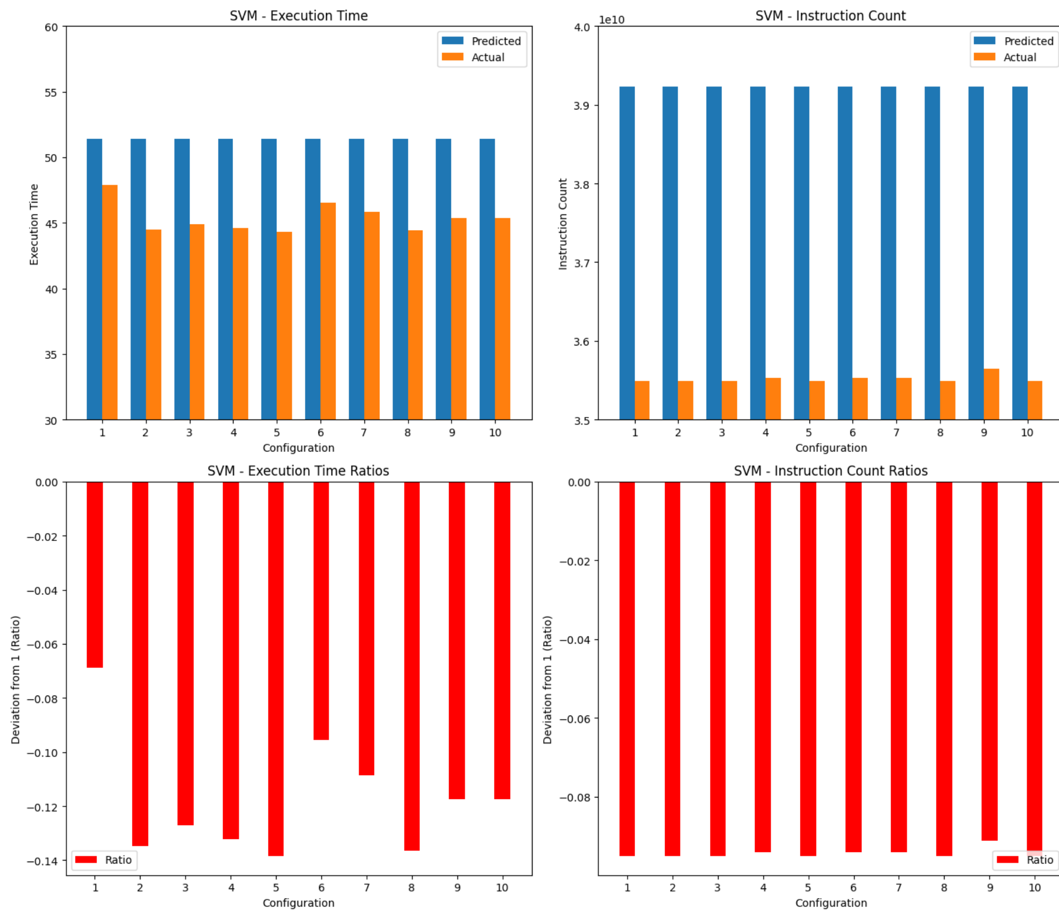

Figure 4.11: Results from SVM-39 model prediction

Config	L0	L1	L2	L3	L4	L5
1	0.90484	1.01121	1.00194	0.99733	0.92207	1.06995
2	0.90485	1.01579	0.89680	0.91274	0.88625	0.87204
3	0.90485	1.03864	0.98076	0.91538	0.87870	0.87121
4	0.90583	1.02265	0.90847	0.90076	0.89687	0.89299
5	0.90485	1.01447	0.90816	0.90574	0.89497	0.87664
6	0.90583	1.11581	0.93568	0.92017	0.92765	0.92645
7	0.90583	1.02573	0.92469	0.91563	0.91780	0.89123
8	0.90485	1.03046	0.90559	0.91007	0.86940	0.87108
9	0.90880	1.02581	0.90873	0.93215	0.89080	0.87497
10	0.90486	1.02906	0.92761	0.91130	0.89075	0.86927
Avg	9.06E-01	1.03E+00	9.30E-01	9.22E-01	8.98E-01	9.02E-01

Table 4.34: SVM-39 model layer instruction count predictions ratios (L0 to L5)

Config	L6	L7	L8	L9	L10	L11	L12
1	1.02583	0.91842	0.90529	0.84802	0.53142	0.53255	0.50864
2	0.89171	0.85567	0.88940	0.86604	0.51477	0.51306	0.50919
3	0.86897	0.84968	0.84070	0.84519	0.51414	0.51344	0.50747
4	0.89375	0.84799	0.84483	0.85064	0.51142	0.51347	0.50850
5	0.89572	0.84911	0.85816	0.86603	0.50880	0.51021	0.54207
6	0.91212	0.90360	0.85574	0.86637	0.51913	0.51604	0.51396
7	0.89830	0.86144	0.85680	0.85153	0.51332	0.51073	0.51008
8	0.87065	0.84528	0.83956	0.84048	0.50855	0.50823	0.50443
9	0.86633	0.91172	0.91734	0.93356	0.51002	0.50909	0.51250
10	0.86710	0.84664	0.85491	0.85711	0.52754	0.50651	0.50613
Avg	8.99E-01	8.69E-01	8.66E-01	8.62E-01	5.16E-01	5.13E-01	5.12E-01

Table 4.35: SVM-39 model layer instruction count predictions ratios (L6 to L12)

4.2.2.4 Random Forest-39 model

Table 4.36 shows the ten best configurations suggested by the Random Forest-39 model. In Figure 4.12 we see the resulting execution times and instruction counts for the configurations suggested by the Random Forest-N model. Finally, Tables 4.37 and 4.38 show a breakdown for the individual layers' instruction count ratios.

Configuration	N1	N2	N3	N4	N5	Unroll
1	4	2	1	1	4	24
2	4	2	1	1	2	24
3	4	2	1	1	1	24
4	8	1	1	1	4	24
5	8	1	1	1	2	24
6	2	2	1	1	2	24
7	2	2	1	1	4	24
8	2	2	1	1	1	24
9	4	1	1	1	2	24
10	4	1	1	1	4	24

Table 4.36: Random Forest-39 configuration predictions

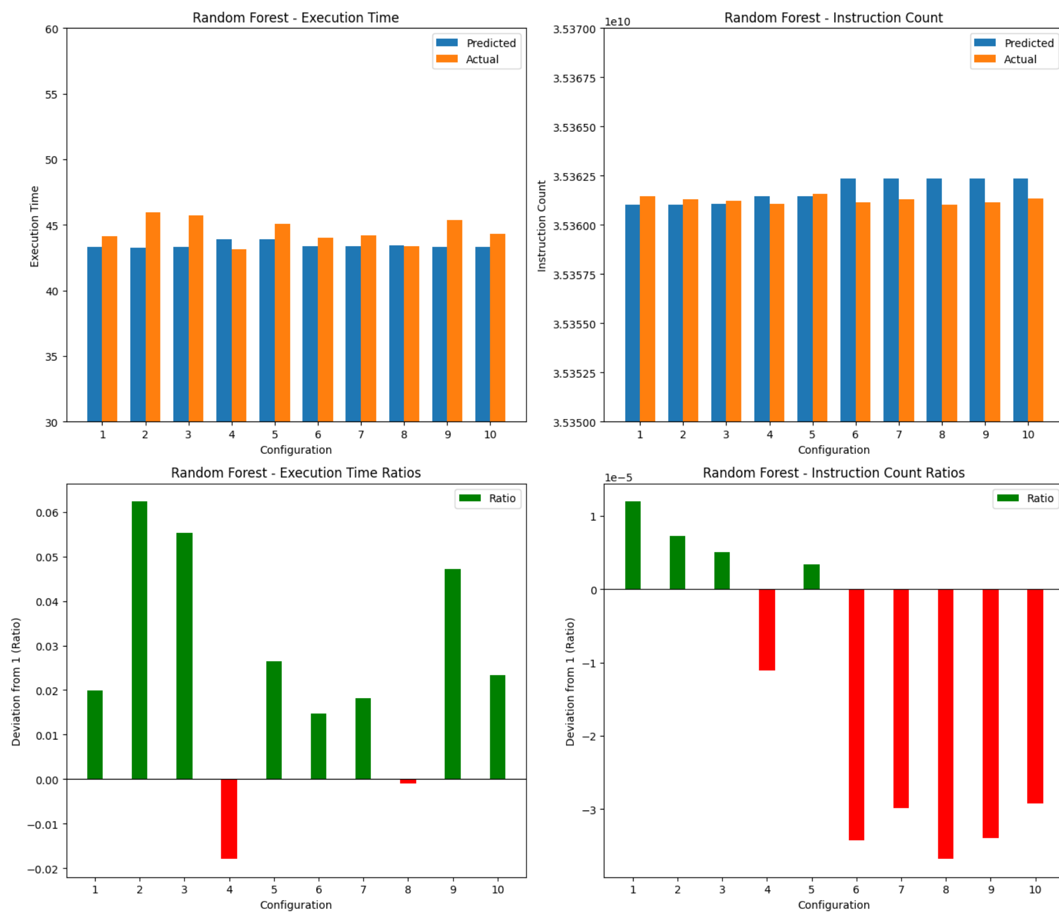

Figure 4.12: Results from Random Forest-39 model prediction

Config	L0	L1	L2	L3	L4	L5
1	0.99995	1.00321	1.02353	0.97385	0.98632	0.98780
2	0.99991	1.01050	1.00815	1.02241	1.02091	1.01000
3	0.99992	0.99520	1.03755	1.02112	0.99918	1.00297
4	0.99991	1.00843	1.03969	1.00808	1.01641	0.99367
5	0.99995	1.03134	1.00696	0.97319	0.99723	0.98836
6	0.99996	1.03576	1.00832	1.00052	0.99752	0.98732
7	1.00000	1.00954	0.99546	0.99350	1.00607	0.99449
8	0.99987	0.99970	1.05256	1.01622	1.03141	0.98786
9	0.99988	1.00636	1.01616	1.00106	0.99716	1.00048
10	0.99987	1.00334	1.03509	0.99774	1.01602	1.02311
Avg	1.00E+00	1.01E+00	1.02E+00	1.00E+00	1.01E+00	9.98E-01

Table 4.37: Random Forest-39 layer instruction count predictions ratios (L0 to L5)

Config	L6	L7	L8	L9	L10	L11	L12
1	0.98431	1.00309	1.02133	1.00061	1.00351	1.01201	0.99942
2	1.02643	1.02993	1.03291	1.01772	1.01165	1.01267	1.00190
3	0.99736	1.02114	1.00978	1.00883	1.01006	1.00674	1.00771
4	1.02311	0.99316	1.01537	1.01821	1.03523	1.01716	1.00225
5	0.98943	1.02036	0.99971	0.98970	0.99099	0.99681	0.99282
6	0.98165	0.98497	1.03502	1.06274	1.07927	1.03478	1.02776
7	0.99773	0.97548	0.98826	1.00752	1.00025	0.99848	0.99874
8	1.02226	1.00585	0.99626	0.98261	0.99010	0.98633	0.99039
9	1.00306	0.98853	1.00492	1.02125	1.00402	1.00120	1.00274
10	1.02608	1.00182	0.99582	0.98899	0.99461	0.99147	0.99395
Avg	1.01E+00	1.00E+00	1.01E+00	1.01E+00	1.01E+00	1.01E+00	1.00E+00

Table 4.38: Random Forest-39 layer instruction count predictions ratios (L6 to L12)

5

Discussion

5.1 Auto-tuning models

We begin with analyzing and discussing the results from the different auto-tuning models. First off, since we are emulating all configurations on RISC-V hardware through QEMU, the numbers are not necessarily representative of execution on real hardware as mentioned previously. However, we can see from the results that lower execution times and lower instruction counts are correlated. We can also see that the execution time predictions and instruction counts predictions are both more accurate, i.e., the ratios between prediction and actual values are closer to each other, as well as being lower in absolute terms. Furthermore, the methodology wouldn't change between using real hardware and emulating since we could gather execution times and instruction counts from the hardware to auto-tune the same parameters.

The ratio graphs (and tables) also give better insight into the accuracy of the predictions made by the different auto-tuning models. Mainly because the magnitude of the instruction counts are to the power of 10 and the absolute difference in counts may not give the full picture. For example, being off in a prediction by 1000 instructions may be a lot if the actual value is 100, but being off by the same 1000 instructions is very small in the case that the actual value is 1 million.

5.1.1 ANN

Breaking down the result from the ANN model. Both the predicted configurations were bad and the predicted values were bad. The predicted configurations resulted in the slowest actual execution times and largest actual instruction counts. With some being around 80 seconds for execution times and instructions count being around $4.5e10$. Not only were the configurations it predicted bad, the values it thought those configurations would have were off by almost 70% for some execution times and almost a factor of 45 for the instruction counts. The ANN model was the exact same configuration used by the previous OpenCL auto-tuning work mentioned earlier, implemented in this work because it worked well previously and would hopefully give a good reference point. Their methodology was followed, using the ANN to generate predictions, then taking the best of those configurations and and testing them to ultimately find the best one.

This discrepancy begs the question: what caused the poor performance? The simple answer is that the dataset used in this thesis is simply too small for the ANN model to perform well. The dataset is around 3 magnitudes smaller than the amount of possible configurations. This means that the ANN model overfits its predictions to the tested configurations and since we predict on unseen configurations they become very inaccurate. Even with the bagging technique used to try to reduce overfitting, by averaging the result of 11 different predictions, the model's results are bad. The bagging technique doesn't work because all 11 predictions are overfitted and thus the average is also overfitted and inaccurate.

Since the ANN model has been proven to be effective in other work, the conclusion from our result must be that given a larger dataset to train on, the ANN model could eventually be trained without being overfitted. For our use-case however, it is not the best model to work with. Especially since we want to reduce the amount of tests needed for good predictions, avoiding the need to do a full exhaustive search.

5.1.2 Linear Regression & SVM

These two models had quite similar performance. Looking at the instruction count ratios, we can see that both models were around 5% off which is quite small and much better than the ANN. We can also see that the linear regression model predicted less instructions than what the actual count was while the SVM predicted more instructions than were actually executed.

The SVM was more accurate concerning the execution time predictions. The linear regression being off by at least 6% and the SVM being off by at most almost 4% and almost being spot on for most. From the layer per layer breakdown of instruction count ratios, we can see that the linear regression consistently overestimated the amount of instructions to be executed. The SVM generally underestimated the amount of instructions but some layers were also overestimated.

Given the good performance by the linear regression, there definitely seems to be a linear correlation between the configurations and the execution times and instruction counts. However, there may be some non-linearity considering the SVM with a non-linear kernel performed better. But since they are relatively close, this non-linearity may not be the largest influence in the execution time or instruction count.

Moreover, even though the models' predicted best configurations perform quite similar to each other, it may be preferable to use the SVM since having a model that overestimates the resources it will take gives the user a margin of safety. In a performance critical situation we wouldn't want to be surprised that the execution takes a longer time than we expected. However, overestimating the time it will take may lead to resources being wasted since we have more resources available in actuality than we predicted. Even though it may not be critical to the scenarios we use CNNs for in this thesis, in other scenarios such as processing images gathered from a camera in a self-driving car the distribution of resources becomes more important. Can we improve further? Yes.

5.1.3 Random Forest

The random forest model had by far the best performance. The actual execution times and instruction counts by the predicted configurations were the lowest and the ratios between predictions and actual performance was the closest. With actual execution times from the predicted configurations being around 20%-25% faster than the non auto-tuned version. Moreover, the predicted execution times being off by at most 3% and the instruction counts being off by at most 0.012%, or approximately $\frac{1}{10000}$.

The random forest's result ensures that we receive the best configurations possible and that the predictions for those configurations are accurate. Regarding safety margins, the instruction counts are overestimated a minuscule amount meaning we are on the safe side of estimations and we can be sure to know how many resources our program will take beforehand.

In contrast with the ANN, the random forest avoids overfitting and predicting inaccurate results. However, even though the results are good, it may be biased towards configurations already present in the training set. Since the training dataset is so small, as mentioned previously in the ANN discussion, instead of overfitting and predicting that some unseen configurations will perform exceptionally well, it only predicts already seen configurations will perform well. Potentially leading to a limited ability to generalize to entirely new and unseen configurations.

5.2 Comparing with non auto-tuned executions

We can see immediately that even without auto-tuning, simply vectorizing the code and using RISC-V JIT instructions give a great performance increase to the naive implementation. For a slightly longer creation time we get an almost 15 times faster execution time. The instruction count is increased, however just looking at the total instruction count does not give the whole picture, since the total instruction count includes the instructions executed during creation time as well. What we can compare to get a better picture is the performance in comparison of the layers which are the instruction counts during execution.

5.2.1 Network-agnostic models

In the following subsections we will start comparing the two network-agnostic models to the original auto-tuner models. We will being with discussing the results for the models using the actual N values, then we continue with comparing to the models using the additional 39 layer parameters. In the following table the blocking factor models are named with no suffix and the models using the N values are named with a '-N'. Furthermore, the models using all the 39 additional layer parameters are named with a '-39'. For brevity, only convolutional layers 0-4 are summarized in the following table but all data is contained in the previous chapter.

Model	L0	L1	L2	L3	L4
ANN	4,39E+01	1,51E+01	8,09E+00	1,15E+01	1,10E+00
ANN-N	1,44E+02	5,38E+01	2,71E+01	6,03E+01	3,40E+01
ANN-39	2,45E+02	8,81E+01	4,62E+01	9,30E+01	4,19E+01
Linear Regression	1,05E+00	1,13E+00	1,12E+00	1,11E+00	1,10E+00
Linear Regression-N	1,12E+00	1,46E+00	1,44E+00	1,45E+00	1,42E+00
Linear Regression-39	1,05E+00	1,13E+00	1,12E+00	1,11E+00	1,10E+00
SVM	9,53E-01	1,02E+00	9,92E-01	1,01E+00	9,74E-01
SVM-N	9,44E-01	1,00E+00	9,98E-01	9,68E-01	9,91E-01
SVM-39	9,06E-01	1,03E+00	9,30E-01	9,22E-01	8,98E-01
Random Forest	1,00E+00	1,01E+00	1,02E+00	1,00E+00	1,01E+00
Random Forest-N	1,00E+00	1,01E+00	1,03E+00	1,01E+00	9,99E-01
Random Forest-39	1,00E+00	1,01E+00	1,02E+00	1,00E+00	1,01E+00

Table 5.1: Average ratio between prediction and actual instruction count for blocking factor auto-tuners, N value auto-tuners, and auto-tuners using the 39 additional layer parameters (L0 to L4)

5.2.2 Blocking factor vs actual N value

The main difference in the training data is essentially the magnitude of the parameter values. Since the training data is the same in every other aspect the immediate thought is that the performances shouldn't differ by that much.

We can see that the performances and accuracy of predictions by the models using the actual N values are worse than the models using the blocking factors. In general the same pattern emerges, i.e., that the random forest performs best in both cases and the ANN overfits to the training data. Furthermore, the random forest's accuracy doesn't seem to be affected by the difference in the training parameters. The SVM's performance seems to be improved for most layer predictions but not all.

We can break down and reason about why some of the models are more suited to using one of the versions of training parameters or the other. Starting with the ANN model, the larger magnitudes may cause the model to have a more difficult time converging to a stable result. The difference in possible values is quite stark, either between 1-32 or between 1-50176. This large span of input values may lead to unstable updates in the ANN's gradient function. Contrast that with the ANN model using the blocking factor, in comparison it is essentially working with normalized input parameters. This may lead to it converging faster to a stable result.

Continuing with the linear regression. It also suffers from the large magnitudes of the input parameters. Mainly, input parameters with large scales, e.g., 1-50176, can negatively impact the model coefficients. Especially when one layer's N value is the maximum of 50176 and another layer has a small N value of 196. Then the model may make one or more of the coefficients a disproportionate size compared to the real impact it has on the instruction counts.

The SVM has quite similar performance between versions. Perhaps surprising con-

sidering it tries to fit a hyperplane to the data points, in a way a multidimensional version of linear regression. However, there is an important difference in the way linear regression tries to fit the line and the way the SVM tries to fit the hyperplane. That is the fact that the SVM uses a 'rbf' kernel which transforms the data to a higher dimension to find non-linear relations. This transformation may lessen the importance of the input scales, making the SVM model more robust to the large differences between input values.

Analyzing the random forest models' performances we can see that have minimal differences in accuracy. The blocking factor version being off by at most around 2% and the one using the N values being off by at most 3%. This is not too surprising. Random forests use decision trees that split on the different values and the absolute differences don't matter as much. I.e., the fact that 196 is smaller than 50176 is more important the the fact that they differ by 49980. Therefore, the fact that the random forest had a very good predictive power for one set of input parameters was a strong indicator that it would have a very good predictive power for the other set. Especially considering the target values of instruction counts stayed the same between the two versions.

5.2.3 Blocking factor vs 39 additional parameters

In table 5.1 we can see that the ANN model performs even worse when we introduce the new parameters. The reason for this degradation in performance is most likely due to the fact that the 39 parameters are static since we only train on VGG16. In other words we have 39 parameters that do not change for any of the entries. This means that they introduce complexity to the model without any benefits, leading to overfitting and poor generalization. In theory, the benefit of training on the layer parameters is that we could change one or more layers and still predict the performance of them. Meaning we generalize the model's prediction power to multiple networks. However, in the case of the ANN model, it seems the benefit of being able to switch layer configurations is outweighed by the much worse performance for the specific network we use.

The linear regression had the exact same predictions for the '-39' version and the original version. Quickly comparing it to the '-N' version we can draw the conclusion that the magnitudes of the N values have a greater affect on the predictive power than anything else. This is also quite a reasonable result if we consider the way linear regression works.

As mentioned previously in section 2.1.6 with the example of predicting house prices, linear regression attempts to find the coefficients that best fit the line to the data. The addition of 39 static parameters will essentially not affect this process at all. Since they stay the same for every entry in the dataset, the only coefficients that have any importance are the exact same as for the original linear regression. These are the blocking factors and the unroll factor. Since they are the exact same values as the original the coefficients the linear regression finds are the same, producing the exact same predictions.

However, we cannot draw any conclusions in the prediction power for unseen layers from this result. In other words ability to transfer the predictions to other networks other than the fact that we could do it. The one caveat for this and for all the other models as well is that the new network we want to try to predict for needs to have the same amount of layers as VGG16. Otherwise, if the network has fewer layers, then we need to add dummy layers so that the model can do a prediction and in the reverse situation with more layers we would have to remove some layers to fit the auto-tuner's input parameters.

The SVM-39 has the worst performance of all the different SVM models. The reason for this is likely that the additional parameters make the process of fitting a hyperplane more complex. Even though the parameters are static, the SVM needs to fit a hyperplane to a total of 45 parameters. Moreover, it transforms the parameters to a higher dimension when it applies the rbf kernel.

The random forest has for all intents and purposes the exact same performance as previous versions. Similarly to the linear regression, since the 39 parameters are static, the only parameters that affect the model's predictions are the blocking factors and the unroll factor. Furthermore, we suffer from the same problem as the linear regression in that we cannot draw a conclusion of its prediction power for unseen layers.

5.2.4 Network-agnostic models summary

It does not seem beneficial to try to make the auto-tuner work on multiple networks from one training data set. Regardless of model, even if the performance is quite bad as in the case for the ANN, the versions being trained specifically on VGG16 performs much better.

The fact that the models aren't actually evaluated on different networks doesn't change the fact that their performance on the one model they are trained on degrades or is unaffected. In other words, we increase the data- and training complexity for the models for either no improvement in performance or for worse performance.

Consider the two models that were unaffected by the addition of 39 parameters, linear regression and random forest. Since they haven't seen any other layer configurations than VGG16, any decisions that they will take regarding unseen network layer configurations will have no basis in any concrete data. Even if we assume they will have a similar accuracy on unseen networks as for VGG16, it seems more efficient to train a model for a specific network. The reasoning here is that to be able to take the whole network configuration into consideration we had to add 39 new parameters. This increases the training input parameters from 6 to 45, an increase by a factor of 7.5. Additionally, if we wanted to add the amount of tuning parameters, such as looking at M and K as well as N or unrolling each layer separately, then we would have an exponential increase of parameters which is not sustainable.

Moreover, training the models on different networks requires the different networks we want the auto-tuner to work on needs to share the same amount of layer groups. In the case of VGG16 we can reduce the 13 layers to 5 groups that share the same N

values after applying im2col. If we want the auto-tuner to work with a model that doesn't share this property of 5 groups we need to either add dummy data to fill out the missing groups or we need to remove some groups due to having too many. In both cases this makes the predictions unreliable since they do not reflect reality.

5.3 Winograd

The Winograd implementation was not a success. This is due to two main factors: the fact that the RISC-V JIT requires the programmer to manually keep track of the registers and the complexity of the Winograd algorithm. Furthermore, this complexity caused the creation times for the network using Winograd to become a couple of magnitudes larger than the creation time for the network using im2col+GEMM, rendering data collecting impossible given the time of this thesis.

To illustrate the problem of large creation times here is an example of converting a simple nested for loop to JIT code. For simplicity's sake, imagine every instruction takes 1 time unit to execute and every JIT instruction takes 1 time unit to add. When we consider creation time we will simplify and only consider additional time taken due to adding JIT instructions.

```
for(int i = 0; i < 10; i++){
    for(int j = 0; j < 10; j++){
        // Loop Body
        // Assume 10 instructions
    }
}
```

This small example would take $10 * 10 * 10 = 1000$ time units to execute in our scenario, with a creation time of 0 since we don't push any JIT instructions. Let's say we convert only the inner loop and its contents to JIT code:

```
for(int i = 0; i < 10; i++){
    general_register j;
    general_register j_end;
    load_constant(j, 0);
    load_constant(j_end, 10);
    L("JLOOP");
        // Loop Body
        // Assume 10 JIT converted instructions
    addi(j, j, 1);
    blt(j, j_end, "JLOOP");
}
```

This example will be equivalent to unrolling the inner loop 10 times because the outer loop will push the inner loop instructions each iteration. Assuming the definitions of the variables take no time, the total time we have now is $10 * (10 + 5) = 150$ time units for creation time and $10 * 10 * 10 = 1000$ time units for execution time. Of course in these examples we don't vectorize the code which would remove a

substantial amount of execution time.

Converting the code fully to JIT code, which would be the goal, would give this:

```
general_register i;
general_register i_end;
load_constant(i, 0);
load_constant(i_end, 10);
L("ILOOP");
    general_register j;
    general_register j_end;
    load_constant(j, 0);
    load_constant(j_end, 10);
    L("JLOOP");
        // Loop Body
        // Assume 10 JIT converted instructions
    addi(j, j, 1);
    blt(j, j_end, "JLOOP");
addi(i, i, 1);
blt(i, i_end, "ILOOP");
```

This results in a creation time of $10 + 5 + 5 = 20$ time units and an execution time of $10 * 10 * 10 = 1000$ time units.

As stated, in the thesis the code has been vectorized thanks to JIT instructions which means that the execution times are lowered as a result of fewer instructions being executed and so the trade-off is higher creation time for lowered execution time. However, the important point here is that if we do not convert all the code to JIT instructions but keep some for loops we get an exponential creation time increase per for loop left.

Essentially, the problem with the Winograd algorithm for this thesis has been that manually keeping track of registers made the conversion of for loops and loop bodies very complex meaning that some for loops were left unconverted. This in turn lead to large creation times meaning gathering enough data points for auto-tuning purposes became infeasible.

Does this mean that continuing with Winograd using the RISC-V JIT is a bad idea or impossible? The short answer is simply no. One solution would be to keep the generated Winograd kernels for one layer for successive layers. Especially in the case of VGG16 and other CNNs that have multiple layers that share the same dimensions. In other words, to pre-generate the kernels for multiple layers in the network. Another solution would be to convert all of the Winograd code to JIT instructions which was initially the goal in this thesis but which wasn't feasible in the time frame.

6

Conclusion

In this chapter we will draw conclusions based on the previous discussion and answer if the research questions and goals stated in section 1.2 were achieved and met.

6.1 Continuing with oneDNN

As mentioned in the beginning of this thesis, oneDNN is a proven backend for many popular deep learning applications and libraries such as PyTorch and Tensorflow. Its more common usages is for optimized algorithms on x86 CPUs and on some GPUs. The question is therefore if it is beneficial to continue working with oneDNN for RISC-V and more specifically for the RISC-V Vector Extension.

Without the use of auto-tuning and only considering using vectorized instructions we saw that the creation time for the VGG16 network was increased by about 25% while decreasing the execution time by a factor of 14. Using the vectorized instructions in oneDNN is without a doubt worth it.

Even though the Winograd convolution was unsuccessful vectorizing im2col and GEMM was relatively straight forward after learning of the RISC-V JIT worked. Furthermore, expanding the JIT with more instructions was also very easy. There is also room for improvement in the work done in this thesis. For example, looking into reusing kernels between layers to decrease the creation time and implementing techniques other than im2col+GEMM. Therefore, we can only conclude that there is a lot of potential in continuing with oneDNN.

6.2 Auto-tuning model success

The thesis was successful in developing a state-of-the-art method for auto-tuning im2col+GEMM with consistent and accurate predictions without having to do an exhaustive search. Of the four different auto-tuning models tested the random forest performed the best. The predictions the model made on never before seen configurations were in the range of 2 percentage points of the actual values on average.

The nature of the random forest means it is able to predict accurately using both the blocking factors and implicitly learning the network characteristics and also using the actual network layer sizes and explicitly learn the network characteristics and

how they affect the execution metrics. The results also show that training the model on specifically one network is more beneficial than attempting to have the model be good at predicting multiple different networks.

The linear regression and SVM were not as successful but still at most being off by 13 percentage points on average. Meaning they were still quite suited to the data set used in this thesis. The ANN on the other hand was not well suited to the thesis, being off by 4000% in some cases. The reasons for each models performance was discussed in depth previously but off note is that the more understandable models had better performance than the black box model.

6.3 Future Work

The most immediate topic for consideration for future work is the Winograd implementation. It was not successfully implemented in this thesis but Winograd is still a powerful convolution algorithm. As discussed, methods to pre-generate and reuse kernels and to speed up the creation times for the networks would be beneficial to make the testing process more efficient.

Furthermore, when it comes to the auto-tuner adding more parameters to auto-tune such as looking at the M and K parameters from im2col would increase the amount of possible configurations and may lead to even greater time savings. Additionally, doing even more tests to give more training data would avoid issues such as overfitting or bias to already seen configurations.

Lastly, running the code on real hardware would give the best and most accurate picture of the performance, both of the algorithms and of the auto-tuner. Running the algorithms on real hardware would also enable gathering more data to tune on. Meaning not only more accurate execution times and instruction counts but also enabling gathering other statistics such as memory usage and cache misses.

Bibliography

[1] L. Alzubaidi, J. Zhang, A. J. Humaidi, *et al.*, "Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions," en, *J. Big Data*, vol. 8, no. 1, p. 53, Mar. 2021.

[2] K. O'Shea and R. Nash, *An introduction to convolutional neural networks*, 2015. arXiv: 1511.08458 [cs.NE].

[3] A. d. L. Santana, A. Armejach, and M. Casas, "Efficient direct convolution using long simd instructions," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '23, Montreal, QC, Canada: Association for Computing Machinery, 2023, pp. 342–353. DOI: 10.1145/3572848.3577435. [Online]. Available: https://doi.org/10.1145/3572848.3577435.

[4] S. R. Gupta, N. Papadopoulou, and M. Pericàs, "Accelerating cnn inference on long vector architectures via co-design," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 145–155. DOI: 10.1109/IPDPS54959.2023.00024.

[5] Y. Xu, S. Martínez-Fernández, M. Martinez, and X. Franch, *Energy efficiency of training neural network architectures: An empirical study*, 2023. arXiv: 2302.00967 [cs.LG].

[6] E. P. Initiative. (), [Online]. Available: https://www.european-processor-initiative.eu/accelerator/.

[7] T. E. P. Project. "Software." (), [Online]. Available: https://eupilot.eu/software/.

[8] *oneAPI Deep Neural Network Library Developer Guide and Reference oneDNN v3.5.0 documentation.* [Online]. Available: https://oneapi-src.github.io/oneDNN/index.html (visited on 02/22/2024).

[9] N. Tollenaere, G. Iooss, S. Pouget, *et al.*, "Autotuning convolutions is easier than you think," *ACM Trans. Archit. Code Optim.*, vol. 20, no. 2, Mar. 2023, ISSN: 1544-3566. DOI: 10.1145/3570641. [Online]. Available: https://doi.org/10.1145/3570641.

[10] T. Chen, L. Zheng, E. Yan, *et al.*, *Learning to optimize tensor programs*, 2019. arXiv: 1805.08166 [cs.LG].

[11] G. Alaejos and A. Castelló, *Automatic generators for a family of matrix multiplication routines with apache tvm*, Sep. 2023. arXiv: 2310.20347 [cs.CL].

[12] S. R. Gupta, N. Papadopoulou, and M. Pericàs, "Challenges and opportunities in the co-design of convolutions and risc-v vector processors," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance*

Computing, Network, Storage, and Analysis, ser. SC-W '23, Denver, CO, USA: Association for Computing Machinery, 2023, pp. 1550–1556. DOI: 10.1145/3624062.3624232. [Online]. Available: https://doi.org/10.1145/3624062.3624232.

[13] T. L. Falch and A. C. Elster, "Machine Learning Based Auto-tuning for Enhanced OpenCL Performance Portability," en, in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, arXiv:1506.00842 [cs], May 2015, pp. 1231–1240. DOI: 10.1109/IPDPSW.2015.85. [Online]. Available: http://arxiv.org/abs/1506.00842 (visited on 04/29/2024).

[14] *OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems*, en, Section: API, Jul. 2013. [Online]. Available: https://www.khronos.org// (visited on 04/29/2024).

[15] *HiPerCoRe/KTT*, original-date: 2017-01-18T10:36:30Z, Mar. 2024. [Online]. Available: https://github.com/HiPerCoRe/KTT (visited on 04/29/2024).

[16] F. Petrovi, D. Stelák, J. Hozzová, *et al.*, "A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit," *Future Generation Computer Systems*, vol. 108, pp. 161–177, Jul. 2020, ISSN: 0167-739X. DOI: 10.1016/j.future.2020.02.069. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19327360 (visited on 04/29/2024).

[17] *Gptune/GPTune*, Apr. 2024. [Online]. Available: https://github.com/gptune/GPTune (visited on 04/29/2024).

[18] Y. Liu, W. M. Sid-Lakhdar, O. Marques, *et al.*, "GPTune: Multitask learning for autotuning exascale applications," en, in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Virtual Event Republic of Korea: ACM, Feb. 2021, pp. 234–246, ISBN: 978-1-4503-8294-6. DOI: 10.1145/3437801.3441621. [Online]. Available: https://dl.acm.org/doi/10.1145/3437801.3441621 (visited on 04/29/2024).

[19] S. R. P, *Different implementations of the ubiquitous convolution*, en, Nov. 2022. [Online]. Available: https://medium.com/@sundarramanp2000/different-implementations-of-the-ubiquitous-convolution-6a9269dbe77f (visited on 04/28/2024).

[20] A. Lavin and S. Gray, *Fast algorithms for convolutional neural networks*, 2015. arXiv: 1509.09308 [cs.NE].

[21] M. S. B. Maind and M. P. Wankar, "Research Paper on Basic of Artificial Neural Network," en, *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 2, no. 1, pp. 96–100, Jan. 2014, Number: 1, ISSN: 2321-8169. DOI: 10.17762/ijritcc.v2i1.2920. [Online]. Available: https://ijritcc.org/index.php/ijritcc/article/view/2920 (visited on 05/02/2024).

[22] J. Schmidhuber, "Deep Learning in Neural Networks: An Overview," en, *Neural Networks*, vol. 61, pp. 85–117, Jan. 2015, arXiv:1404.7828 [cs], ISSN: 08936080. DOI: 10.1016/j.neunet.2014.09.003. [Online]. Available: http://arxiv.org/abs/1404.7828 (visited on 05/06/2024).

[23] Z. Liu, Y. Qiu, and S. Jafarinejad, "12 - Artificial intelligence application to the nexus of renewable energy, water, and the environment," in *The Re-*

newable Energy-Water-Environment Nexus, S. Jafarinejad and B. S. Beckingham, Eds., Elsevier, 2024, pp. 399–422, ISBN: 978-0-443-13439-5. DOI: 10.1016/B978-0-443-13439-5.00012-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780443134395000120 (visited on 05/06/2024).

[24] *Activation Function - an overview | ScienceDirect Topics*. [Online]. Available: https://www.sciencedirect.com/topics/engineering/activation-function (visited on 05/06/2024).

[25] L. Breiman, "Random Forests," en, *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001, ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. [Online]. Available: https://doi.org/10.1023/A:1010933404324 (visited on 05/06/2024).

[26] *What Is Linear Regression? | IBM*, en-us, Aug. 2021. [Online]. Available: https://www.ibm.com/topics/linear-regression (visited on 05/22/2024).

[27] *All You Need to Know About Support Vector Machines*, en-US. [Online]. Available: https://www.spiceworks.com/tech/big-data/articles/what-is-support-vector-machine/ (visited on 05/22/2024).

[28] K. Simonyan and A. Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, arXiv:1409.1556 [cs], Apr. 2015. DOI: 10.48550/arXiv.1409.1556. [Online]. Available: http://arxiv.org/abs/1409.1556 (visited on 02/09/2024).

[29] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The risc-v instruction set manual, volume i: Base user-level isa," Tech. Rep. UCB/EECS-2011-62, May 2011. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html.

[30] R. O'Connor, *RISC-V Offers Simple, Modular ISA RISC-V International*, en-US. [Online]. Available: https://riscv.org/announcements/2016/04/risc-v-offers-simple-modular-isa/ (visited on 05/01/2024).

[31] *Riscv-v-spec/v-spec.adoc at master ů riscv/riscv-v-spec*, en. [Online]. Available: https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc (visited on 04/30/2024).

[32] *Riscv-opcodes/rv_v at master ů riscv/riscv-opcodes*. [Online]. Available: https://github.com/riscv/riscv-opcodes/blob/master/rv_v (visited on 03/26/2024).

[33] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," en, 2005.

[34] *QEMU TCG Plugins QEMU documentation*. [Online]. Available: https://www.qemu.org/docs/master/devel/tcg-plugins.html (visited on 04/28/2024).

[35] *Roger Ferrer / llvm-epi ů GitLab*, en, 2024. [Online]. Available: https://repo.hca.bsc.es/gitlab/rferrer/llvm-epi (visited on 05/08/2024).

[36] *Index of /epi/ftp*. [Online]. Available: https://ssh.hca.bsc.es/epi/ftp/ (visited on 03/26/2024).

[37] *Riscv-collab/riscv-gnu-toolchain*, original-date: 2014-09-08T05:22:03Z, May 2024. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain (visited on 05/08/2024).

[38] *QEMU*. [Online]. Available: https://www.qemu.org/.

A

Appendix 1