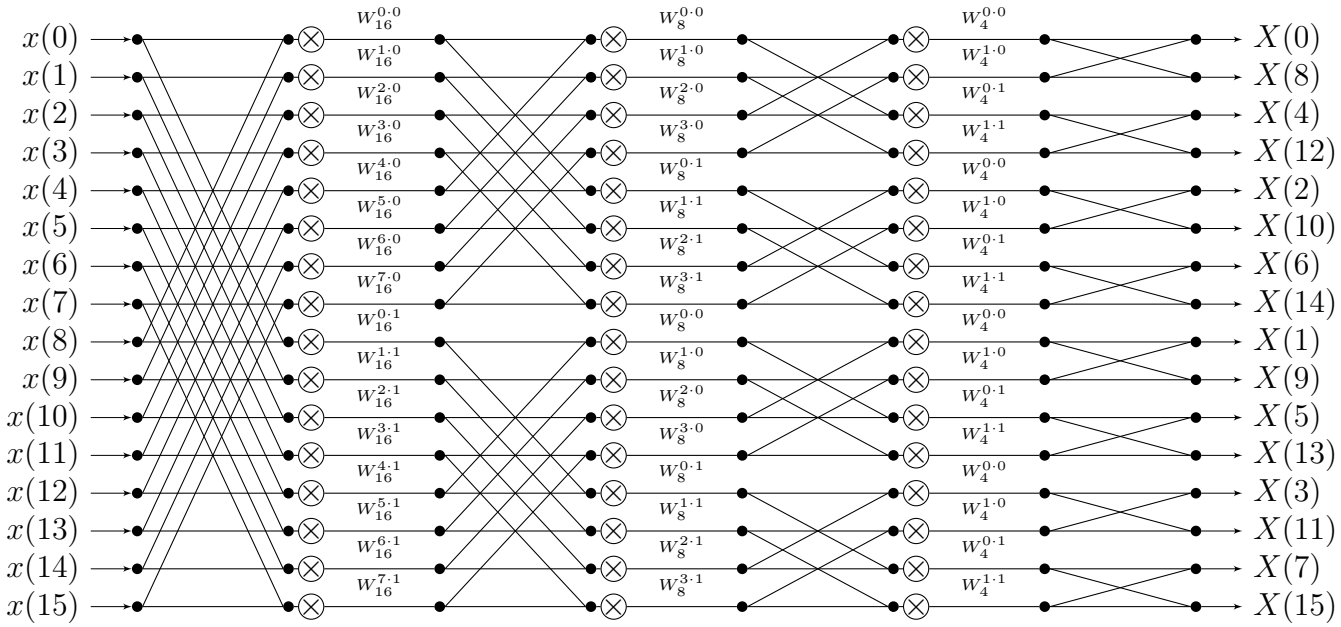
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



From [1]. cc-by-sa 2.0

Vectorizing FFT for faster AI Convolutions

Using the ARM SVE and the RISC-V ”V” extensions

Master’s thesis in Computer science and engineering

Victor El-Hajj

Anton Forsberg

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Vectorizing FFT for faster AI Convolutions

Using the ARM SVE and the RISC-V "V" extensions

Victor El-Hajj
Anton Forsberg

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Vectorizing FFT for faster AI Convolutions
Using the ARM SVE and the RISC-V "V" extensions
Victor El-Hajj, Anton Forsberg

Supervisor: Nikela Papadopoulou, Department of Computer Science and Engineering
Examiner: Miquel Pericas, Department of Computer Science and Engineering

Master's Thesis 2023
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 2023

Vectorizing FFT for faster AI Convolutions
Using the ARM SVE and the RISC-V "V" extensions
Victor El-Hajj, Anton Forsberg Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The Fast Fourier Transform (FFT) is a widely used algorithm in signal processing, communications and image processing. In this thesis we implemented and investigated FFT convolutions that leverage vector length agnostic programming for convolutional neural networks with the ARM Scalable Vector Extension (SVE) and RISC-V "V" vector extensions. Our research aimed to address the limitations of traditional vectorisation techniques that require unportable fixed length vector instructions. We analysed the performance of applying vector length agnostic instructions with different vector lengths and L2 cache sizes. Due to unforeseen issues with simulator programs, we were unable to run all benchmarks and investigate all vector lengths as originally planned. However, our results showed that code using both vector extensions benefit from being portable by showing increasing speedups with simulated vector lengths. At best, there was a speedup of two times compared to the baseline using a short vector length of 512 bits, though vectorised implementations of the General Matrix Multiply (GeMM) and Winograd convolutions outperformed our FFT implementation by three to four times on the SVE architecture and three to eleven times on the RISC-V "V" architecture on a network with small kernel sizes unfavourable to FFT. In conclusion, while the tools for simulating these architectures may be immature our investigation shows that the FFT convolution benefits from vector length agnostic programming.

Keywords: Computer science, engineering, project, thesis, HPC, FFT, CNN, vector length agnostic programming, RISC-V "V", ARM SVE.

Acknowledgements

We wish to especially thank our supervisors Nikela Papadopoulou and Sonia Rani Gupta for their help navigating the often cryptically documented tools and simulators we've used in this thesis.

Victor El-Hajj and Anton Forsberg, Gothenburg, 2023-06-22

Contents

List of Figures

List of Tables

Glossary

CNN Convolutional Neural Network. xi, 1, 3–5, 18, 19, 41

DFT Discrete Fourier Transform. 9, 10

EPI European Processor Initiative. 8, 20

FFT Fast Fourier Transform. v, xi–xv, 1–3, 5, 9–11, 13–15, 19, 21, 23–25, 27–31, 33–37, 39, 40

FFTN Fast Fourier Transform (FFT) of size N. 9–11

GeMM General Matrix Multiply. v, xi, xiii, 1, 3, 5, 8, 20, 23, 25, 26, 29, 31–33, 36, 37, 39–41

IFFT Inverse Fast Fourier Transform. xi, 1, 13, 15, 16, 19

ISA Instruction Set Architecture. 1, 3, 6–8, 19, 20

SIMD Single Instruction Multiple Data. 5, 7, 8

SVE Scalable Vector Extension. v, xi, xiii, 1, 3, 6–8, 14, 16, 18–21, 23–25, 27–29, 39–41

VLA Vector Length Agnostic. 1, 3, 5–8, 39

1

Introduction

Previous work that implemented the FFT on the ARM Scalable Vector Extension (SVE) Instruction Set Architecture (ISA) has not focused on convolution operations in Convolutional Neural Networks (CNNs) [2] [3]. In addition, FFT implemented with variable length vector instructions have shown promise to improve performance by leveraging additional parallelism. This could suggest that it would also be the case for FFT applied to convolution operations. The focus on CNN is important because additional optimisations to the FFT and Inverse Fast Fourier Transform (IFFT) algorithms are possible with CNN-specific knowledge [4].

There is a clear gap in previous research, which focuses on Winograd and General Matrix Multiply (GeMM), which is to implement and evaluate the performance of an FFT convolution algorithm that leverage variable length vector instructions and focus on convolution operations in CNNs, which investigated in this thesis. Our goal is to implement the FFT convolution targeted at CNNs using Vector Length Agnostic (VLA) ISAs. This would be compared with Winograd and GeMM as well as auto-vectorised and baseline FFT to investigate and compare the performance of different algorithms and vector extensions.

The following list describes the broader goals of this thesis:

- Implementing a FFT convolution for CNNs using the RISC-V V extension and the ARM SVE VLA ISA.
- Co-design, investigating performance impact of architecture parameters like L2 cache size and vector length.
- Evaluating performance impact of changes and comparing to baseline and other ISAs and implementations.

As the RISC-V vector extension is a relatively immature platform we did not have access to real hardware to run on and were limited to simulations. In addition, the co-design of the micro-architectural parameter choices and algorithm design will be limited to the maximum vector length and cache size.

The rest of this paper is organised as follows. Section 2 contains the background and theory required to understand our work. It brings up in more detail what convolutional neural networks, vector length agnostic programming, the NEON, ARM SVE and RISC-V "V" extensions are as well as establishing the link to previous

work. Section 3 will cover our methodology, mainly what changes were made to vectorise the FFT convolution and how the benchmarks were run the results evaluated. Section 4 will describe the results of our benchmarks and commentary on the data. Finally, section 5 covers discussion of the results and the conclusion we draw from it.

2

Theory

To understand the contents of this thesis it benefits to have a understanding of previous work, CNNs, VLA, architectures like NEON, ARM and SVE as well as algorithms for convolutions like Winograd, GeMM and FFT. These topics are introduced in this chapter.

2.1 Previous work

The concept of using VLA ISAs to evaluate the possible performance increases related to CNNs has been investigated in the past. In one paper in particular Gupta et al [5] investigates accelerating CNN inference on long vector architectures with co-design. This investigation was carried out by implementing and optimising the GeMM and winograd convolutions with ARM SVE and RISC-V "V" extension. This thesis aims to do a similar investigation for the FFT convolution.

2.2 Background

This thesis looks into vectorisation of CNNs. To understand this work a understanding of the algorithms and technologies involved is needed. This section provides a brief introduction of CNNs, Vector Length Agnostic programming, the ARM Scalable Vector Extension, the RISC-V "V" extension and some convolution algorithms such as the Fast Fourier Transform, Winograd and General Matrix Multiply.

2.2.1 Convolutional neural networks

A Convolutional Neural Network is a type of deep-learning architecture inspired by the vision of living creatures, and they are used in many areas like image classification, object detection, text recognition and natural language processing [6].

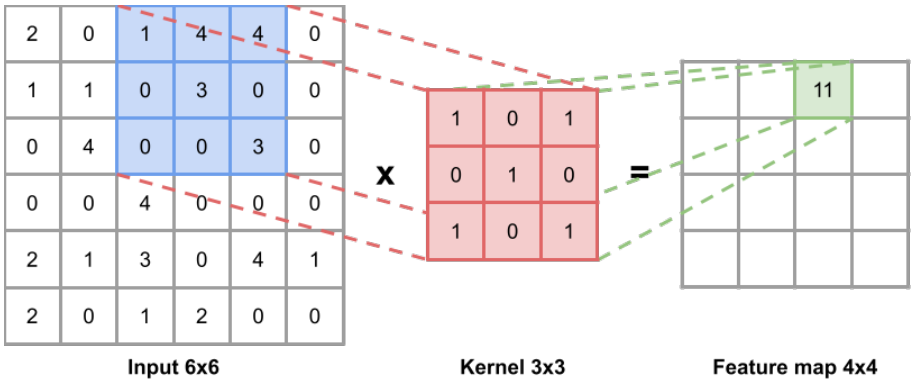


Figure 2.1: Illustration of a simplified convolution layer in a CNN.

In more detail, a CNN consists of several layers: convolution layers, pooling layers and fully-connected layers. A convolutional layer, as illustrated in figure 2.1, is used to compute a feature map where each neuron in the feature map is calculated from a region of neurons in the previous layer. This is known as the Local Receptive Field, which is illustrated in figure 2.1 as the blue region. This works by applying a set of kernels, also known as filters. Each kernel convolves across a region of the previous layer and applies an element-wise nonlinear activation function.

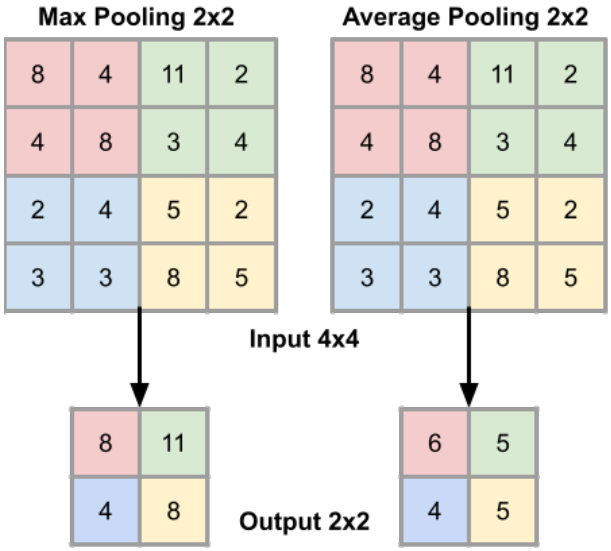


Figure 2.2: Illustration of 2x2 max and average pooling layer in a CNN.

A pooling layer produces a feature map of reduced resolution by down-sampling and is usually placed between convolutional layers to increase shift-invariance (invariance to small displacements). The two most common types of pooling layers can be seen in figure 2.2. Max pooling is where you take the maximum value and average pooling where you take the average. A pooling layer also helps to prevent over-fitting by reducing the amount of parameters.

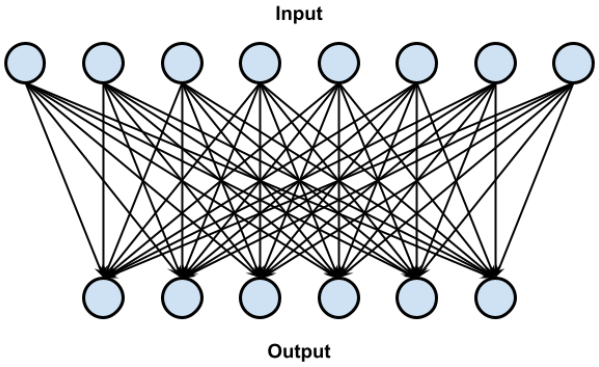


Figure 2.3: Illustration of a fully connected layer in a CNN.

Lastly, a fully-connect layer connects all neurons of a previous layer to all neurons of the new layer as can be seen in figure 2.3. The input can either be a flattened convolutional or pooling layer or a previous fully connected layer. These layers are for global semantic information and require a large amount of parameters.

There are many algorithms used for the convolutions in CNNs, with different concurrency characteristics[4]. One family of algorithms is GeMM. While a lot of recent research has been focused on optimising and improving CNNs based on the Winograd convolution, in many cases a FFT based method outperforms the Winograd method, especially in architectures with a high compute-to-memory ratio. But FFT based methods can be just as fast or faster when receiving a similar amount of optimisation work [7].

The popular deep neural network accelerators, such as Googles TPU and Nvidas tensor core are built on general matrix multiplication (GeMM). Because of this they utilise the algorithm implicit GeMM (also called implicit im2col), that converts a convolutional layer into a GeMM operation [8]. Implicit GeMM does not improve the concurrency characteristic, compared to a direct implementation of the convolution operation [4]. But implicit GeMM outperforms the direct implementation due to better processor utilisation.

Because of performance scaling and lower numerical accuracy on large filters, Winograd is strictly used for small kernels [4]. FFT convolutions on the other hand make large kernel sizes inexpensive, but can give poor performance on small kernels [9]. Thus, there does not exist a one-size-fits-all solution to the choice of convolution algorithm. Therefore, it is important to give research-focus to alternative algorithms such as FFT convolutions.

2.2.2 Vector length agnostic programming

Pohl et al [10] explains the origins of VLA programming, Single Instruction Multiple Data (SIMD) has been used to increase data level parallelism and has seen a trend of increasing vector lengths. One problem with vector extensions has been portability, where code has had to be written specifically for one vector architecture. VLA programming solves this by vectorising code independently of the vector length of

the target platform. This allows the same program to be able to run on different hardware with different vector length.

Two recently proposed VLA ISAs are the ARM SVE and the RISC-V "V" extension, which we cover in more detail in the following sections. Both of these ISA use vector registers which support many different lengths depending on the hardware used. The common feature of both instruction sets are instructions whose vector lengths are not fixed and which can be determined at execution to allow for vector length agnostic code.

```
void vector_add(float64_t *x, float64_t *y, float64_t *dst) {
        uint32_t i;
        svbool_t pq;
        svfloat64_t v_y, v_x, result;

        uint32_t n = 3;
        uint64_t numVals = svlen_f64(v_x);

        for (i=0; i<n; i+=numVals) {
                pq = svwhilelt_b64_s64(i, n);
                v_x = svld1_f64(pq, x+i);
                v_y = svld1_f64(pq y+i);
                result = svadd_x(pq, v_x, v_y);
                svst1_f64(pq, dst+i, result);
        }
}
```

Figure 2.4: Vector Lengh Agnostic code example using ARM SVE

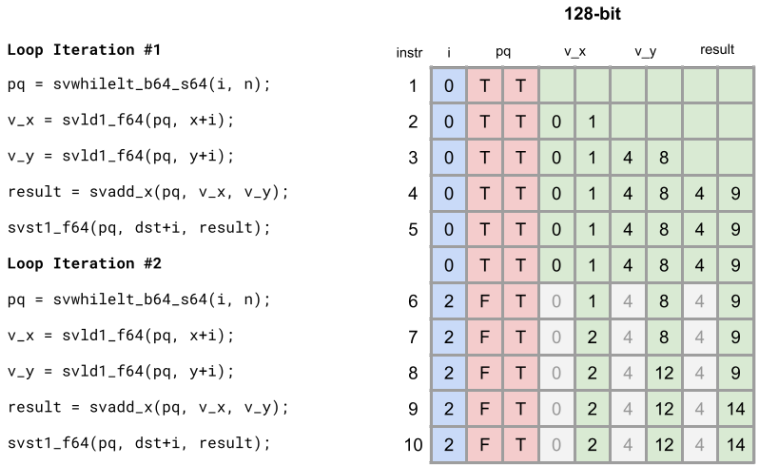


Figure 2.5: Illustration of how the code in figure 2.4 runs with 128 bit vector length.

Figure 2.4 contains an example of how VLA code can be written with ARM SVE. The simple program takes two arrays of three 64-bit elements, loads them as vectors,

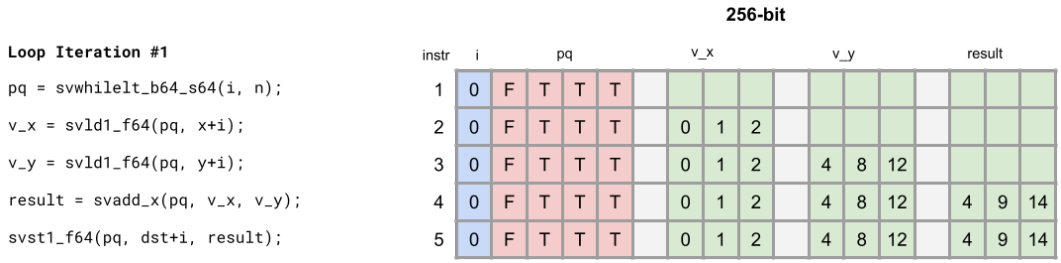


Figure 2.6: Illustration of how the code in figure 2.4 runs with 256 bit vector length.

adds them and stores the resulting vector in a destination array. Figure 2.5 and figure 2.6 demonstrates what happens when the same code is run on two different processors with different vector lengths. Both instances achieve the same result, though as the vector length in figure 2.5 is only large enough to store two 64 bit elements the loop requires two iterations to finish, while the larger vector length in figure 2.6 allows the program to complete in just one iteration. As can be seen in both figures the `pq` value holds a predicate which masks the instructions. This is used with `sv_whilelt` to calculate the required mask to avoid instructions going out of bounds.

2.2.3 NEON

ARM NEON is SIMD ISA with either 32 64-bits wide or 16 128-bits wide vector registers for a total length of 2048-bits [11]. Similar to ARM SVE and RISC-V ”V” instructions will perform the same operation on all lanes. These registers can be further divided into unsigned/signed 8, 16, 32 and 64-bit data values. In contrast to the ARM SVE and RISC-V ”V” extension the NEON extension only supports fixed length vector programming. It came before SVE and shares a similar instruction set.

The ARM NEON extension was developed for general purpose SIMD, examples mentioned are to accelerate multimedia codecs and power savings.

2.2.4 ARM SVE

The ARM SVE is an VLA ISA extension that allows vector registers with lengths between 128 and 2048 bits in 128 bit increments as described by Stephens et al [12]. Stephens et al explains that the architecture of SVE features 32 scalable vector registers with a implementation specified size that essentially work as extensions to the 32 128 bit SIMD registers of the NEON ISA. It also features 16 predicate registers used for instruction masking. Each 128 bit element can be further divided into 8, 16, 32 or 64 bit values.

The predicate registers provide further control over the SVE instructions. Each predicate has 8 bits per 64 bit vector element, allowing byte granularity. These provide control over things such as masking which elements are used and modified in an addition, or for avoiding reaching out of index in a loop.

ARM SVE is at its core an extension of the NEON (also known as Advanced SIMD) ISA, a fixed-length vector architecture, where the first 128 bits of each vector register functions as a NEON vector register. The instructions are also very similar. This allows us to make a comparison with NEON in later sections.

2.2.5 RISC-V "V" Extension

The RISC-V "V" extension is a VLA ISA extension that allows vector registers with more variable lengths. While the SVE extension allows vector registers with multiples of 128 bits up to 2048 bits, the RISC-V "V" extension allows any power of two up to 65536 bits, as described by the European Processor Initiative (EPI), the main force behind the "V" extension, on their website [13]. This takes the shape of 32 vector registers, which in addition are divided up into elements of at least 8 bits, up to the value of the vector length.

Another difference from the SVE extension is that instead of using predicate registers to mask instructions the RISC-V "V" provides instructions for requesting a specific vector length, which then returns a granted vector length. By providing this value to further instructions it can then limit what data is affected, in contrast to the SVE extension which processes the whole vector. The EPI further claims that for some versions of their Vector Processor Unit (VPU) a shortened vector length also shortens the latency of instructions, with the unused tail not requiring any computation [13].

A limiting factor the exploration of the RISC-V "V" extension is that, as the EPI claims, there is yet no hardware capable of running "V" extension instructions [13].

2.2.6 Winograd

Winograd is an algorithm for performing convolutions based on Winograd's minimal filtering algorithms [14]. The algorithm is fast for small filter and batch sizes. The advantages come from minimising the amount of multiplications used during matrix multiplication. As multiplications are a slow operation this can reduce algorithmic complexity by up to 2.25 times.

2.2.7 GeMM

GeMM stands for general matrix multiplication. It is part of BLAS (Basic Linear Algebra Subprograms), which is a standard for low level linear algebra routines. The subroutines are typically categorised into three "levels", with each level corresponding to the complexity of the routine. GeMM is part of "level 3" of BLAS, which details matrix-matrix operations [15]. This can then be used when convolutions are calculated trough matrix multiplications. This specific operation has received a lot of optimisation for decades and is a popular target for benchmarks.

2.2.8 FFT

FFT, the Fast Fourier Transform, is based on the convolution theorem which states that a convolution $f * g$ can be performed by $F^{-1}(F(f) \circ F(g))$, where F is the

fourier transform and F^{-1} is the inverse fourier transform. The Discrete Fourier Transform (DFT) is an adaptation of this for discrete values. DFT takes a sequence of complex values and transforms them into the Fourier domain, the equation for this can be seen below.

$$X(k) = DFT_k^N\{x\} = \sum_{N=0}^{N-1} x(n)W_N^{nk}, for k = 0, 1, ..., N-1 \tag{2.1}$$

where $W_n = e^{-i(2\pi/n)}$ is the complex root of unity (i.e. a complex number z that satisfies the equation $z^n = 1$) and are also called twiddle factors. The benefit of transforming the input and kernel to the Fourier domain is that the convolution can be done with a simple element-wise multiplication.

The FFT is a fast version for computing the DFT. It takes advantage of the symmetrical and periodic nature of DFT to divide it into smaller sub-problems, significantly reducing computational complexity from $O(n^2)$ to $O(n \log n)$.

The radix-2 CooleyTukey algorithm is one possible implementation of a FFT and in its simplest form, the algorithm operates on two complex numbers and is known as a butterfly. In a butterfly operation, the first output is the sum of the two inputs: $y_0 = x_0 + x_1$ and the second output is the difference of the two inputs: $y_1 = x_0 - x_1$. A signal-flow graph illustrating this operation can be found in Figure 2.7.

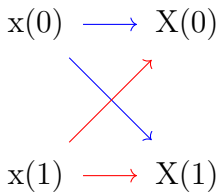


Figure 2.7: Signal-flow graph of the butterfly operation, see equation 2.2 for the mathematical interpretation.

$$\begin{aligned} y_0 &= x_0 + x_1 \\ y_1 &= x_0 - x_1 \end{aligned} \tag{2.2}$$

From now on, we will use the term FFT of size N (FFTN) to refer to an FFT with an input size of N. For larger input sizes, the radix-2 CooleyTukey algorithm starts by performing a butterfly operation with N inputs and then multiplying the result with twiddle factors. Next, it divides the input into two equal parts and apply FFTN/2 to each part separately. This process repeats itself until you reach the base case of FFT2. Figure 2.8 illustrates a signal-flow graph depicting this process for FFT16.

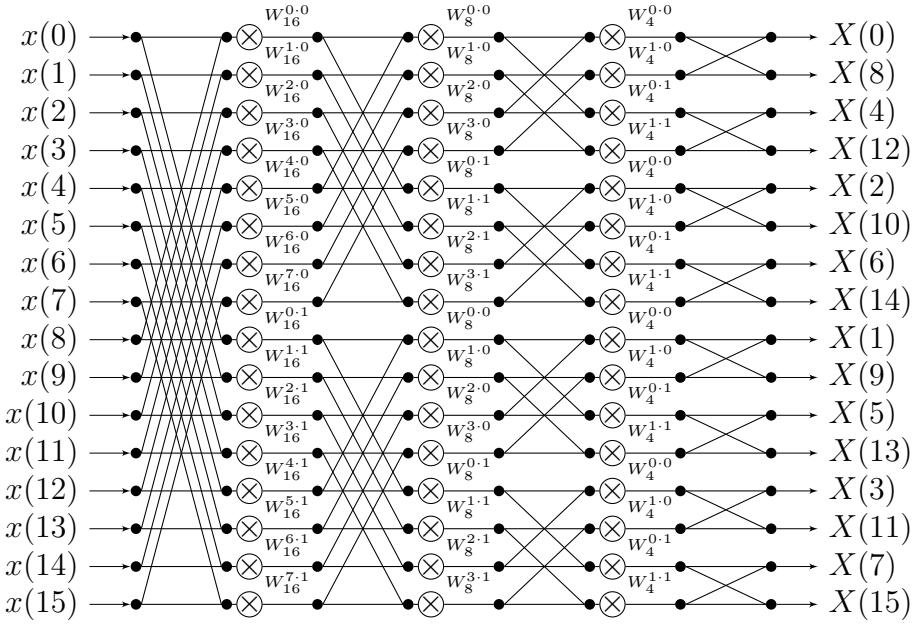


Figure 2.8: Signal-flow graph for radix-2 CooleyTukey with a input size of 16, $\bigotimes W$ is the multiplication with the twiddle factors. cc-by-sa 2.0 [1].

As explained, FFT operates on complex numbers, however the data in CNN are real values. As a result half the input to the FFT will just be zeros and thus using more memory then needed. However, it is possible to leverage the symmetry of DFT to transform two real value sequences simultaneously with a single complex FFT [16]. This is achieved by taking two sequences $x(n)$ and $x(n)$ and treating one sequence as the real values and the second as the imagery values, $z(n) = x(n) + iy(n)$. The result of the DFT will be:

$$DFT_K^N\{z\} = \{X_r(k) - Y_i(k)\} + i\{X_i(k) + Y_r(k)\} = Z[k] \tag{2.3}$$

From this we can extract the two real transforms using equation 2.4 and 2.5.

$$X[k] = \frac{Z[N-k]^* + Z[k]}{2}, k = 0, ...N/2 - 1 \tag{2.4}$$

$$Y[k] = \frac{Z[N-k]^* - Z[k]}{2}, k = 0, ...N/2 - 1 \tag{2.5}$$

Due to the symmetry of the DFT, it is sufficient to only calculate the first N/2 values. In eq 2.4 and 2.5 z^* is the complex conjugate:

$$\begin{aligned} z &= a + ib \\ z^* &= a - ib \end{aligned} \tag{2.6}$$

If however, we only have one real-valued sequence $x(n)$ we want to transform to the Fourier domain. Instead of using a complex FFTN, one could use a smaller

FFTN/2 to save on computational resources [16]. In this case the even elements of the sequence are interpreted as real values and the odd elements are interpreted as imaginary values. This new complex sequence $z(n) = x(2n) + ix(2n + 1)$ can then be used as input to the FFTN/2 function. The output can then be separated like in equation 2.4 and 2.5, resulting in the transform for the even $X_e(2n)$ and odd $X_o(2n + 1)$ elements. Finally, to combine them and get the transform of $x(n)$, we can use equation 2.7.

$$X[k] = X_e[k] + W_N^k X_o[k], k = 0...N - 1 \tag{2.7}$$

To perform a two-dimensional convolution of size NxN, one can apply a FFT to the rows and then to the columns, or vice versa.

3

Methods

This chapter firstly covers how the code was implemented. Secondly, it covers what tools were used to cross compile the code for the two architectures. Finally, it also covers how the cross-compiled code was simulated for the benchmarks and for what simulated architectures it was run.

3.1 Algorithmic Optimizations

Our convolution algorithm consist of four main parts:

- FFT on the kernel
- FFT on the input
- Tuple multiplication between the kernel/input transforms
- IFFT on the combined transform

For the kernel/input transform, pseudo code for a scalar implementation can be seen in figure 3.1. The first step in the implementation was a one-dimensional transform for each column. Since input data consist of real values, we can reduce the number of operations by utilising a real FFT function. In this case the real FFT was implemented by using a regular complex FFT transform of half the size, from which we can extract the real-valued transform. See section 2.2.8 for the full explanation on how to do this. Secondly the scalar algorithm in figure 3.1 performs a regular FFT on all the rows. Lastly in the `dual_real_fft` function we transform the first two rows to real valued sequences, using equation 2.4 and 2.5. The reason we needed to do this for the first two rows is because we actually just used a complex FFT on them in the real FFT function, thus we need to convert them to real-valued FFTs here instead. This was how NNPACK reference FFT convolution was implemented [17] and what we will use for basis in our vectorised version.

```
void FFT_scalar(float data[], float transform[]){
    int BLOCK_SIZE = 8; //or 16
    float block[BLOCK_SIZE][BLOCK_SIZE];

    for(int col = 0; col < column_count; col++)
        real_fft(block[0][col]);

    for(int row = 0; row < BLOCK_SIZE/2; row++)
        complex_fft(block[0][row]);

    dual_real_fft(block[0][1], block[0][2]);

    transform = block;
}
```

Figure 3.1: Scalar implementation of the FFT algorithm based on NNPACKs reference implementation [17].

In our vectorised SVE implementation for the complex FFT we parallelise over the individual transform and across the transforms done for each row. This function can be seen in figure 3.2 and would replace the second for-loop in figure 3.1. The for-loop will go over 32 float values, therefore this function can utilise 1024-bit long vector registers ($32 * 32$). The `butterfly` function in figure 3.2 simply follows equation 2.2 by calculating the new a vector to be $a = a + b$ and the new b vector to be $b = a - b$. `cmulc_twiddle` performs a complex multiplication between two vectors. For this purpose, ARM SVE offers built-in instructions for complex arithmetic. Finally, `shuffle` rearrange the vectors for the next stage. Figure 2.8 provides an example of how the pairing in the butterfly function changes between stages.

Since RISC-V "V" lacks instructions for complex arithmetic, our RISC-V implementation of function 3.2 divides a and b into real and imaginary values. Because we now uses four vectors ar, ai, br, bi we can only saturate 512-bit long vector registers, instead of 1024-bit.

If we go back to the scalar FFT implementation in 3.1, the first for-loop can be vectorised in a similar way as above, but it uses a FFT of half the size. The original for-loop is dependent on the value of *colum_count* that can be between 1 and 8. This can limit the saturation of the vector register if *colum_count* is small (128-1028 bits depending on *colum_count*). The transform of the kernel is especially effected by this, because here *colum_count* is equal to the size of the kernel, which is commonly only 1 or 3. Because the kernel transform needs to be done for each input channel, we can improve saturation by doing multiple channels at the same time. The amount of channels we did simultaneously was based on how many float values fit in a full vector register. We are then guaranteed to fill the vector registers even if *colum_count* is 1 (assuming there are enough input channels). But due to time constraints, this was only implemented for block size 16 on the SVE implementation. Similar improvements would also had be done for the FFT on the

input and IFFT, if we had more time.

These where the main steps in vectoring the FFT function for block size 8 and 16. The inverse FFT would mostly be the same, the big difference is that it is done in the revers order. Overview of the scalar IFFT implementation can be seen in figure 3.3.

```
void COMPLEX_FFT8x8__SVE(float BLOCK[]){

    int BLOCK_SIZE = 8;

    //pre-computed twiddle factors
    svfloat32_t twiddle_1, twiddle_2;

    for (int i = 0; i < BLOCK_SIZE * BLOCK_SIZE/2; i += VL){

        //load data from block
        a = svld1_gather_index(block);
        b = svld1_gather_index(block);

        // stage1
        butterfly(a, b);
        cmulc_twiddle(b, twiddle_1);
        shuffle(a, b);

        // stage2
        butterfly(a, b);
        cmulc_twiddle(b, twiddle_2);
        shuffle(a, b);

        // stage3
        butterfly(a, b);

        //store a and b in block
        svst1_scatter_index(block, b):
        svst1_scatter_index(block, a);
   }
}
```

Figure 3.2: Pseudo code for our vectorized implementation of the FFT algorithm, replaces the second for loop in the scalar implementations (figure 3.1).

```
void iFFT_scalar(float transform[], float output[]){
    int BLOCK_SIZE = 8; //or 16
    float block[BLOCK_SIZE][BLOCK_SIZE];

    dual_real_ifft(block[0][1], block[0][2]);

    for(int row = 0; row < BLOCK_SIZE/2; row++)
        complex_ifft(block[0][row]);

    for(int col = 0; col < column_count; col++)
        real_ifft(block[0][col]);

    output = block;
}
```

Figure 3.3: Scalar implementation of the IFFT algorithm based on NNPACKs reference implementation [17].

The tuple multiplication was vectorised over the output channels. Pseudo-code for this can be seen in figure 3.4. Inside the loop we need to perform multiply and add instructions between the input and kernel transforms. Doing just this will only saturate up to 1024-bits of vector length for block size 8 and 4096-bits for block size 16. As ARM can handle up to 2048 bits and RISCV 16384 bits (or potentially even more) we need to improve the data parallelism. Therefore when the vector length is a multiple of the maximum vector length ($VL_{max} = BLOCK\_SIZE^2/2 * 32$) we can do multiple iterations of the for loop at a time (in figure 3.4 each iteration increases with $VL * split$). To get back to the maximum allowed vector length after the for-loop we add together the two halves of the vector. This works for the SVE implementation since only when the block size is 8 we can get a multiple of 2 when VL is 2048. Our RISCV-V "V" implementation was the same as the SVE implementation presented in 3.4, except that it can handle larger multiples than 2 in the final step to get back to the maximum allowed vector length.

If we now look at an overview of the whole convolution in figure 3.5, we can see longer vector lengths also reduce the amount of tuple multiplication we needed to do by reducing the total amount of iterations for the innermost for-loop ($iterations = tile\_elements/min(tile\_elements, 2 * VL)$).

Our implementation also supports multi core parallelism, the iFFT are parallelized over the output channels and the rest of the functions are parallelized over the input channels.

```
1
2  void tuple_multiplication(input_tranform, kernel_trasnform,
3                            output_channels, output);
4
5  // BLOCK_SIZE = 8/16 => maxLength = 1024/4096
6  maxLength = BLOCK_SIZE * BLOCK_SIZE / 2;
7  int split =  max(1, floor(VL / maxLength));
8  for(int i = 0; i < output_channels * VL; i +=VL * split){
9     //load data into the vectors
10    a0r = svld1(pg,input_tranform);
11    b0r = svld1(pg,kernel_trasnform);
12    b0i = svld1(pg,kernel_trasnform);
13
14     //svmla(pg, a,b,c) = a+b*c
15     acc00r = svmla(pg,acc00r, a0r, b0r);
16     acc00r = svmla(pg,acc00r, a0i, b0i );
17
18      //same for acc01r, acc10r, acc11r,
19      //          acc00i,acc01i,acc10i,acc11i
20     ...
21 }
22
23 if(split>1){
24     add_low_half_and_high_half(acc00r);
25     add_low_half_and_high_half(acc00i);
26     ...
27 }
28 //store output
29 svst1(pg, output, acc00r);
30 svst1(pg, output, acc00i);
31 ...
32 }
```

Figure 3.4: Pseudo code of our the tuple multiplication function for ARM SVE

```
void convolution(float input[], float kernel[], float output[]){

    int tile_size = 8; // or 16
    int tile_elements = tile_size * tile_size;

    int tuple_elements = min(tile_elements, 2 * VL);
    int tuple_count = tile_elements/tuple_elements;

    float output_transform [outpu_channel]= {0};

    for(int c = 0; c < channels; c++){

        float kernel_trasnform[] = FFT_2D(kernel[c]);

        float input_tranform[] = FFT_2D(input[c]);

        for (int t = 0; t < tuple_count; t++) {

           output_transform =
            tuple_multiplication(input_tranform, kernel_trasnform);
        }
    }

    for(int oc = 0; oc < outpu_channel; oc++)
        output[out] = iFFT(output_transform[out]);
}
```

Figure 3.5: Structure of our convolution algorithm

Table 3.1: Maximum Vector utilisation for each function (actual utilisation can be lower depending on the kernel size, number of input/output channels and the amount of input data).

function	Block size	SVE	RISCV
FFT/IFFT	8	1024-bit	512-bit
Tuple multiplication	8	2048-bit	16384-bit
FFT/IFFT	16	2048-bit	1024-bit
Tuple multiplication	16	2048-bit	16384-bit

3.2 Experimental setup

We chose to focus on Darknet, an open source neural network framework written in C and cuda [18]. This provided an easy way to run and benchmark several popular CNNs, like YOLOv3, which is used for real time object detection [19].

Another benefit of using Darknet as a base was that it optionally depends on NNPACK, an acceleration package for neural network computations which provides backends for various processor architectures, like ARM NEON and X86_64[20]. This allowed us to implement our work as additional backends to NNPACK which allowed us to benchmark and test existing CNNs with Darknet.

Our aim was to make the FFT, IFFT as well as matrix multiplications, which make up the FFT convolution, data parallel. The following sections describe for each ISAs how we modified the code to make it data-parallel and vector length agnostic, as well as how NNPACK was compiled for the various benchmarks.

The different setups describe how we compiled NNPACK. To avoid factoring the time spent in Darknet for our benchmarks where we are solely interested in the performance of the convolutional operations we first measure the time each benchmark took within Darknet without any prediction and then subtracted that from the total time. This yielded the time spent on the convolutions.

To simulate the performance of these different programs and architectures we made use of the gem5 simulator, which is a tool for computer architecture research that allows cycle-level simulations for, amongst others, the SVE ISA[21].

In addition, to be able to confirm that our new NNPACK backends are functionally correct we made use of the built-in NNPACK tests which provided a convenient method of testing convolutions with random input data and seeing if the outputs are as expected. To run these tests quickly we made use of the QEMU emulator, which allows for faster testing and iteration than gem5 with the drawback of not being deterministic or simulating a system in enough detail to run our benchmarks [22].

3.2.1 ARM SVE

To be able to cross-compile to ARM SVE programs we made use of the `aarch64-linux-gnu` toolchain. We targeted the ARMv8 architecture to be able to work with SVE instructions. Finally we always used the `-Ofast` optimization level for a fair comparison. This can also be used for the auto-vectorised benchmark, as `-msve-vector-bits` defaults to ”scalable”, which generates vector length agnostic code.

```
$ aarch64-linux-gnu-gcc -march=arm8-a+sve -Ofast
```

There was one more case to consider. For the baseline benchmarks without auto-vectorization we appended `-fno-tree-vectorize` and `-fno-tree-slp-vectorize` to disable auto-vectorization.

```
$ aarch64-linux-gnu-gcc -march=arm8-a -Ofast \
-fno-tree-vectorize \
-fno-tree-slp-vectorize
```

For using gem5 to simulate ARM we used the default ARM build combined with the Syscall Emulation mode gem5 configuration. Additionally, we combined this with the `MinorCPU` CPU model. With gem5 we then ran simulated hardware with

different L1 and L2 cache sizes as well as vector lengths for our benchmarks to evaluate the performance impact of different hardware configurations.

3.2.2 RISC-V "V" Extension

To be able to cross-compile to the RISC-V "V" extension we made use of the `llvm-EPI-development-toolchain-cross` cross-compiler provided by the EPI [13]. We specified that we target the riscv64 architecture and use the `-Ofast` flag for fair comparison. The `-mepi` flag allows the compiler support for the EPI vector instructions and intrinsics. This setup was also used for the auto-vectorized benchmark as the `-mrvv-vector-bits=<arg>` flag defaults to "scalable", which generates vector length agnostic code.

```
$ clang --target=riscv64-unknown-linux-gnu -mepi -Ofast
```

There was one more case to consider, the baseline comparison without auto-vectorization. To disable auto-vectorization we appended the `-fno-vectorize` and `-fno-slp-vectorize` flags to the previous command.

```
$ clang --target=riscv64-unknown-linux-gnu -mepi -Ofast \
-fno-vectorize \
-fno-slp-vectorize
```

Finally, to simulate these RISC-V "V" extension programs we used the "plct-gem5" fork of gem5 simulator with added RISC-V "V" support as the official version of gem5 does not yet provide support for simulating RISC-V architectures with the "V" extension. [23]

3.3 Benchmarks

To evaluate the performance of our vector length agnostic implementations we ran a series of benchmarks across different combinations of L2 cache size and vector lengths, measuring the performance of running Darknet with the YOLOv3-tiny network, which is a smaller version of the YOLOv3 network mentioned earlier. In addition, we compared the results with a baseline implementation with and without auto-vectorization, as well as the existing NNPACK backend using the ARM NEON ISA, which is a fixed length vector extension that preceded SVE.

To extend the comparison to other algorithms we also run benchmarks for Gupta et al's vectorised Winograd and GeMM convolutions [5].

Table 3.2: L2 Cache sizes for benchmarks @ gem5.

L2 Cache Size
1MB
8MB
64MB
256MB

For all configurations we investigate the impact of L2 cache size on performance as shown in table 3.2.

3.3.1 ARM SVE

For our ARM benchmarks we used the same CPU model, clock rate and L1 cache size as can be seen in table 3.3. Finally, for the SVE configuration for each L2 cache size we investigate the impact of different vector lengths as shown in table 3.4. The initial goal of this thesis was to investigate longer l possible vector lengths in the ARM SVE extension, including 1024 and 2048-bits. However, in our investigations we found that the gem5 simulator did not work with these vector lengths and we were limited 256 and 512 bits. In addition, the implementation for the 16x16 FFT convolution requires a minimum of 512 bits, only allowing one possible vector length.

Additional parameters passed to gem5 were: setting the cacheline size to 64, the L1 data and instruction cache associativity to 4 and the L2 associativity to 8. The intention was to see not only how different hardware vector lengths affect the performance of vector length agnostic code, but to also see how it interacts with different cache sizes.

Table 3.3: Hardware setups for ARM benchmarks @ gem5.

	Baseline	NEON	SVE
Processor	MinorCPU	MinorCPU	MinorCPU
Clock Rate	2GHz	2GHz	2GHz
L1 Data Cache Size	64kB	64kB	64kB
L1 Instruction Cache Size	16kB	16kB	16kB
L2 Cache Size	Variable	Variable	Variable
Vector Length	Not applicable	128-bits	Variable

Table 3.4: Vector Lengts for SVE configuration for ARM benchmarks @ gem5.

SVE Vector Length
256
512

3.3.2 RISC-V ”V” extension

For our RISC-V benchmarks we used the same CPU model, clock rate and L1 Cache Size as can be seen in table 3.5. For all configurations we investigated the impact of L2 cache size on performance as shown in table 3.6. Finally, for the ”V” extension configuration for each L2 cache size we investigated the impact of different vector lengths as shown in table 3.7. In contrast to SVE the ”V” extension allows for larger vector lengths and we did not run into simulator issues, permitting a fuller investigation of the impact of vector lengths.

Additional parameters passed to gem5 were: setting the cacheline size to 64, the L1 data and instruction cache associativity to 4 and the L2 associativity to 8.

	Baseline	”V” extension
Processor	MinorCPU	MinorCPU
Clock Rate	2GHz	2GHz
L1 Data Cache Size	64kB	64kB
L1 Instruction Cache Size	16kB	16kB
L2 Cache Size	Variable	Variable
Vector Length	Not applicable	Variable

Table 3.5: Hardware setups for RISC-V ”V” benchmarks @ gem5.

Table 3.6: L2 Cache sizes for RISC-V benchmarks @ gem5.

L2 Cache Size
1MB
8MB
64MB
256MB

Table 3.7: Vector Lengts for ”V” extension configuration for RISC-V benchmarks @ gem5.

RISC-V ”V” Vector Length
256
512
1024
2048
4096
8192

4

Results

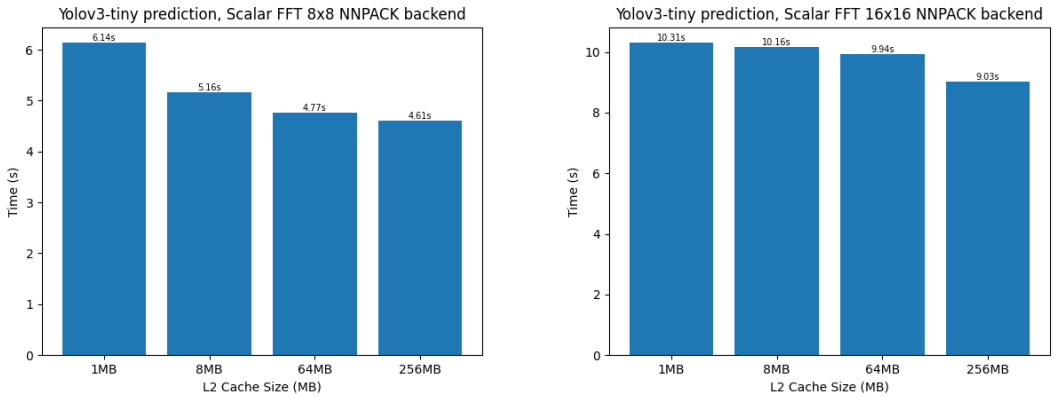
This chapter describes the results of the benchmarks described in previous chapters. The goal was to investigate the impact of vector length and cache on the performance of network prediction, as well as to compare the FFT algorithm with the GeMM and Winograd algorithms. The experimental setup for the following benchmarks is described in more detail in section 3.2.

4.1 ARM

The first half of this chapter describes the results for the ARM architectures. In addition to SVE, we run NEON benchmarks due to the similarity with the SVE architecture combined with the fact that it is already implemented as a NNPACK backend. The final benchmark of this section is our FFT implementation, with comparison tables to the other benchmarks.

4.1.1 Baseline

The ”baseline” configuration is the out-of-the-box FFT algorithm provided by NNPACK with all auto-vectorisation by the compiler disabled and with a simulated hardware architecture without the vector extension.



(a) ARM Scalar FFT 8x8

(b) ARM Scalar FFT 16x16

Figure 4.1: ARM Scalar FFT benchmarks with different L2 cache sizes.

The results without any kind of vector programming were as expected. They were comparably slow, with the $8x8$ FFT finishing the network prediction after a simulated 6.14 seconds with the smallest cache and 4.61 seconds with the largest cache as can be seen in figure 4.1a. In comparison, the $16x16$ FFT version as seen in figure 4.1b took about twice as long for Yolov3-tiny network prediction, 10.31 seconds for 1MB cache and 9.03 seconds for the 256MB cache. The slower speed for the $16x16$ variant was generally expected as the large kernel size was at a disadvantage due to Yolov3-tiny being a small network with small kernel sizes.

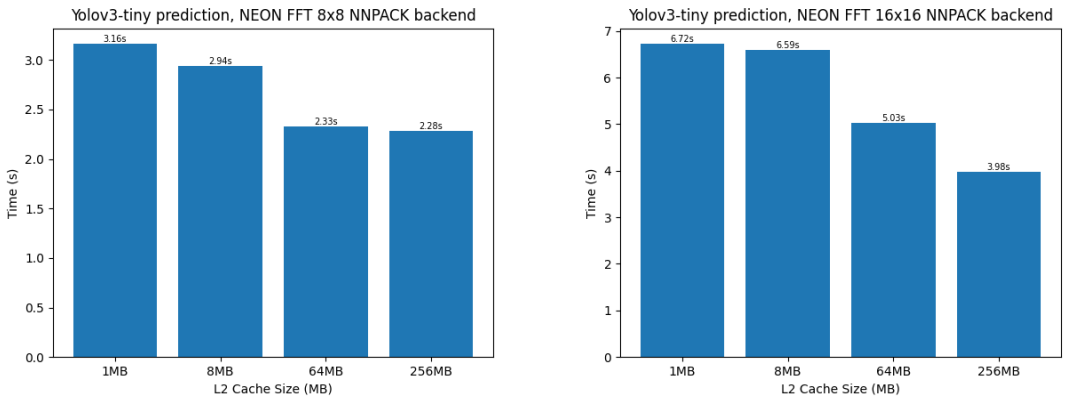
Table 4.1: Speedup by L2 cache size for baseline FFT.

Cache size	1MB	8MB	64MB	256MB
8x8	1.0	1.19	1.29	1.33
16x16	1.0	1.01	1.04	1.14

The speedup for baseline FFT increased with L2 cache size as can be seen in table 4.1. The $8x8$ variant has a speedup of up to 1.33 with the largest cache size, while the $16x16$ variant only achieved a speedup of 1.14 with the largest cache size.

4.1.2 NEON

The NEON FFT implementations are provided out-of-the-box by NNPACK.



(a) ARM NEON 8x8

(b) ARM SVE 16x16

Figure 4.2: ARM NEON FFT benchmarks with different L2 cache sizes.

For these benchmarks the slowest prediction time for the NEON $8x8$ setup was 3.16 seconds with the 1 MB cache and the fastest 2.28 seconds with the 256 MB cache. The $16x16$ variant took 6.72 seconds for the smallest 1 MB cache and 3.98 seconds for the larger 256 MB cache.

It can be seen in table 4.2 that both variants showed a speedup with increased cache sizes. The $8x8$ variant achieved a maximum speedup of 1.39 with the 256 MB L2 cache while the $16x16$ variant achieved a maximum speedup of 1.68 with the 256

Table 4.2: Speedup by L2 cache size for NEON FFT.

Cache size	1MB	8MB	64MB	256MB
8x8	1.0	1.07	1.36	1.39
16x16	1.0	1.02	1.34	1.68

MB L2 cache. Notably, both variants experience a significant jump in speedup after the increase in L2 cache size from 8 MB to 64 MB.

Table 4.3: NEON FFT speedup compared to baseline with same cache size.

Cache size	1MB	8MB	64MB	256MB
8x8	1.94	1.76	2.05	2.02
16x16	1.53	1.54	1.98	2.27

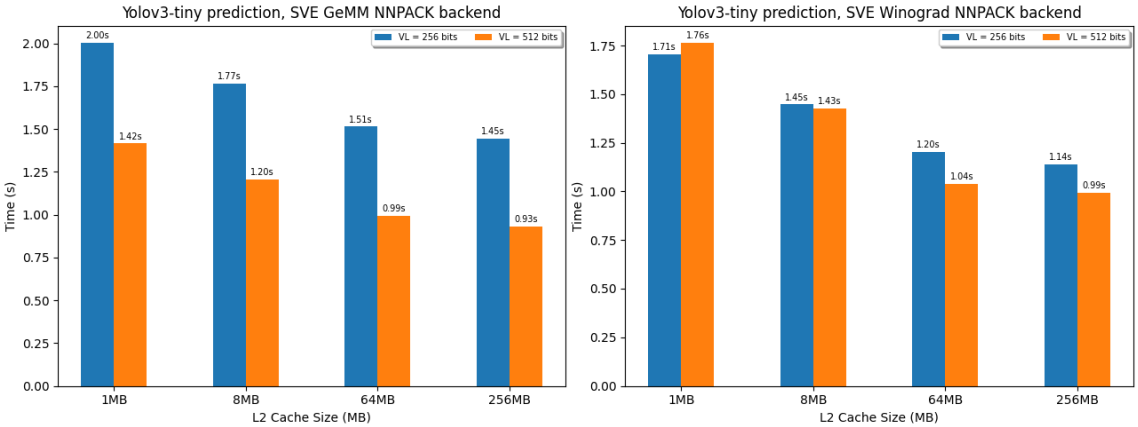
Comparing the performance of the short vector NEON architecture against the baseline version we can observe the speedup in table 4.3. Both the $8x8$ and $16x16$ variants saw a greater than two speedup at the largest cache size.

4.1.3 SVE

For the ARM SVE architecture we ran benchmarks for both our vectorised implementation of FFT as well as the scalar implementation provided by NNPACK that has been automatically vectorised by the compiler as mentioned in section 3.2.1. In addition, we compared these results with Gupta et al's vectorised Winograd and GeMM implementations [5].

4.1.3.1 Winograd and GeMM

In this section we show the results from running the Winograd and GeMM algorithms in order to compare with the various FFT results.



(a) ARM SVE GeMM (b) ARM SVE Winograd

Figure 4.3: ARM SVE Winograd and GeMM benchmarks with different L2 cache sizes and vector lengths.

As can be seen in tables 4.3a and 4.3b these algorithms were fast. The slowest result for GeMM was 2 seconds with 1 MB L2 cache and 256 bits vector length, while the fastest was 0.93 seconds with 256 MB L2 cache and 512 bits vector length. The slowest Winograd result was 1.76 seconds with 1 MB L2 cache and 512 bits vector length, interestingly this was the only result where an increase in vector length saw a slower prediction. Finally, the fastest Winograd prediction was 0.99 seconds with 512 bits of vector length and 256 MB of L2 cache.

Table 4.4: Speedup by L2 cache size for Winograd and GeMM.

Cache size	1MB	8MB	64MB	256MB
GeMM VL=256	1.0	1.13	1.32	1.38
GeMM VL=512	1.0	1.18	1.43	1.53
Winograd VL=256	1.0	1.18	1.43	1.5
Winograd VL=512	1.0	1.23	1.69	1.78

Table 4.4 shows the speedup per cache size for GeMM and Winograd for each vector length. It can be seen that the fastest speedup for both algorithms was with the larger 512 bit vector length and the largest 256 MB L2 cache, with a speedup of 1.53 for GeMM and a speedup of 1.78 for Winograd.

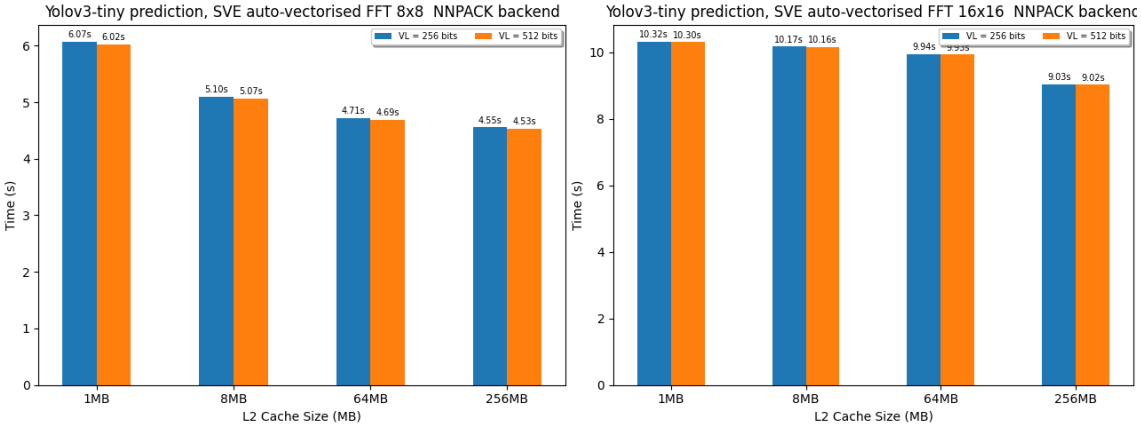
Table 4.5: Winograd and GeMM speedup compared to 8x8 baseline with same cache size.

Cache size	1MB	8MB	64MB	256MB
GeMM VL=256	3.07	2.92	3.16	3.18
GeMM VL=512	4.32	4.30	4.81	4.96
Winograd VL=256	3.59	3.56	3.98	4.04
Winograd VL=512	3.49	3.60	4.59	4.66

The resulting speedups compared to the $8x8$ benchmark can be seen in table 4.5. The GeMM algorithm provides the slowest speedup of 3.07 with a vector length of 256 bits and a L2 cache size of 1 MB. The largest speedup of 4.96 was seen with a vector length of 512 bits and a L2 cache size of 256 MB. Winograd on the other hand has the smallest speedup of 3.49 at 256 bits vector length and 1 MB L2 cache. The largest Winograd speedup was 4.66 with 512 bits of vector length and 256 MB of L2 cache.

4.1.3.2 Auto-vectorised baseline

This section shows the results of the benchmarks for the baseline implementation auto-vectorised by the compiler.



(a) ARM SVE Auto-vectorised FFT 8x8 (b) ARM SVE Auto-vectorised FFT 16x16

Figure 4.4: ARM SVE Auto-vectorised FFT benchmarks with different L2 cache sizes and vector lengths.

The two auto-vectorised benchmarks saw an insignificant speedup with increasing vector lengths. The auto-vectorised results are also within ±1% of the baseline so comparison tables are omitted.

4.1.3.3 Our implementation

This section has the results of the benchmarks for our implementation of the FFT algorithm using SVE instructions, as well as comparisons to all the previous ARM benchmarks.

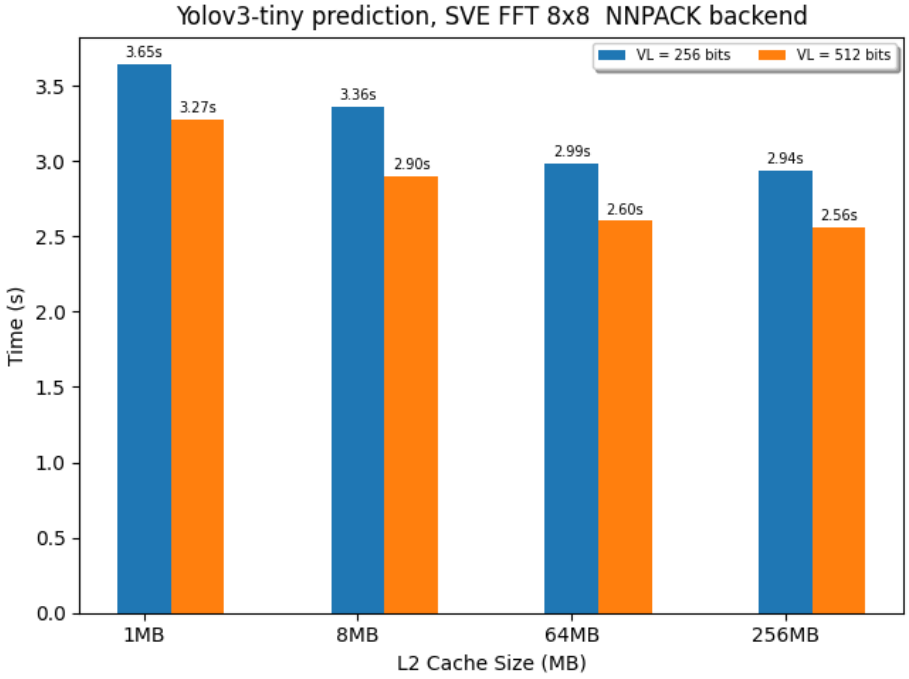


Figure 4.5: FFT 8x8 benchmarks with different L2 cache sizes and vector lengths.

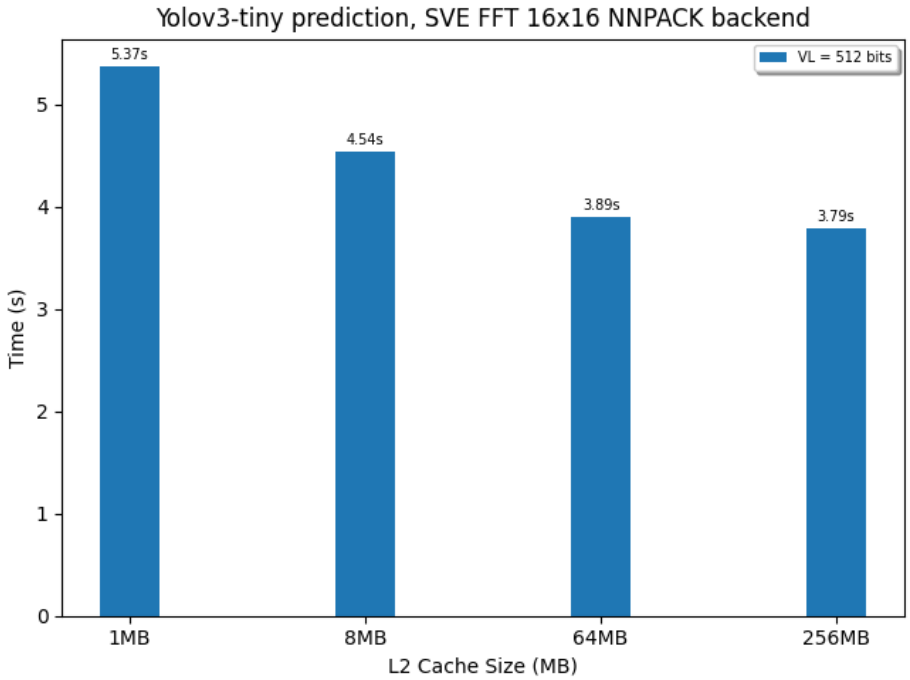


Figure 4.6: FFT 16x16 benchmarks with different L2 cache sizes.

The results for FFT 8x8 with SVE instructions can be seen in figure 4.5. The slowest time with a vector length of 256 bits and a L2 cache size of 1 MB was 3.65 seconds, while the fastest time with vector length 512 and L2 cache size 256 MB was 2.56s. In comparison, the 16x16 variant which was only able to run with 512 bits had a slowest time of 5.37 seconds with 1 MB of L2 cache and 3.79 seconds with 256 MB of L2 cache.

Table 4.6: Speedup by L2 cache size for SVE FFT.

Cache size	1MB	8MB	64MB	256MB
Speedup 8x8, VL=256 bits	1.0	1.09	1.22	1.24
Speedup 8x8, VL=512 bits	1.0	1.13	1.26	1.28
Speedup 16x16, VL=512 bits	1.0	1.18	1.38	1.42

The impacts of increasing cache size can be seen in table 4.6. Each variant was faster with increasing cache size, though the impact was not linear. As can be seen in the table the 8x8 version has a larger speedup as cache size increases for the larger vector length, similarly the 16x16 variant has the largest speedup for each cache size increase.

Table 4.7: SVE FFT speedup compared to baseline with same cache size. 8x8 compared to 8x8 and 16x16 compared to 16x16.

Cache size	1MB	8MB	64MB	256MB
Speedup 8x8, VL=256 bits	1.68	1.53	1.60	1.57
Speedup 8x8, VL=512 bits	1.88	1.78	1.83	1.80
Speedup 16x16, VL=512 bits	1.92	2.24	2.56	2.40

Table 4.8: SVE FFT speedup compared to NEON with same cache size. 8x8 compared to 8x8 and 16x16 compared to 16x16.

Cache size	1MB	8MB	64MB	256MB
Speedup 8x8, VL=256 bits	0.87	0.88	0.78	0.78
Speedup 8x8, VL=512 bits	0.96	1.01	0.89	0.89
Speedup 16x16, VL=512 bits	1.25	1.45	1.29	1.05

Tables 4.7 and 4.8 shows speedup of the SVE FFT compared to the baseline and NEON versions. It can be seen that both variants of SVE FFT were faster than the baseline version, with the $16x16$ variant having the largest speedup. Compared to the NEON backend the $8x8$ SVE variant was actually slower with both 256 and 512 bits of vector length compared to the 128 bits of NEON. If the trend observed would continue there might be a speedup at larger vector lenghts, though we were unable to run those as described earlier. Finally, the $16x16$ SVE variant did have a speedup compared to NEON, with the largest speedup being 1.45 at 8MB L2 cache size and the smallest speedup being 1.05 at 256 MB L2 cache.

Compared to the auto-vectorised implementation we found that the network prediction time was within $\pm 1\%$ of the baseline version, therefore their results were not compared separately with our SVE implementation.

Table 4.9: SVE FFT speedup compared to GeMM with same cache size and vector length.

Cache size	1MB	8MB	64MB	256MB
Speedup 8x8, VL=256 bits	0.55	0.53	0.51	0.49
Speedup 8x8, VL=512 bits	0.43	0.41	0.38	0.36
Speedup 16x16, VL=512 bits	0.26	0.26	0.25	0.25

Table 4.10: SVE FFT speedup compared to Winograd with same cache size and vector length.

Cache size	1MB	8MB	64MB	256MB
Speedup 8x8, VL=256 bits	0.47	0.43	0.40	0.39
Speedup 8x8, VL=512 bits	0.54	0.49	0.40	0.39
Speedup 16x16, VL=512 bits	0.33	0.31	0.27	0.26

The final comparisons of our SVE results was with the results of the Winograd and GeMM benchmarks that can be seen in tables 4.9 and 4.10. The results showed that for the Yolov3-tiny network our FFT $8x8$ implementation with a vector length of 256 bits takes between 1.82 and 2.04 times as long for network prediction compared with the GeMM implementation with the same cache sizes and vector length. Likewise, the FFT $8x8$ implementation with a long vector length of 512 bits took between 2.33 and 2.78 times as long for network prediction. Finally, the FFT $16x16$ implementation with a vector length of 512 bits took between 3.85 and 4 times as long for network prediction.

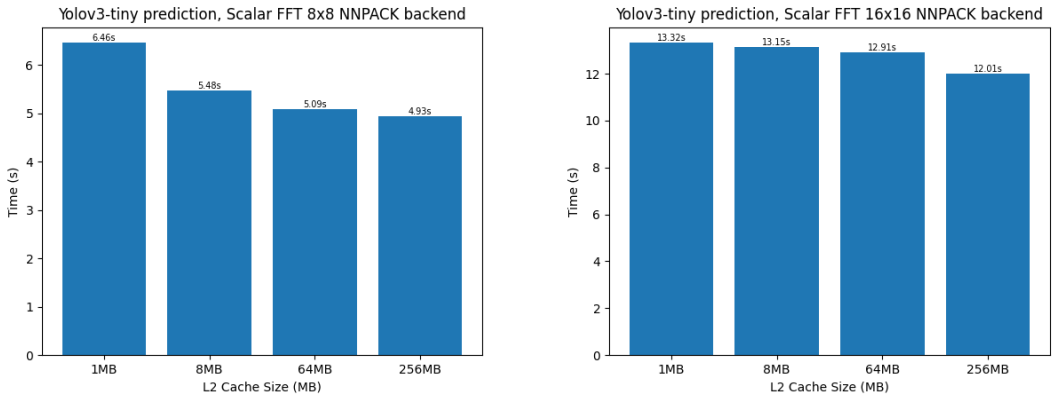
Compared with Winograd the results are: FFT 8x8 with a vector length of 256 bits took between 2.13 and 2.56 times along for network prediction, FFT 8x8 with a vector length of 512 bits took between 1.85 and 2.56 times as long for network prediction and FFT 16x16 with 512 bits vector length takes between 3 and 3.85 times as long for network prediction.

4.2 RISC-V

The second half of this chapter describes the benchmark results for the RISC-V architectures and algorithms.

4.2.1 Baseline

The ”baseline” configuration was provided by NNPACK and was run with all auto-vectorisation by the compiler disabled and with a simulated hardware architecture without the vector extension.



(a) RISC-V FFT 8x8

(b) RISC-V FFT 16x16

Figure 4.7: RISC-V FFT benchmarks with different L2 cache sizes.

The results without any kind of vector programming were comperatively slow as expected, the 8x8 FFT finishes the network prediction after a simulated 6.46 seconds with the smallest cache and 4.93 seconds with the largest cache as can be seen in figure 4.7a. In comparison, the 16x16 FFT version as seen in figure 4.7b took over twice as long for Yolov3-tiny network prediction, 13.32 seconds for 1MB cache and 12.01 seconds for the 256MB cache. The slower speed for the 16x16 variant was generally expected as the large kernel size was at a disadvantage due to Yolov3-tiny being a small network with small kernel sizes.

Table 4.11: Speedup by L2 cache size for baseline FFT.

Cache size	1MB	8MB	64MB	256MB
8x8	1.0	1.18	1.27	1.31
16x16	1.0	1.01	1.03	1.11

Looking at the speedup as the L2 cache size increases as seen in table 4.11 it can be seen that the $8x8$ variant had a speedup of up to 1.31 with the largest cache size, while the $16x16$ variant only achieved a speedup of 1.11 with the largest cache size.

4.2.2 "V" Extension

This section has the benchmarks run using the RISC-V "V" extension. In addition to our vectorised implementation of the FFT algorithm we also ran benchmarks for the auto-vectorised version of the scalar FFT implementation as well as Gupta et al's vectorised Winograd and GeMM convolutions [5].

4.2.2.1 Winograd and GeMM

In this section we show the results from running the Winograd and GeMM algorithms with the RISC-V "V" extension in order to compare with the FFT results.

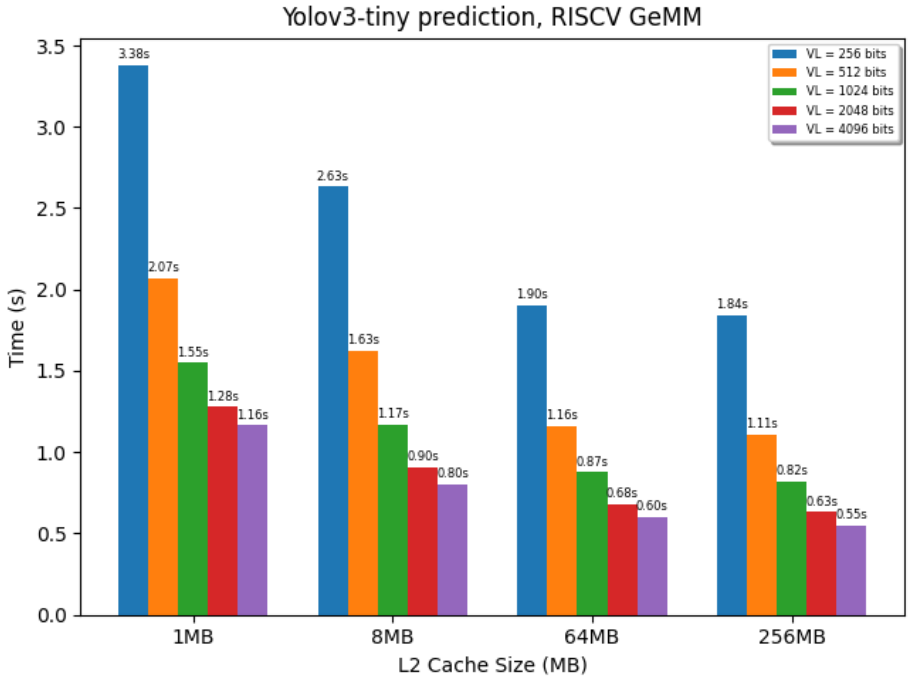


Figure 4.8: RISC-V GeMM benchmarks with different L2 cache sizes and vector lengths.

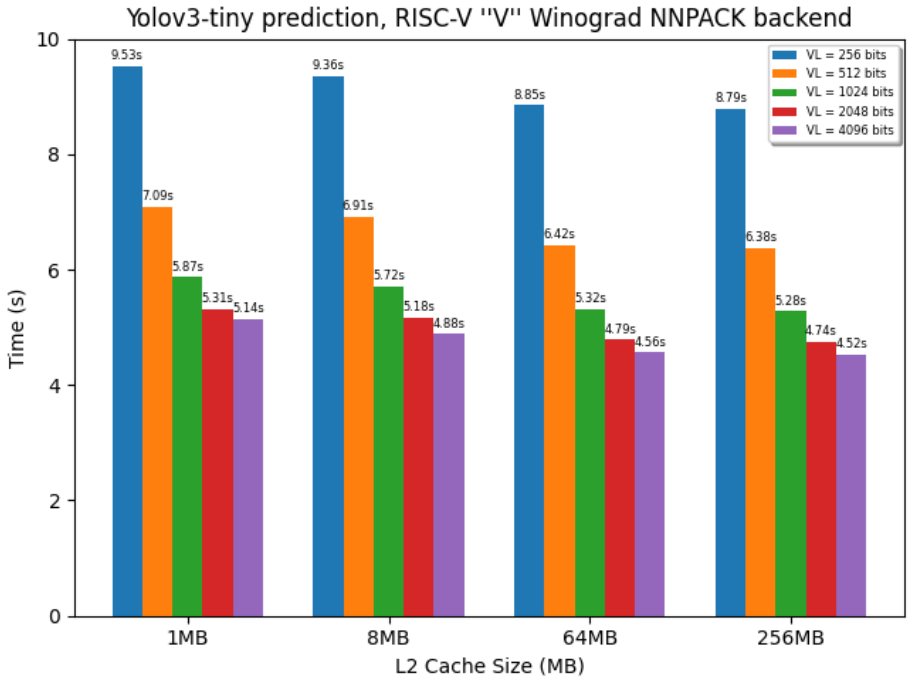


Figure 4.9: RISC-V Winograd benchmarks with different L2 cache sizes and vector lengths.

As can be seen in tables 4.8 and 4.9 these algorithms were comparatively fast, in comparison to the baseline results. The slowest result for GeMM was 3.38 seconds with 1 MB L2 cache and 256 bits vector length, while the fastest was 0.55 seconds with 256 MB L2 cache and 4096 bits vector length. Something to be noted was that at the time of the presentation the RISC-V "V" Winograd benchmark was still running so results are incomplete.

Table 4.12: Speedup by L2 cache size for Winograd and GeMM.

Cache size	1MB	8MB	64MB	256MB
GeMM VL=256	1.0	1.29	1.78	1.84
GeMM VL=1024	1.0	1.27	1.78	1.86
GeMM VL=4096	1.0	1.45	1.93	2.10
Winograd VL=256	1.0	1.02	1.08	1.08
Winograd VL=1024	1.0	1.03	1.10	1.11
Winograd VL=4096	1.0	1.05	1.13	1.14

Table 4.12 shows the speedup per cache size for GeMM and Winograd for selected vector lengths. It can be seen that the fastest speedup for GeMM was with 4096 bits vector length and the largest 256 MB L2 cache, with a speedup of 2.10. For Winograd the fastest speedup was similarely with 4096 bits of vector length and the largest 256 MB L2 cache, with a speedup of 1.14.

Table 4.13: Winograd and GeMM speedup compared to 8x8 baseline with same cache size.

Cache size	1MB	8MB	64MB	256MB
GeMM VL=256	1.91	2.08	2.68	2.68
GeMM VL=1024	4.17	4.68	5.85	6.01
GeMM VL=4096	5.57	6.85	8.48	8.96
Winograd VL=256	0.67	0.58	0.58	0.56
Winograd VL=1024	1.10	0.96	0.96	0.93
Winograd VL=4096	1.26	1.12	1.12	1.09

The resulting speedups compared to the $8x8$ benchmark for selected vector lengths can be seen in table 4.13. The GeMM algorithm provides the slowest speedup of 1.91 with a vector length of 256 bits and a L2 cache size of 1 MB. The largest speedup of 8.96 was seen with a vector length of 4096 bits and a L2 cache size of 256 MB. Winograd on the other hand has the smallest speedup of 0.56 at 256 bits vector length and 256 MB L2 cache. The largest Winograd speedup was 1.26 with 4096 bits of vector length and 1 MB of L2 cache.

4.2.2.2 Auto-vectorized baseline

This section has the results of the benchmarks run using the scalar baseline FFT implementation that was auto-vectorised by the compiler.

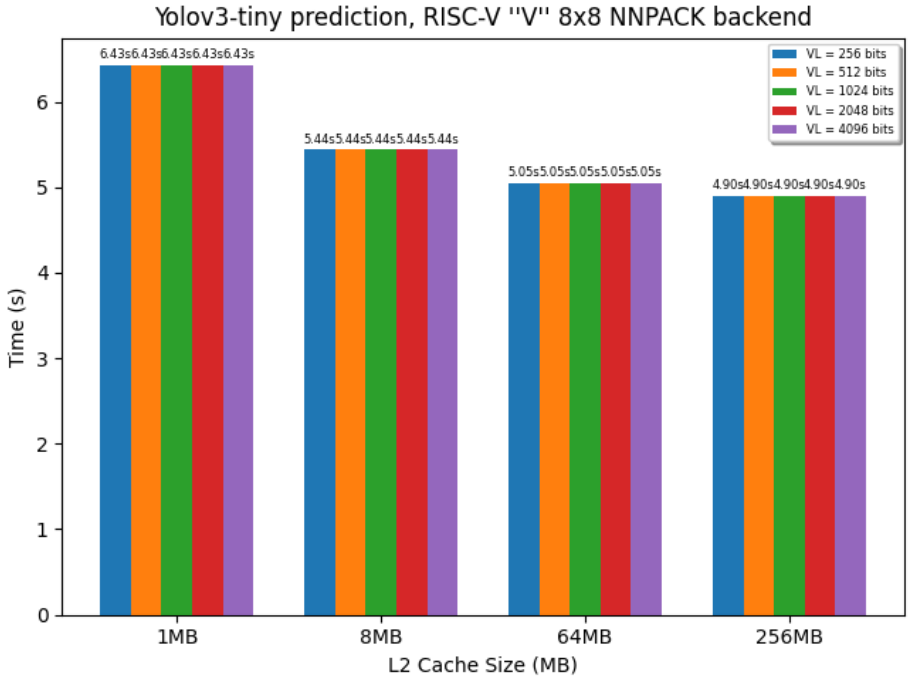


Figure 4.10: RISC-V auto vectorised FFT $8x8$ benchmarks with different L2 cache sizes and vector lengths.

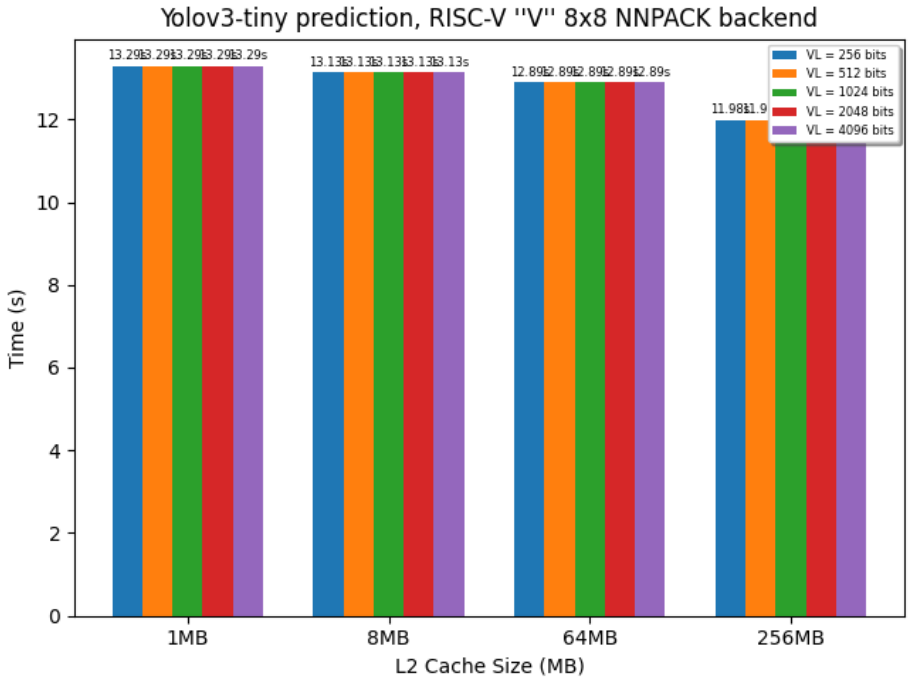


Figure 4.11: RISC-V auto vectorised FFT 16x16 benchmarks with different L2 cache sizes and vector lengths.

These two benchmarks saw an insignificant speedup with increasing vector lengths. The auto-vectorised results are also within ±1% of the baseline so a comparison tables are omitted.

4.2.2.3 Our implementation

The final section describes the results of the benchmarks run using our implementation of the FFT algorithm with the RISC-V "V" extension, it also includes comparisons with the results from the previous sections. Due to limitations with the RISC-V implementation the 8x8 benchmark only includes vector lengths 256, 512 and 1024 bits of vector length. The 16x16 benchmark includes 512, 1024, 2048 and 4096 bits of vector length.

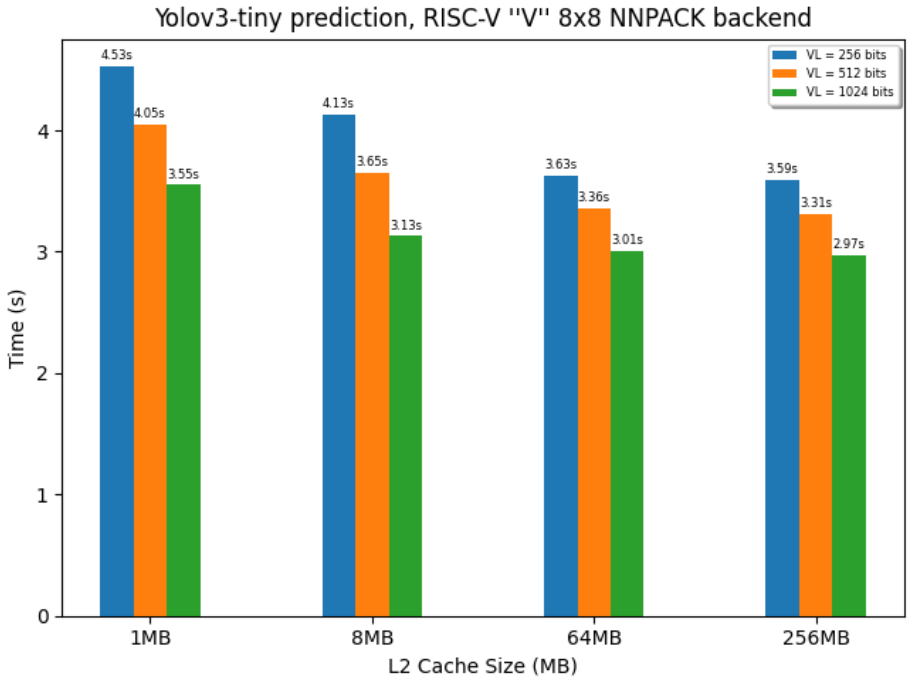


Figure 4.12: FFT 8x8 benchmarks with different l2 cache sizes and vector lengths.

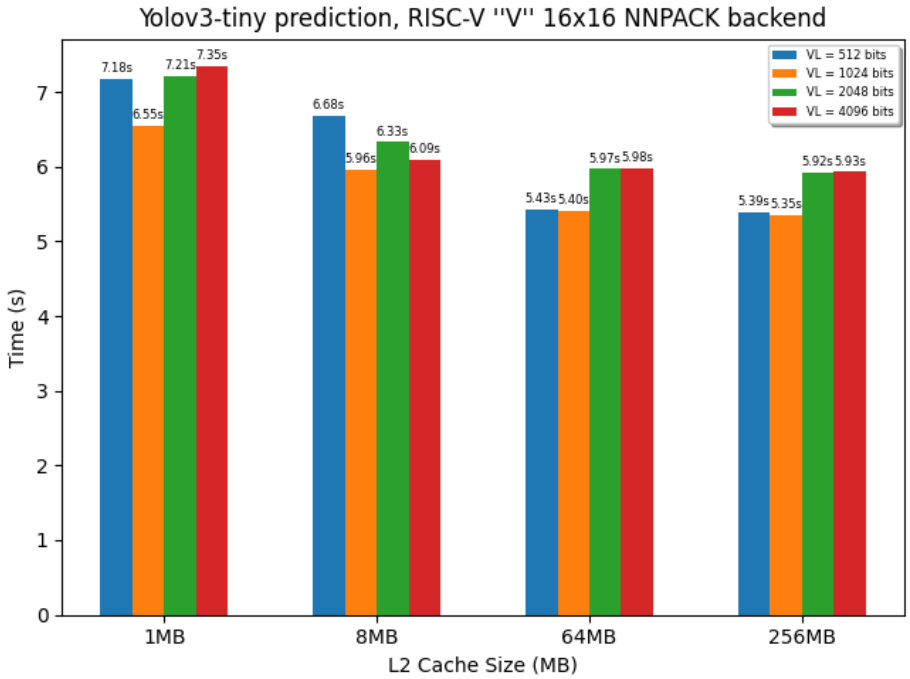


Figure 4.13: FFT 16x16 benchmarks with different l2 cache sizes.

The results for FFT $8x8$ with RISC-V "V" instructions can be seen in figure 4.12. The slowest time with a vector length of 256 bits and a L2 cache size of 1 MB was 4.53 seconds, while the fastest time with vector length 512 and L2 cache size 64 MB was 3.01. In comparison, the $16x16$ variant had a slowest time of 7.35 seconds with

1 MB of L2 cache and 4096 bits of vector length. The fastest time was and 5.43 seconds with 64 MB of L2 cache. and 512 bits of vector length.

Table 4.14: Speedup by L2 cache size for RISC-V "V" FFT.

Cache size	1MB	8MB	64MB	256MB
Speedup 8x8, VL=256 bits	1.0	1.10	1.25	1.26
Speedup 8x8, VL=512 bits	1.0	1.11	1.21	1.22
Speedup 8x8, VL=1024 bits	1.0	1.13	1.17	1.20
Speedup 16x16, VL=512 bits	1.0	1.07	1.30	1.33
Speedup 16x16, VL=1024 bits	1.0	1.10	1.21	1.22
Speedup 16x16, VL=4096 bits	1.0	1.21	1.23	1.24

The impacts of increasing cache size can be seen in table 4.14. Each variant was faster with increasing cache size, though the impact was not linear. As can be seen in the table the $8x8$ version has a larger speedup as cache size increases for the larger vector length, similarly the $16x16$ variant has the largest speedup for each cache size increase.

Table 4.15: RISC-V "V" FFT speedup compared to baseline with same cache size. 8x8 compared to 8x8 and 16x16 compared to 16x16.

Cache size	1MB	8MB	64MB	256MB
Speedup 8x8, VL=256 bits	1.43	1.33	1.40	1.09
Speedup 8x8, VL=512 bits	1.60	1.50	1.51	1.49
Speedup 8x8, VL=1024 bits	1.82	1.75	1.69	1.66
Speedup 16x16, VL=512 bits	1.86	1.97	2.42	2.23
Speedup 16x16, VL=1024 bits	2.03	2.20	2.39	2.24
Speedup 16x16, VL=4096 bits	1.81	2.15	2.16	2.03

Table 4.15 shows speedup of the RISC-V "V" FFT compared to the baseline versuin. It can be seen that both variants of RISC-V "V" FFT were faster than the baseline version, with the $16x16$ variant having the largest speedup.

Compared to the auto-vectorised implementation we found that the network prediction time was within $\pm 1\%$ of the baseline version, therefore their results are not compared separately with our RISC-V "V" implementation.

The final comparisons of our RISC-V "V" results was with the results of the Winograd and GeMM benchmarks that can be seen in tables 4.16 and 4.17. The results show that the Yolov3-tiny network our FFT $8x8$ implementation took between 1.35 (VL=256, L2 Cache=1MB) and 3.57 (VL=1024, L2 Cache=256MB) times as long for network prediction compared with the GeMM implementation with the same cache sizes and vector length. Finally, the FFT $16x16$ implementation took between 3.45 (VL=256, L2 Cache = 1MB) and 11.11 (VL=4096, L2 Cache = 256MB) times as long for network prediction.

Table 4.16: RISC-V "V" FFT speedup compared to GeMM with same cache size and vector length.

Cache size	1MB	8MB	64MB	256MB
Speedup 8x8, VL=256 bits	0.74	0.64	0.52	0.51
Speedup 8x8, VL=512 bits	0.51	0.45	0.35	0.34
Speedup 8x8, VL=1024 bits	0.44	0.37	0.29	0.28
Speedup 16x16, VL=512 bits	0.29	0.25	0.29	0.21
Speedup 16x16, VL=1024 bits	0.24	0.20	0.16	0.15
Speedup 16x16, VL=4096 bits	0.17	0.15	0.10	0.09

Table 4.17: SVE FFT speedup compared to Winograd with same cache size and vector length.

Cache size	1MB	8MB	64MB	256MB
Speedup 8x8, VL=256 bits	2.10	2.27	2.14	2.45
Speedup 8x8, VL=512 bits	1.75	1.62	1.91	1.93
Speedup 8x8, VL=1024 bits	1.65	1.83	1.77	1.78
Speedup 16x16, VL=512 bits	0.98	1.03	1.18	1.18
Speedup 16x16, VL=1024 bits	1.06	0.96	0.99	0.99
Speedup 16x16, VL=4096 bits	0.70	0.80	0.76	0.76

Compared with Winograd the results are: FFT $8x8$ was up to 2.27 times as fast as Winograd in the best case (VL=256 bits, L2 Cache=8MB), though it should be noted our Winograd results are anomalously slow. The slowest result for the $8x8$ comparison was a speedup of 1.65 at a vector length of 1024 bits and a cache size of 1 MB. FFT $16x16$ achieves a speedup of between 0.70 with 4096 bits of vector length and a L2 cache size of 1 MB and 1.18 with 4096 bits of vector length and a L2 cache size of 256 MB.

5

Conclusion

In this study, we aimed to implement the FFT convolution for CNNs using the ARM SVE and RISC-V "V" extensions while investigating the performance impact of architectural parameters and comparing the results to different convolutions and implementations. Although we were not able to fully investigate the effect of longer vector lengths due to simulator issues, our findings show that for the network and application tested VLA programming using the ARM SVE and RISC-V "V" extensions to vectorise FFT was viable and outperforms the baseline. The SVE implementation showed a speedup of up to two times compared to the scalar baseline with the short vector lengths tested. Notably, the same application experienced an increasing speedup on simulated architectures with longer vector lengths, highlighting the benefits of portable VLA code. In addition, we find that the compiler auto vectorisation for SVE was not able to take advantage of increasing vector lengths and cache sizes to the same degree as our manually vectorised code. Our FFT implementation is outperformed by GeMM and Winograd by 2-4 times on the SVE architecture and 3-11 times on the RISC-V "V" architecture, which was expected for a network using small kernel sizes where the advantages of FFT compared to Winograd and GeMM are minimal. Another thing to take into consideration is that due to a lack of time our implementations did not take advantage of multiple input channels, which further advantaged GeMM and Winograd.

5.1 Reflections

We faced a lot of challenges regarding the relative obscurity of the tools used. It was, even with help, hard to set up the tools required. First we needed cross compilers to compile the VLA code, then we needed to adapt the Darknet and NNPACK projects to compile with those compilers. Then we needed to get the compiled binaries to run with the gem5 simulator, where we encountered numerous issues at each step. These difficulties, combined with the fact that each benchmark took a long time to run, made it hard to iterate on our implementations and narrowed the scope of our thesis significantly.

In hindsight, the methodology would have benefited from less ambitious benchmarks, with fewer investigated cache sizes and vector lengths to cut down on simulation time. This could have allowed us to investigate more networks with different kernel sizes and different FFT algorithms to get more insight on in which cases FFT could

benefit the most from vectorisation in comparison to other alternatives.

In general, the results align with what we expected. In all cases there was a speedup with vectorisation, which increased further with larger vector lengths. We also expected GeMM and Winograd to outperform FFT on networks primarily composed of layers with small kernel sizes, like the Yolov3-tiny network we used for our benchmarks. Unfortunately, we were not able to find a network with more suitable kernel sizes compatible with the tools we were using. The one thing that sticks out is how the compiler SVE vectorised scalar implementation of FFT did not significantly benefit from vector length or cache size increases, suggesting it was unable to vectorise the application by itself.

5.2 Problems with the gem5 simulator

During our work we ran into major issues with the gem5 simulator used for simulating different hardware architectures. For the ARM SVE part of the project this manifested as page fault exceptions whenever the simulator was run with a vector length greater than 512 bits. We spent a large part of the time working on this thesis trying to solve this issue. At first we suspected an issue with our code, but after encountering the same issue with the auto-vectorised code and the already implemented GeMM and Winograd code we realised it was an issue with gem5. We then tried using older and newer versions of gem5 as well as different versions of the cross compiler, but to our frustration the issue never went away. This greatly limited the scope of our benchmarks, only allowing us to run with 256 and 512 bits of vector length rather than also including 1024 and 2048 bits as originally planned.

Additionally, we ran into a similar issue with the plct-gem5 fork that had added support for the RISC-V ”V” extension to the gem5 simulator. In this case we encountered the page fault exception first when the vector length reached 8192 bits, limiting us to running benchmarks with 256, 512, 1024, 2048 and 4096 bits of vector length. Similarly to the first case we were never able to figure out the origin of these page fault exceptions.

Finally, another issue with gem5 is the time each simulation takes. A single simulated prediction using Darknet with the Yolov3-tiny network for one cache size and one vector length takes up to 6 hours. Now consider the fact that our SVE benchmarks originally aimed to cover four different vector lengths and four different cache sizes. That means just one of the benchmarks can take up to four days to run. This made both investigating and iterating over our implementations very difficult and time-consuming.

5.3 Future work

Due to the time spent on the previously mentioned issues with gem5 there were parts of our original goals we were unable to carry out that would be suitable for future work.

5.3.1 Multiple input channels

Something our implementation does not take advantage of that GeMM and Winograd does is using multiple input channels. This increases the data parallelism further which could lead to improvements, especially with longer vector lengths and larger cache sizes.

5.3.2 Investigate with real hardware

As our benchmarks were run in simulated hardware architectures it would be interesting to perform similar tests using real hardware. This is possible with the SVE extension, though options for running the RISC-V ”V” extension on real hardware are very limited.

5.3.3 Benchmarks with further vector lengths

It would be interesting to extend our benchmarks with the full possible vector lengths of the ARM SVE and RISC-V ”V” vector extensions rather than the limited subset we were able to work with. This could be possible with a patched gem5 simulator or with physical hardware rather than using simulators.

5.3.4 Different FFT algorithms

This thesis has mainly focused on the benchmark results of our own FFT algorithm as well as the scalar FFT implementation provided by NNPACK. It would be of interest to further expand this by comparing multiple different FFT algorithms against eachother.

5.3.5 Different CNNs

In this thesis all benchmarks have run the Yolov3-tiny network due to it’s small size, both in the size of the layers and in the amount of layers. There is a lot of room to expand upon this with other networks, especially networks with larger kenel sizes where the FFT algorithm has a chance to outperform Winograd and GeMM.

5.3.6 Coarse-grained auto-tuning

Finally, we were interested in coarse-grained auto-tuning of convolutions in CNNs. With this we mean that instead of running benchmarks for just one complete network it would be interesting to run benchmarks for multiple different networks layer by layer. An investigation could then conclude if it is possible to predict for each convolutional layer which algorithm would be optimal, taking into account software and hardware parameters like kernel size, vector length and cache size.

Bibliography

[1] Kjell Magne Fauske, *Example: Radix-2 fft signal flow*, 2006. [Online]. Available: https://texample.net/tikz/examples/radix2fft/ (visited on 06/01/2023).

[2] D. Takahashi and F. Franchetti, "Ffte on sve: Spiral-generated kernels," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPCAsia2020, Fukuoka, Japan: Association for Computing Machinery, 2020, pp. 114–122, ISBN: 9781450372367. DOI: 10.1145/3368474.3368488. [Online]. Available: https://doi.org/10.1145/3368474.3368488.

[3] N. Kitai, D. Takahashi, F. Franchetti, T. Katagiri, S. Ohshima, and T. Nagai, "An auto-tuning with adaptation of a64 scalable vector extension for spiral," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2021, pp. 789–797.

[4] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[5] S. R. Gupta, N. Papadopoulou, and M. Pericas, *Accelerating cnn inference on long vector architectures via co-design*, 2022. arXiv: 2212.11574 [cs.DC].

[6] J. Gu, Z. Wang, J. Kuen, *et al.*, "Recent advances in convolutional neural networks," *Pattern Recognition*, vol. 77, pp. 354–377, 2018, ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2017.10.013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0031320317304120.

[7] A. Zlateski, Z. Jia, K. Li, and F. Durand, "The anatomy of efficient fft and winograd convolutions on modern cpus," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19, Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 414–424, ISBN: 9781450360791. DOI: 10.1145/3330345.3330382. [Online]. Available: https://doi.org/10.1145/3330345.3330382.

[8] Y. Zhou, M. Yang, C. Guo, *et al.*, "Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2021, pp. 214–225.

[9] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, *Fast convolutional nets with fbfft: A gpu performance evaluation*, 2014. arXiv: 1412.7580 [cs.LG].

[10] A. Pohl, M. Greese, B. Cosenza, and B. Juurlink, "A performance analysis of vector length agnostic code," in *2019 International Conference on High Performance Computing & Simulation (HPCS)*, 2019, pp. 159–164. DOI: 10.1109/HPCS48598.2019.9188238.

[11] V. G. Reddy, "Neon technology introduction," *ARM Corporation*, vol. 4, no. 1, pp. 1–33, 2008.

[12] N. Stephens, S. Biles, M. Boettcher, *et al.*, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017. DOI: 10.1109/MM.2017.35.

[13] *V for vector: Software exploration of the vector extension of risc-v*, May 2020. [Online]. Available: https://www.european-processor-initiative.eu/v-for-vector-software-exploration-of-the-vector-extension-of-risc-v/.

[14] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 4013–4021. DOI: 10.1109/CVPR.2016.435.

[15] B. Kågström, P. Ling, and C. van Loan, "Gemm-based level 3 blas: High-performance model implementations and performance evaluation benchmark," *ACM Trans. Math. Softw.*, vol. 24, no. 3, pp. 268–302, Sep. 1998, ISSN: 0098-3500. DOI: 10.1145/292395.292412. [Online]. Available: https://doi.org/10.1145/292395.292412.

[16] H. V. Sorensen, D. Jones, M. Heideman, and C. Burrus, "Real-valued fast fourier transform algorithms," *IEEE Transactions on acoustics, speech, and signal processing*, vol. 35, no. 6, pp. 849–863, 1987.

[17] M. Dukhan, *Nnpack*, https://github.com/Maratyszcza/NNPACK, 2016.

[18] J. Redmon, *Darknet: Open source neural networks in c*, http://pjreddie.com/darknet/, 2013–2016.

[19] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[20] M. Dukhan, *Maratyszcza/nnpack: Acceleration package for neural networks on multi-core cpus*. [Online]. Available: https://github.com/Maratyszcza/NNPACK.

[21] J. Lowe-Power, A. M. Ahmad, A. Akram, *et al.*, "The gem5 simulator: Version 20.0+," *CoRR*, vol. abs/2007.03152, 2020. arXiv: 2007.03152. [Online]. Available: https://arxiv.org/abs/2007.03152.

[22] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05, Anaheim, CA: USENIX Association, 2005, p. 41.

[23] Plctlab, *Plctlab/plct-gem5: Upstream: Https://github.com/ralc88/gem5*. [Online]. Available: https://github.com/plctlab/plct-gem5.