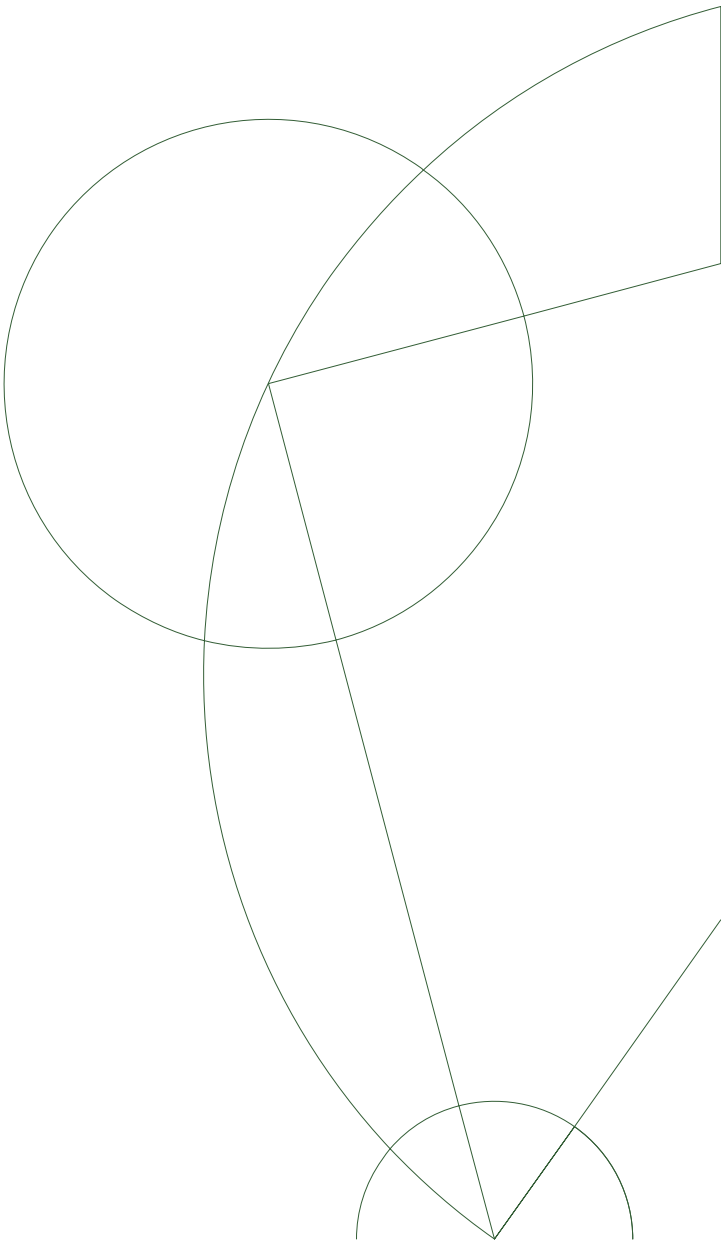
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Bsc Thesis in Computer Science

Christopher Kyed <xrl165>

Convolution in Futhark

Implementing fast algorithms for image processing in Futhark



Supervisor: Troels Henriksen

15th of June 2019

Abstract

In modern times, convolutional neural networks play a big part in how we solve many complex problems. As these problems are often very computational demanding, we want to ensure that we have the most efficient algorithms implemented in the the convolutional layer, as this is where most of the computations are performed. In this thesis we explored 4 algorithms for convolution and how well we were able to implement them in the data-parallel GPGPU focused language Futhark. Here we saw that Futhark was indeed expressive enough to handle our implementation and its functions. Furthermore, we saw that the language was flexible enough, so that we were able to tune each algorithm to achieve even greater run times. The run times were found as we benchmarked our programs against each other, as well as against a popular Python library function. The programs performed convolution across many image sizes in a matter of milliseconds. As expected, this highly outperformed the Python library, thus confirming the reason for performing convolution in parallel as opposed to sequentially. The individual run times of each algorithm were surprising, however, as they did not match our theoretically based expectations. This highlighted the friction between a mathematical formulation and an optimal Futhark implementation. Due to the nature of the Futhark compiler and the lack of knowledge thereof, we were not able to accommodate all the benefits of these algorithms. As our understanding and the compiler itself continue to grow, we can expect even faster implementations in the future.

Keywords: Image Processing, Algorithms, General-Purpose Graphics Processing Units, Neural Network, Futhark.

Contents

1 Introduction **3**
- 1.1 Thesis objective . . . 3
- 1.2 Thesis structure . . . 4
- 1.3 Overview of Futhark . . . 4
- 1.4 Code and data . . . 5

2 Technical Background **6**
- 2.1 Convolution . . . 6
- 2.2 Im2col convolution . . . 9
- 2.3 Memory efficient convolution (MEC) . . . 10
- 2.4 Winograd Minimal Filtering Algorithm . . . 11
- 2.5 Pros vs. Cons . . . 14

3 Design and Implementaiton **15**
- 3.1 Algorithm Design . . . 15
- 3.2 Implementation . . . 17
- 3.3 Testing . . . 24
- 3.4 Shortcomings . . . 24
- 3.5 Integration into neural network . . . 25

4 Results **27**
- 4.1 Benchmarks . . . 28
- 4.2 Practical example . . . 30
- 4.3 Reflection . . . 30

5 Conclusion **32**
- 5.1 Future work . . . 32
- 5.2 Evaluation . . . 32
- 5.3 Acknowledgements . . . 33

A Benchmarks **34**
- A.1 Futhark benchmarks results . . . 34

B Programs **36**
B.1 Interpret data no prior padding 36

C Tests **37**
C.1 Test of correctness 37

D Image convolution example **39**
D.1 Edge detection 39

Chapter 1

Introduction

In a modern society where computers and other devices dictate how we solve problems, it is vital that we develop the most suitable algorithms to solve these. There is a continual increase in the amount of new problems that we are able to solve due to the computational power that computers have offered us during the last couple of decades. It is therefore our responsibility to ensure that we make the best use of this power. Self-driving cars, medical image analysis and natural language processing are all examples of this. This is closely tied to convolutional neural networks, where making best use of computational power is vital due to the sheer volume of computations that need to be carried out. Many of these problems, such as image processing, can be broken down into simple arithmetic operations, and have therefore been around for many decades. As a result, the area of arithmetic complexity of computations has been heavily explored in the past by people like Strassen, Winograd and Cooley & Tukey [14]. It is not until recent years, however, that their ideas and research have become very relevant as we now have a significant use for them. Nowadays, we implement their findings into convolutional neural networks that can produce impressive results with regard to image recognition problems. As these networks often take several days to train and use immense computational power, it becomes apparent now more than ever before, that we need to go back and analyze the research previously carried out on this matter.

1.1 Thesis objective

The objective of this thesis is to shed some light on how well we can implement algorithms for image processing in the data-parallel language, Futhark[1]. The motivation behind the project came from Minh Tran's and his work [13] concerning convolutional neural networks in Futhark. Only one implementation of convolution (Im2col) was made in this project due to time constraints. As a result, we want to pick up where he left off and explore algorithms that might speed up the convolutional layer of the neural network.

[1] `https://futhark-lang.org/`

Firstly, it is our goal to investigate how well these algorithms can be implemented and how well they perform. By doing this, we are able to determine how suitable Futhark is for convolution operations in general, and if the language has the necessary expressiveness for our task. This is especially interesting as Futhark has many limitations, such as constraints to semantics and data-structures, in order to produce such efficient code. Secondly, as Futhark produces high-performance code for GPU's, we also explore the most efficient methods for performing convolutional computations on these devices. By doing this, we are able to showcase the difference in architecture between GPU and CPU's. This shows the difference between the CPU's sequential and the GPU's parallel execution and how well convolution is suited for both of these.

1.2 Thesis structure

Chapter 2 explains the basic principles of convolution and the arithmetic that go into computing these. Furthermore, the chapter aims to shed light on the technical details of the algorithms which are implemented in this project.
Chapter 3 includes a detailed overview of the code and the implementation stage of this project. In addition to this, it explains the design process as well as which choices were made and why.
Chapter 4 gives an overview of the results of the project and the reasoning behind these. We also explore how our algorithms compare with other implementations, and how they measure up to our expectations.
Chapter 5 aims to sum up the thesis and provide a conclusion that includes an evaluation of the project, and an overview of the future work that lies ahead.

1.3 Overview of Futhark

Futhark is a purely functional data-parallel array language that belongs to the ML-family. More specifically, it is a high-level, monomorphic, statically typed, strictly evaluated language, developed by the APL group at the Department of Computer Science at the University of Copenhagen. It was developed with the purpose of offering a programming language, with an optimising compiler generating high-performance code for GPU's via OpenCL. Futhark can either compile executable files that can be run directly on the GPU, or into libraries for Python or C. It is important to note here that Futhark is not intended as a general-purpose language, its goal is to express computational kernels that are linked to popular languages [8]. In other words, we are able to accelerate the part of a program that is computationally demanding, hence the latter of the two types of compilation represents the intended use of the language.

Futhark has a range of build-in functions known as Second-Order Array Combinators (SOACs). It is through these that Futhark achieves much of its data parallelism, as these functions are compiled directly to data-parallel code. The functions in question are, among others, `map`, `reduce`, `scan` `scatter`. The signature and functionality of the

first four are similar to the ones you would find in other popular functional languages such as F# or Haskell.
We can construct the type signature and function semantics as follows:

- `map` $(f : a \to x) \to (as : [n]a) \to [n]x$
  `map f [`x_1`, `x_2`,..., `x_n`] = [f `x_1`, f `x_2`,..., f `x_n`]`
  Takes a function and applies it to each element of a given array.

- `reduce` $((op : a \to a \to a) \to (ne : a) \to (as : [\,]a) \to a$
  `reduce` $\oplus$ `ne [`x_1`, `x_2`,..., `x_n`] = [ne` $\oplus$ x_1 $\oplus$ x_2 ... $\oplus$ x_n`]`
  Takes a binary operator $\oplus$ that is associative, a neutral/identity element and an array. The function then combines all the array elements using the given operator.

- `scan` $(op : a \to a \to a) \to (ne : a) \to (as : [n]a) \to [n]a$
  `scan` $\oplus$ `ne [`x_1`, `x_2`,..., `x_n`] = [`x_1`, `x_1 $\oplus$ x_2`,..., `x_1 $\oplus$ ... $\oplus$ x_n`]`
  Very similar to `reduce`, however, instead of returning one combined result, it returns each intermediate result as well as the final one.

Another set of functions that are heavily relied upon throughout the implementation have to do with manipulating data representations of arrays. These include:

- `Transpose`: Turn matrix rows into columns
- `Flatten`: collapse array combining the outer two dimensions of an array.
- `Unflatten`: splits the outer dimension of an array in two.
- `Zip`: construct an array of pairs from two arrays.
- `Unzip`: turn an array of pairs into two arrays.

These operations are mostly "free" in Futhark because of the compilers' ability to easily fuse these operations together with other larger computations.

As Futhark focuses heavily on compiling data-parallel code, this results in there being constraints on the expressiveness and semantics of the language. The most prominent of these in our case being, that it does not support irregular arrays. This means that all inner arrays must have the same shape, for example $[[1, 2], [3, 4, 5]]$ is not allowed. This of course imposes a problem for us, as our algorithms are designed for taking in and outputting different multi-dimensional arrays. We explore later in chapter 3 how this limitation ended up affecting our implementation and design.

1.4 Code and data

The project and the algorithms developed can be found at `https://github.com/Kyediis/Futhark_Convolution`. This repository includes the implemented algorithms, benchmarking, testing as well as a practical example.

Chapter 2

Technical Background

Convolution is an operation which describes the way that one function affects and changes another. It is therefore used in describing many different mathematical concepts. In this report we refer to it as the way that one image is able to shape and affect another through a *convolutional operation*. The main goal of such an operation is to change the input to represent specific features. These features all depend on the specific function we decide to convolve with. Similarly a series of different functions can be used to draw out a more specific set of attributes from our input.

How we implement this convolution differs from problem to problem, and in each of these sub-problems there exists many different algorithms to achieve this convolution. Which algorithm to use differs greatly according to situation, due to the many trade-offs between the algorithms as we see in the next section. Here we explore in further detail what convolution entails for image processing, and how convolutional operations are performed in this case. We furthermore cover 4 algorithms that are all aimed at this and the attributes that they possess.

2.1 Convolution

In a strictly theoretical manner, a convolution describes the function that is achieved when we have one function shaped and modified by another [2]. We can write up the formal notation for convolution as follows:

Definition 2.1.1 $(f * g)(t) \equiv \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$

Convolution is thus defined as the integral of the product of the two functions after one is reversed and shifted. This can be understood as a weighted average of the function $f(\tau)$, at the moment t. Here the weight is given by $g(-\tau)$ shifted by amount t. In other words, when our t changes our weighing function focuses on different parts of our input function. This is indeed very abstract and hard to wrap your head around, fortunately we can look at a much more practical example of how we can use convolution.

2.1.1 Convolutional operations

The term convolutional operation refers to this same ”shaping” as described above, however in this case we represent the functions as 2D matrices instead, or more specifically ”signals”. This is useful, as we can now use convolution to represent how an image can be shaped and affected by another.
To see how this is implemented for image processing we introduce three terms:

- **Input image**:
  Our input image is the first function that we want to convolve with and shape. We can represent an image as a three-dimensional array as we have $n \times m$ pixels spread across c colour channels giving us $n \times m \times c$. We then split our image by channel to obtain the two-dimensional signals, which is what we want to perform our convolution on.

- **Feature detector**:
  The feature detector is the second function that we use to shape our image. This is also a signal which we choose to represent as a two dimensional matrix. These are typically smaller than the image and usually have orders of 1x1, 3x3, 5x5, 7x7. These kernels do come in many different shapes and are not necessarily symmetric in dimensions and values. However, modern kernels usually are of order $3x3$ with perfectly symmetrical values. In the project we have therefore chosen to focus on these, as it simplifies the implementation slightly, which we see in the next section.

  The main purpose of these feature detectors are, as the name states, to find features in our image. Examples of these are edge detection, blurring and sharpening, which can be used in a number of ways to extract specific part of an image. More precisely, these signals are constructed in such a way that when they are convolved with a picture, it makes us put emphasis on specific pixels while ignoring others.
  Bear in mind that the terms feature detector, kernel and filter can all be used interchangeably, but for the sake of consistency we choose to use **kernel** throughout this report.

- **Feature map**:
  The output is what we call a ”feature map” or an ”activation map”, and it is the output signal of our convolution. It contains a set of new pixels that have all been shaped by the input kernel. As a result, we now have an image whose values put emphasis on a certain feature which is determined by the kernel that was chosen.

2.1.2 Direct Convolution

We can now look at how a set of convolutional operations are performed:

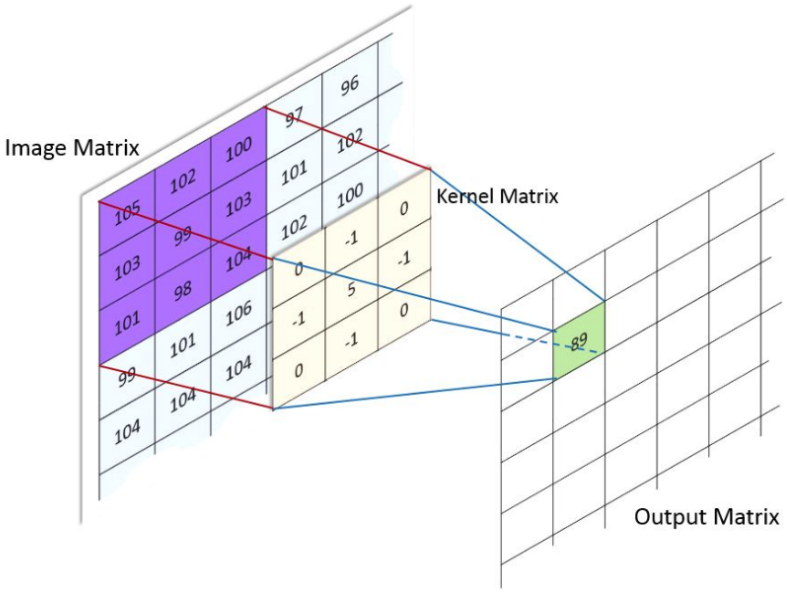


Figure 2.1: Convolutional operation

As visualized in Figure 2.1 we place the kernel over a corresponding image tile with matching shapes. We then first do pointwise multiplication between the kernel and image tile, also known as the *Hadamard product*. Then we sum up all the values and place this value in our activation map. We then slide our kernel by our step size, usually one, and calculate another value for our output as shown in Figure 2.2. This is usually done in row-major, and we are done when we have convolved our kernel with each unique tile. This kind of convolutional operation is also known as *Direct Convolution* and is the most straightforward way to do this set of calculations. We also have low memory overhead with this approach as we do not need to allocate anything else than our input, kernel and output.

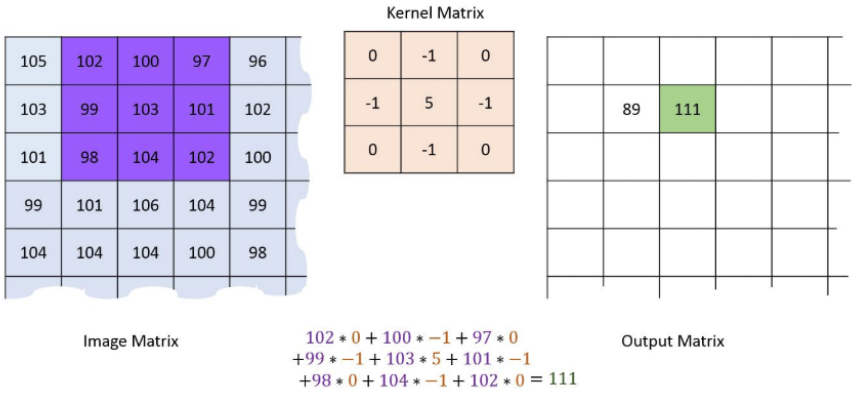


Figure 2.2: Direct Convolution

The reader might realize that our activation map ends up being smaller than the input image as we cannot go outside the edges. This is indeed the case, and we do have an edge case problem in our feature map, fortunately there are ways to handle this.

2.1.3 Edge cases

The most simple way to handle this problem is to simply ignore the edges and place zeros while we are traversing the edge of our output. However, this means that the output image is ”fuzzy” around the edges and is therefore not a desirable result.

A better alternative is to ”pad” the image with zeros before we do the convolutional operations. This means that we add zeroes around our input image so that our operations do not go outside the borders when calculating the edge values for the activation map. By doing this we are able to make sure that the edges are based on the convolution of at least 4 pixels of our input image, instead of 0. It is important to note here that the number of zeroes required to pad the image correctly is linear with the size of the kernel.

2.2 Im2col convolution

The name Im2col is derived from the operation of turning our input image into columns. The main idea of this approach is to take every tile that would be multiplied with the kernel and organize them into a new a *lowered matrix*.

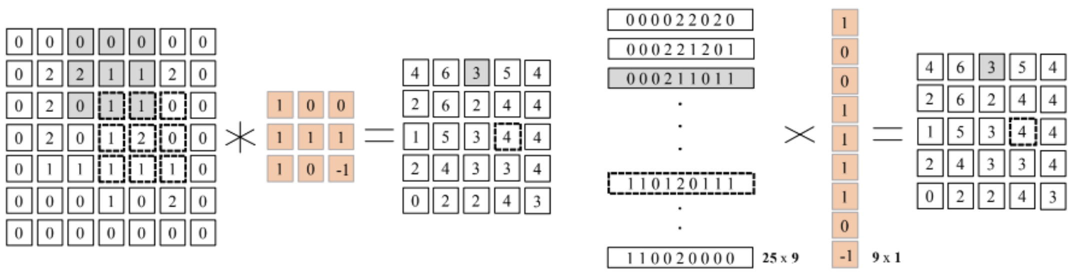


Figure 2.3: Im2col convolution

As seen in Figure 2.3 we take each of the sub-matrix tiles and collapse them into vector form. We then place them all into this lowered matrix, so that each of the rows correspond to the row major sequence of the tiles multiplied for our output. We then collapse the dimensions of our kernel as well, so that we can use vector dot products between our new kernel and each of our rows. The shaded and dotted areas on our input matrix show how 2 tiles are convolved with direct convolution. We observe in the second picture how the same two tiles are lowered into our Im2col matrix, and again used to determine the correct convolution values.

The reason why we use this lowered matrix, is so that we are able to use optimized libraries to compute our convolutions more efficiently. A library such as BLAS[1] provides a set of low-level routines for performing linear algebra operations. These operations take advantage of specific hardware to accelerate performance for specific machines, and can therefore give us a significant performance increase.

The benefit of this approach does come with a significant cost though. As we can see, lowering our input matrix results in having to store a $(o_h * o_w) \times (k_h * k_w)$ matrix with o and k denoting the height and width of feature map and kernel respectively. This gives us a large temporary memory overhead as we need to store a lowered matrix that is even larger than our original image. As a result, it can be derived that the memory requirement grows quadratically with the problem size [6].
This is rather problematic and devices with restricted memory are sure to fall victim to the problems that this algorithm imposes. Fortunately there are other algorithms that handle this problem, one of these being MEC.

2.3 Memory efficient convolution (MEC)

As seen in the previous section, the main problem with Im2col convolution is the major memory overhead that we have. More specifically, this is created by the redundancy in the lowered matrix which scales when we have a larger kernel and smaller step-size. In addition, the overhead gets even worse when our kernel is small relative to the image size. This is the most crucial problem, as this size is the standard among most convolutional neural networks [6] today. The main idea of MEC is therefore to reduce the redundancy which is formed in the lowered matrix, while keeping the same computational pattern so we can still use the BLAS library. We can see how it achieves this by looking at an example:

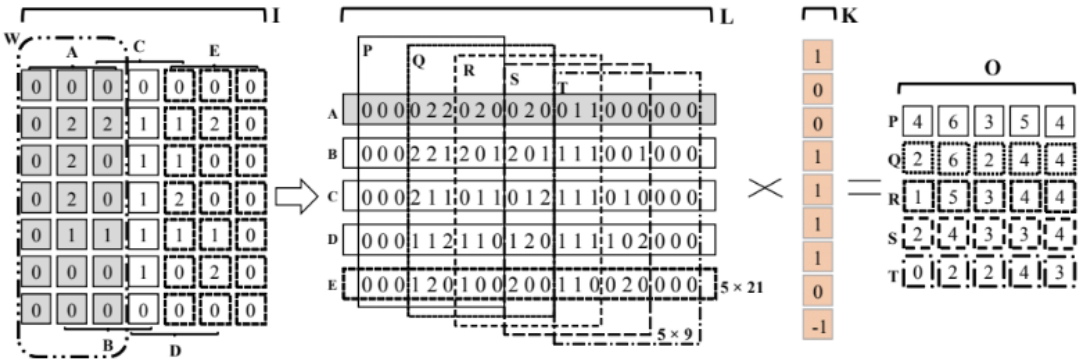


Figure 2.4: MEC convolution

[1] http://www.netlib.org/blas/

As seen in Figure 2.4 instead of lowering each individual sub-matrix, we lower 3 columns at a time. This results in a lowered matrix with size $o_w \times (i_h * 3)$ compared to the $(o_h * o_w) \times (k_h * k_w)$ from Im2col. For the two examples given in the article [6], we formulate the reduction as $1 - \frac{5*(9*3)}{(5*5)*(3*3)} * 100 \approx 54\%$. As we can see here, the new matrix is 54% smaller, and this reduction gets increasingly better for larger inputs.

Now that we have a lowered matrix L that can be allocated more efficiently, we need to make sure we can do the same BLAS operations as in Im2col. We can achieve this by creating 5 abstract partitions within L. These all have the size $o_w \times (k_h * k_w)$, and are obtained by shifting right by 3 in L for each subsequent partition. We can now see that all of our partitions represent the same lowered matrix that Im2col creates. This means, we get our output by simply multiplying our flattened kernel with each row of each partition. We have now succeeded in generating the same lowered matrix but with much less data redundancy, hence reducing memory overhead.

2.4 Winograd Minimal Filtering Algorithm

The last algorithm we have to cover is WMFA or Winograd convolution which is also aimed at reducing data redundancy and with this, arithmetic complexity. We can further showcase this by an example of direct convolution on 4 adjacent corner tiles:

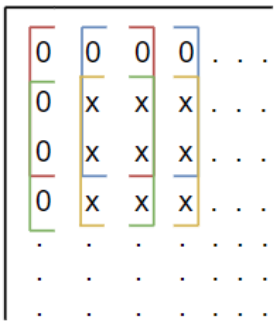


Figure 2.5: Overlapping computations

In Figure 2.5, we have a 0 padded picture where we have taken 4 direct convolutions. Each pair of coloured brackets represents one convolution using a 3×3 kernel. We observe here, that the 4 inner-most elements are reused in all four convolutions, and only the 4 corner elements are unique to each convolutional computation. S. Winograd therefore developed an algorithm to take advantage of this property and convolve these four inner elements at once.

2.4.1 F(2, 3)

Following a 1D example, we formulate WMFA as follows: It is a function that given the size of the output and kernel, returns the values of the output. We do not need the data tile's dimension as it can be derived from the other two inputs. In this specific example we examine $F(2,3)$, meaning we have an output vector of size 2, kernel vector with size 3 and return the values of the 2 output elements.

From theorem 1. [14, p.39] we see that S. Winograd found that the number of general multiplications needed for a minimal filtering algorithm is $\mu(F(m,r)) = m + r - 1$. We can confirm this by testing the 1D algorithm introduced in [14, p.43], letting:

$$data = \begin{bmatrix} i_0 & i_1 & i_2 & i_3 \end{bmatrix}, \quad kernel = \begin{bmatrix} k_0 & k_1 & k_2 \end{bmatrix} \tag{2.1}$$

And define:

$$F(2,3) = \begin{bmatrix} i_0 & i_1 & i_2 & i_3 \end{bmatrix} \begin{bmatrix} k_0 \\ k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix} \tag{2.2}$$

We than have that:

$$m_1 = (i_0 - i_2)k_0 \qquad m_2 = (i_1 + i_2)\frac{k_0 + k_1 + k_2}{2} \tag{2.3}$$

$$m_3 = (i_1 - i_3)k_2 \qquad m_3 = (i_2 + i_1)\frac{g_0 - k_1 + k_2}{2} \tag{2.4}$$

The main idea here is that we transform our data and kernel which is the additions, subtractions and shifts by one as seen from Equations 2.3 and 2.4. Using this transformed data, we are then able to use fewer multiplications as stated by theorem 1. We can confirm this, as we only have 4 multiplications, which adheres to $\mu(F(2,3)) = 2 + 3 - 1 = 4$ with μ denoting number of multiplications.

To simplify things we can express the transformation step as 3 matrices (B^T, G, A^T) instead. Each of these matrices represent the transformation of input data, kernel and output respectively. They therefore have values so that when we multiply our data and kernel vectors with them, we get the 4 same computations from 2.3 and 2.4.
With this notation we can write up the Minimal Filtering Algorithm for 1D input as:

$$O = A^T\left[(Gk) \odot (B^T d)\right]$$

where $\odot$ denotes the hadamard product and matrices given by:

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}, \quad G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}, \quad A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$d = \begin{bmatrix} i_0 & i_1 & i_2 & i_3 \end{bmatrix}^T, \quad k = \begin{bmatrix} k_0 & k_1 & k_2 \end{bmatrix}^T$$

By unrolling the matrix multiplications, one can easily verify that these correspond to arithmetic transformation we observed in the first algorithm.

2.4.2 F(2x2, 3x3)

By nesting the minimal 1D algorithm found in the previous section with itself we are able to construct an algorithm for a two dimensional input:

$$O = A^T \Big[\big[(GkG^T) \big] \odot \big[(B^T dB) \big] \Big] A$$

This nested function now represents the algorithm which determines the output for $F(2x2, 3x3)$. Our function therefore now operates on an 2×2 output matrix and a 3×3 kernel, as opposed to 2 vectors. The data tile d above is now represented by $d = (m+k-1) \times (m+k-1) = (2+3-1) \times (2+3-1) = 4 \times 4$ tile. If we analysis the arithmetic complexity like before we here have $\mu(F(m \times n, k \times r)) = \mu(F(m,k))\mu(F(n,r)) = (2+3-1)(2+3-1) = 16$ multiplications. When we compare this to the 36 multiplication we need in direct convolution, 9 for each of the 4 output elements, we have an arithmetic complexity reduction of $\frac{16}{36} = 2.25$. This is a vital reduction as it applies to each of the $4x4$ input tiles, or "neighbourhoods", that our input image is divided into (visualized in Figure 2.5).

This looks very good, however, we cannot completely obtain this speedup in practice as we until now have ignored the cost of transforming our data, kernel and output. We do this, as the performance penalty is small in practice and because our transformations can be amortized to only make up a small part of the total computation [3]. If we look at the data transformation stage, it consists of two 4×4 matrix multiplications. However, if we skip the 0's and multiplication by one, we can simply reduce these two computations to 32 additions/subtractions. The same holds for the kernel and output that reduce to 28 floating point instructions and 24 additions/subtractions respectively. Furthermore, we only need to transform the kernel once, given that we do not change kernels during convolution. We then reuse this transformed kernel for each neighbouring image tile in our image. The case for data and output is different, however, as we here need to make a transformation for each stride.

2.4.3 F(4x4, 3x3)

Another version of the WMFA method is using a function that outputs 4×4 output instead. Here we divide our image into 6×6 input tiles instead which further reduces our arithmetic complexity as we now have a $\frac{144}{36} = 4$ fold reduction in multiplications compared to direct convolution. To derive the new set of transformation algorithms/-matrices we use a new set of interpolation points as described in [4].

However, there are some issues however when choosing this approach to Winograd convolution. while we have achieved almost twice the reduction in multiplications compared to $F(2x2, 3x3)$, if we look at our transformation costs they are now 156 additions (data), 72 floating point operations (kernel) and 100 additions (output). When we compare this to the other function, we see that our transformation stage computations increase

quadratically with the tile size. And as a result, for larger tile sizes the increase in transformation cost overwhelms any savings in computations [11]. This being said, the trade-off between these two functions is still beneficial in some cases, resulting in both of them being used today. We can further confirm this, as the current hardware sweet spot is reached exactly by these two types of WMFA. These are therefore usually implemented in the convnet layer of a convolutional neural network, as we see in section 3.5.

From this we can conclude that WMFA is not suitable for larger kernels and input tile sizes. In this case it may therfore be beneficial to use FFT-based convolution instead when performing the convolution. This approach is not covered in this thesis but can be further explored in [4].

2.5 Pros vs. Cons

Summing up, there are both pros and cons regarding each of our algorithms.

- **Direct**: Direct convolution has a very simple structure and is therefore easy to implement. Furthermore, it has low memory overhead as we do not transform our input data in any way. The simplicity of the algorithm, is unfortunately also the reason why it suffers from the worst arithmetic complexity. This makes it the most naive implementation on paper, as we do not attempt to make our implementation more efficient.

- **Im2col**: The Im2col algorithm's biggest strength is its lowered matrix which makes us able to take advantage of the efficient BLAS operations. However, the way we lower this matrix creates a lot of memory overhead as we need to allocate a matrix that is many times larger than our input.

- **MEC**: MEC has the same benefits as Im2col as it also uses the BLAS library. In addition to this, it is more memory efficient than Im2col, as it lowers a smaller matrix while achieving the same end result. Even though the memory overhead here is smaller than for Im2col, we still need to allocate a significant amount which is this algorithms largest drawback.

- **Winograd**: Winograd convolution is the algorithm with the most favourable computational complexity due to how it exploits overlapping data. This does come with some computational costs, as we need to make many transformations. These costs are insignificant though, as long as our Winograd function takes and returns small matrices. However, this overhead increases significantly when we attempt to make larger transformations, making any savings in general multiplications redundant.

Chapter 3

Design and Implementaiton

In this chapter we give an in depth look at the implementations related to this project. We go over the general design and implementation phase and give an overview of how each algorithm was constructed. Here also go over some of the many challenges with parallel programming[5, ch. 12] that we found, and how we attempted to combat these.

Before we start, it is important to note that for each of the algorithmic implementations, we have assumed that we have already split our input image into its respective colour channels. This means that each algorithm only works on one $m \times n$ matrix at a time. Furthermore it was decided to only focus on implementations for 3×3 fully symmetrical kernels, as these are very common as explained in Section 2.1. This ensures that we do not need to flip the kernel on its axis during convolution.
In addition to this, it was also chosen to work on data with dimensions whose product is divisible by four. Even though this requirement only applies to Winograd convolution it was chosen as a standard for ease of implementation. Finally, recall from Chapter 1 that Futhark only accepts 2 dimensional arrays with same number of elements in each of the inner arrays. In our case, this means that we cannot have an image with fewer pixels in some columns than others. Even though we could solve this problem with padding, we decided to refrain from using any input of this sort.

3.1 Algorithm Design

The design of our algorithms all adhere to the same overall principles. We first want to pad each of the input data with zeroes as explained in Section 2.1. Following this we want to represent the input data in a specific way depending on the algorithm. We then convolve this data with our kernel in the manner described by the algorithm. Finally we reshape the output if needed and return our finished convolution.

3.1.1 Generic modules

The concepts above that appear in all of the algorithms, are padding as well as matrix and vector multiplication. As a result, it was chosen to construct two modules that would hold these functions.
The first one is our padding module that handles padding each image with zeroes before it is passed in as an argument to our algorithms:

```
module pad = {
  let padImage [rows][cols]
               (image: [rows][cols]f32): [][]f32 =
    map (\row ->
      map (\col ->
             if row > 0 && row < rows+1 && col > 0 && col < cols+1
             then
               unsafe
               image[row-1, col-1]
             else
               unsafe
               0)
        (0...cols+1))
      (0...rows+1)
}
```

Listing 3.1: Padding module

The padding module from listing 3.1 is fairly straight forward. We pass in an image where the data is represented as a 2D matrix. We use two nested `map` operations to traverse the images' dimensions plus two, as our padded picture becomes two elements wider and taller. For each element we check if it is outside the original images boundaries, if it is we place a 0 in the new array, and if not we simply place the original value. It is important to note here that we use the keyword `unsafe`. This is because the Futhark compiler cannot determine whether these indices are safe or not. This keyword is important to take note of, as it is used frequently in the other programs.

```
module matmul = {
  let matmul [n][m][p]
             (x: [n][m]f32) (y: [m][p]f32): [n][p]f32 =
    map (\xr ->
      map (\yc ->
            reduce (+) 0 (map2 (*) xr yc))
              (transpose y))
        x

  let vecmul [n]
             (x: [n]f32) (y: [n]f32): f32 =
    reduce (+) 0 (map2 (*) x y)
}
```

Listing 3.2: Matmul module

The matrix multiplication module in Listing 3.2 has two functions that both have to do with taking dot products, as these are not implemented in the Futhark language yet. Recall from Chapter 2 that we need both normal matrix multiplication and vector multiplication in order to implement our algorithms. As a result we have made use of `reduce` and a `map2` to create the sought out computation.

3.2 Implementation

3.2.1 Direct

If we were to write up the pseudo code for direct convolution from Section 2.1 it would have the following form:

Algorithm 1 Direct convolution

1: PadImage Im
2: **for** $row = 0$ to $Im_{rows} - 1$ **do**
3: **for** $col = 0$ to $Im_{cols} - 1$ **do**
4: DirectConvolution *padded* kernel row col //Conv for each output element.
return Output

Our goal is to first make sure the image is padded with zeros and then traverse it calling our direct convolutional operation on each kernel sized tile. We then return the output that we get from these two loops. In our program the nested for-loop is implemented as a nested map as we saw in the case for modules 3.1.1. The maps take in an array of values that all iterate from 0 to number of columns and rows respectively. By doing this we are then able to pass in the row and column value to our direct convolution function as shown in Listing 3.3:

```
1  let directConvolution [rows][cols] (data: [rows][cols]f32)
2                       (kernel: [3][3]f32) (row: i32) (col: i32): f32 =
3    unsafe
4    let sum =
5      data[row-1,col-1]*kernel[0,0] + data[row-1,col]*kernel[0,1] +
6      data[row-1,col+1]*kernel[0,2] + data[row,col-1]*kernel[1,0] +
7      data[row,   col  ]*kernel[1,1] + data[row,col+1]*kernel[1,2] +
8      data[row+1,col-1]*kernel[2,0] + data[row+1,col]*kernel[2,1] +
9      data[row+1,col+1]*kernel[2,2]
10   in sum
```

Listing 3.3: Direct convolution

In practice, this means that we do not need to use array slicing to access the specific tile, as we can simply use indexing to achieve the same result. This is very important as currently using array slicing inside of map operations makes the Futhark compiler conservative. It cannot guarantee that all slices in the body of a map have the same sizes for all iterations. This means that it anticipates the worst case, that they are not the same size, thus impacting performance. This was discovered late in the design phase, and as a result all of our algorithms was affected by it. This in turn meant that we had to perform this optimization for all of our programs, as we see later in this chapter.

When taking the Hadamard product between tile and kernel as seen from Listing 3.3, it was chosen to unroll the operation. In the beginning of the implementation stage, we opted for this choice due to simplicity. However as it turns out, this would ultimately also allow for us to do the operation without using slices inside of our maps. Looking back, we might have chosen to pass in an array slice to a Hadamard product function declared in our module. The fact that we refrained from doing so, is the main reason why this algorithm required the least amount of tuning compared to the others.

3.2.2 Im2col

In order to implement Im2col we follow a similar approach as direct convolution. However, recall from Chapter 2 that we for this algorithm first interpret our data as a lowered matrix. Our initial implementation achieved this by using array slicing to first select each 3×3 tile. These were then each passed to a separate function which used `flatten` to reshape these tiles to vectors. The operations were done inside a single map running from 0 to $(o_h * o_w)$ giving us the correct lowered matrix. However as we explained in Subsection 3.2.1, we wanted to find a way to achieve this without slicing.

```
1  let interpretData [rows][cols]
2                    (data: [rows][cols]f32): [][9]f32 =
3    let res =
4      unsafe
5      map (\flat ->
6                unsafe
7                let i = 1+flat / (cols-2)
8                let j = 1+flat % (cols-2)
9                in [data[i-1,j-1], data[i-1, j], data[i-1, j+1],
10                   data[i,   j-1], data[i,   j], data[i,   j+1],
11                   data[i+1,j-1], data[i+1, j], data[i+1, j+1]
12                  ])
13         (iota ((rows-2) * (cols-2)))
14   in res
```

Listing 3.4: Im2col interpret data

The function seen in Listing 3.4 aims to gather the aforementioned operations and simplify them. First we have our variables i and j that together correspond to the middle

index of each of our 3×3 tiles. Instead of calling a separate function for flattening the tiles, we simply write out the flattened tile in vector form directly.

After this function, much like direct convolution, we pass our interpreted data to the actual convolutional function. Here we used the `vectormul` function from our `multiplication` module, to compute the convolution between the flattened kernel and each of the rows in our interpreted data. Later on, we did find out that unrolling this operation gave us a speed up, which is why this was also chosen in the final part of the implementation process.

A more desirable implementation could have been using the BLAS library to compute these vector vector dot products. However the implementations of these subroutines are written in Fortran and C, meaning we were not able to test these. Porting this code to Futhark was unfortunately not feasible for the current project, but a very relevant subject for future work as touched upon in Chapter 5.

After making the modification discussed above, the algorithm was made much more efficient. However, this made our implementation kernel size specific, similiar to direct convolution. As mentioned earlier, it was chosen to rely solely on 3×3 kernels for this project's implementations. This being said, when making strict input specific implementations we should still bear in mind why this was chosen and how we might avoid it, something that we discuss in the following chapter.

Finally, it was also while tuning this algorithm that we found another aspect of our design, which impacted performance negatively. In our implementation of the padding module we first allocate a whole new array that is larger than our input data. We then copy every element of the input to this new data structure, ultimately creating a significant amount of overhead for each of our algorithms. We therefore had to find a way to add the padding dynamically while we were selecting tiles for convolution. Naturally this made the above code example significantly more confusing and unmanageable as seen from the Im2col example in Appendix B.1. In spite of this, the concept is rather straight forward. We first iterate over the size of the output as we now have no padding. For each position we then check for which values of our indices should yield a padding value using a series of if statements. We then place the correct element in the tile and return this to the function.

3.2.3 MEC

Recall from Section 2.3 that MEC shares many similarities to Im2col. It was therefore chosen to take the same approach, by first allocating our lowered matrix and then choosing the correct values for convolution. The problem with the interpreted matrix for MEC however, is that we have to lower 3 columns at a time. This is in contrast to Im2col where we had to lower a 9 element tile at a time. In practice this means that we do not know how many element to lower at a time before execution. As a result, we had

to lower the 3 first elements of each row at a time, until we reached the bottom of the matrix as seen in Listing 3.5.

```
let interpretData [rows][cols]
                         (data: [rows][cols]f32): [][]f32 =
  let Tdata = (transpose data)
  let res =
    unsafe
    map (\i ->
      map (\j ->
            unsafe
           [if i == 0 || (j == 0 || j == (cols+1))
            then 0 else Tdata[i-1, j-1],
            if j == 0 || j == (cols+1)
            then 0 else Tdata[i, j-1],
            if i == (rows-1) || (j == 0 || j == (cols+1))
            then 0 else Tdata[i+1, j-1]
           ])
          (0...cols+1))
        (0...rows-1)
  in unflatten cols ((rows+2)*3) (flatten (flatten res))
```

Listing 3.5: MEC convolve partitions

When this was done we shift right by 3 elements and repeat. This approach turned out to be very inefficient due to poor spatial locality in memory. This also resulted in the dynamic padding used in the other programs, to be very inefficient and not worth while here. Because of this reasoning we opted for array slicing using pre-padding in spite of the problems covered earlier.

The second thing that differed from Im2col is which part of our interpreted data to multiply with our kernel:

```
let convolvePartitions [rows][cols]
                       (i_data: [rows][cols]f32)
                       (i_kernel: [9]f32): [][]f32 =
  let res =
    unsafe
    map (\row ->
      map (\col ->
             unsafe
             row[col]   * i_kernel[0] +
             row[col+1] * i_kernel[1] +
             row[col+2] * i_kernel[2] +
             row[col+3] * i_kernel[3] +
             row[col+4] * i_kernel[4] +
             row[col+5] * i_kernel[5] +
             row[col+6] * i_kernel[6] +
```

```
16                  row[col+7] * i_kernel[7] +
17                  row[col+8] * i_kernel[8])
18            (range 0 (cols-6) 3))
19          i_data
20    in (transpose res)
```

Listing 3.6: MEC convolve partitions

As seen from Listing 3.6 above, we unroll the same general computation as the case for Im2col. Recall that we have to multiply our kernel with each row of each of our partitions to construct our feature map. We therefore iterate over the lowered matrix extracting each row of 9 elements from each of the partitions. In an early implementation these partition were allocated in a third matrix and then passed to a separate function. This was quickly discarded due to the poor performance and memory usage this caused. And as seen from the program above integrating the partitions into the lowered matrix through indexing was rather simple as well.

A thing to note in this implementation is, that it is important that we iterate over our interpreted data in the outer loop. This is due to the fact that the Futhark compiler, for efficiency reasons, does not attempt to fuse operations together where we map over an array in the inner loop. We therefore moved i_{data} to the outer loop instead, and swapped the variables in our map.

Even with all this ing the inefficient way of allocating our lowered matrix really impacted performance a lot as touched upon later. The compiler has a hard time parallelizing the operations, as well as fusing allocations together with others to reduce peak memory usage. As a result, MEC suffers from poor memory efficiency, the one thing it aimed to improve. This makes us wonder whether or not our implementation is sufficient, and what other ways we can implement this algorithm.

3.2.4 Winograd

When implementing Winograd convolution, the first thing we needed to construct was the transformations. Recall from Chapter 2.4 that we transform our kernel and data in order to have fewer real multiplications during convolution. As also mentioned in this section, we are able to collapse the transformations into a series of additions and subtractions due to the nature of the transformation matrices. As a result, the first implementation was made by unrolling the dot products of the data, kernel and output transformations. This was to ensure that we had the arithmetic complexity reduction that the algorithm provides. However, this came at the cost of readability, as we now had an expanded and confusing function as seen from Listing 3.7:

```
1  let transformData (tile: [][]f32): [4][4]f32 =
2
3    let tile2 = [[tile[0,0]-tile[2,0], tile[0,1]-tile[2,1],
```

```
4                   tile[0,2]-tile[2,2], tile[0,3]-tile[2,3]],
5                 [tile[1,0]+tile[2,0], tile[1,1]+tile[2,1],
6                  tile[1,2]+tile[2,2], tile[1,3]+tile[2,3]],
7               [-tile[1,0]+tile[2,0], -tile[1,1]+tile[2,1],
8               -tile[1,2]+tile[2,2], -tile[1,3]+tile[2,3]],
9               [tile[1,0]-tile[3,0], tile[1,1]-tile[3,1],
10              tile[1,2]-tile[3,2], tile[1,3]-tile[3,3]]]
11
12    let tile3 = [[tile2[0,0]-tile2[0,2], tile2[1,0]-tile2[1,2],
13                  tile2[2,0]-tile2[2,2], tile2[3,0]-tile2[3,2]],
14                [ tile2[0,1]+tile2[0,2], tile2[1,1]+tile2[1,2],
15                  tile2[2,1]+tile2[2,2], tile2[3,1]+tile2[3,2]],
16              [-tile2[0,1]+tile2[0,2], -tile2[1,1]+tile2[1,2],
17              -tile2[2,1]+tile2[2,2], -tile2[3,1]+tile2[3,2]],
18              [ tile2[0,1]-tile2[0,3], tile2[1,1]-tile2[1,3],
19              tile2[2,1]-tile2[2,3], tile2[3,1]-tile2[3,3]]]
20
21      in (transpose tile3)
```

Listing 3.7: unrolled transform

As we started to improve the Winograd algorithm later in the implementation process, it was actually found that we achieved a greater speedup by simply using our matmul module to make the transformations. This was interesting as this approach does not skip the multiplications of ones and zeros.

```
1  let winogradConvolution [rows][cols]
2                          (tile: [rows][cols]f32)
3                          (t_kernel: [4][4]f32): [2][2]f32 =
4      let BT:[][]f32 = [[1.0,0.0,-1.0,0.0],[0.0,1.0,1.0,0.0],
5                        [0.0,-1.0,1.0,0.0],[0.0,1.0,0.0,-1.0]]
6      let AT:[][]f32 = [[1.0,1.0,1.0,0.0],[0.0,1.0,-1.0,-1.0]]
7
8      let t_data = matmul.matmul (matmul.matmul BT tile) (transpose BT)
9      let t_mult = pointwise t_data t_kernel
10     let res = matmul.matmul(matmul.matmul AT t_mult) (transpose AT)
11     in res
```

Listing 3.8: Winograd transformations

Listing 3.8 shows the current implementation of the data transformation. The reason why the former approach is not faster, is that on modern GPU's, efficiency depends more on how fast we can access memory and not on how fast we can do arithmetic. As there are hundreds of arithmetic units on the GPU, more often than not, the bottleneck is not the arithmetic throughput of the chip but rather the memory bandwidth of the chip[12, p.96]. It was therefore chosen to use regular matrix multiplication here as mentioned earlier. Another thing to note, is that the SOAC's that make up this multiplication operation has built-in loop tiling. This transforms a nested loop in such a way that we

can exploit spatial and temporal locality, as our data is now accessed in tiles. In practice, this loop tiling is actually less efficient for small dimensions of input as we are unable make enough use of the locality optimization. It is therefore a poor choice in the case of our algorithms. However, by going back to the unrolled implementation and avoiding this, we would instead lose a significant amount of parallelism.

The second important part of the Winograd implementation, was handling the output of the $F(2 \times 2, 3 \times 3)$ function as explained in Section 2.4. As we now had our transformations implemented, we needed a way to split our input data into the correct neighboring tiles, and perform our convolution:

```
let convolveTilesSecond [rows][cols]
                        (data: [rows][cols]f32) (t_kernel: [4][4]f32)
                        (h_tiles:i32) (v_tiles:i32) : [][]f32 =
  let res =
      map (\i ->
        flatten (transpose (map (\j ->
          unsafe
          (winogradConvolution
          ([[data[i,j], data[i,j+1], data[i,j+2], data[i,j+3]],
          [data[i+1,j], data[i+1,j+1], data[i+1,j+2], data[i+1,j+3]],
          [data[i+2,j], data[i+2,j+1], data[i+2,j+2], data[i+2,j+3]],
          [data[i+3,j], data[i+3,j+1], data[i+3,j+2], data[i+3,j+3]]])
          t_kernel))
          (range 0 (h_tiles*2) 2))))
        (range 0 (v_tiles*2) 2)
  in unflatten (rows-2) (cols-2) (flatten (flatten res))
```

Listing 3.9: Winograd convolution

Listing 3.9 shows how we, similar to the other algorithms, avoid slicing by writing out the desired tile indices. We then traverse each of the input's tiles passing these to our transform function, which returns the 2×2 output for that given input. This was problematic though, as we now had a 3 dimensional array, holding all the small output tiles. This meant that our tiles did not correspond to the correct sequence of values in our feature map. Fortunately by using `transpose` and `flatten`, we were able to make sure that each value was placed in the correct order in the resulting matrix. We then collapsed this matrix to a single dimension and used `unflatten` to split it into the correct columns.

This rather large amount of reshaping did come with a cost, especially for larger input sizes. That being said, it is something we might be able to improve on. By using the operations such that that the compiler can more easily fuse them together with other operations, we would be able to achieve even better performance.

3.3 Testing

Testing the correctness of our implementation is quite simple, as we can evaluate the output matrix of our convolution with library functions from other programming languages. For these tests it was chosen to validate our implementation using Scipy's Signal library from Python. Here we have a library function `convolve2D`[1] which works in the same way that our implementation does. We note here that we have to give the option "same", as this makes it return a result the same size as the input.

To ensure the validity of the tests, we used a Python library to generate the random data instead of Futhark. This was done to have the same foundation when testing our python and Futhark code. Although we cannot be certain this approach is unbiased, it was chosen due to the integrity of these large python libraries. The data input was created as 2D matrices consisting of the randomly generated float 32 values between 0 and 1. For the kernel we had to choose one with symmetric values and dimensions, as our implementation does not support other types. Our choice therefore fell on a common 3×3 kernel used for sharpening images.

Futhark uses a specific header in its data files that hold information about what the program can expect to read. This meant, we had to write some extra lines to our python generated data files. When this was done for both the input data and result, we were able to test these on our own implementation.

This was done using `futhark test` which is a test environment in Futhark. Here we can specify an input and output file and test if the implementation achieves the output using the given input. To simplify things all algorithms were imported into a single file, as seen from Appendix C.1, so that we could test the correctness of all programs using one call to `futhark test`.

3.4 Shortcomings

As we have seen from this chapter there are several shortcomings in our implementation, and in this section we aim to highlight the most prominent of these.

3.4.1 BLAS library

A significant lack in the current implementation is the fact that we cannot use the BLAS library in Futhark. As both Im2col and MEC rely on these operations to achieve efficiency on paper, not being able to use them is a problematic. We can interpret the BLAS operations as a competing strategy to our data parallel approach. Not being able to test the performance of this library therefore results in a less varied analysis of the

[1] `https://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.signal.convolve2d.html`

results regarding Im2col and MEC. This results in us not being able to determine how well our operations compare to those of the library, and ultimately pinpoint the best implementation for these two algorithms.
The way we would go about solving this is either directly implementing the handwritten Fortran code to Futhark, or using a dedicated porting compiler that would be able to translate this low level code.

3.4.2 Kernel size specific implementations

The extensive tuning of each of the algorithms has resulted in faster run times across each implementation. However, this came at the cost of making our implementation very specific. This is a problematic as our implementations only take one type of input. This imposes great limitations as we cannot use kernels of larger sizes which in some cases are more desirable to use. In addition to this, we might also want to use kernels that are not symmetric, to extract distinct features from our data. Both of these input however, are not an option in our current implementation, which is problematic as we might want to use one or a combinations of these two in a Convolutional network. Furthermore not being able to use these kernels also limits our implementation analysis as some of these might be more suitable for some of our algorithms. As a result, we could see even better run times for some of the algorithms if they were able to take larger non-symmetrical kernels.To solve this we would have to restructure our code, and make it more flexible. Furthermore, we would have to further analyse the compiler in order to write fast code without sacrificing efficiency.

3.4.3 Fusing operations

Only direct and Im2col convolution currently make optimal use of the compilers' ability to fuse operations together. This means that Winograd and MEC still have several compute kernels running throughout the program. Ideally we want to gather these as one single computation in order to achieve faster run times. This would require us to have an extensive knowledge regarding the Futhark compiler and the rules upon which it fuses operations together. With this knowledge we would be able to write tailor-made implementation that would take this project to greater heights.

3.5 Integration into neural network

The reason that a Convolutional Neural Network (CNN) is different from any other multi layer perception (MLP) is the fact that it has hidden layers which we call the *convolutional layers*. These act as the base layers for our CNN and is where all the convolution is computed. The convolutional layer would therefore be the ones in which our implementation would be used. This layer consists of three stages *convolutional stage*, *defector stage* and *pooling stage*[7]. The convolutional stage is where we split our image into its respective colour channels and then run our convolutional algorithms. In the defector stage, we then use an activation function on the outputted activation

maps, to determine whether a neuron should fire or not. In the final stage we have the pooling layer which controls overfitting by progressively reducing the spatial size of the network[1].
In practice we may have many convolutional layers each with individual filters. By extracting low-level features at early layers, we can gradually find more and more complex patterns as we pass the result from one layer to the next. This is the reason why we can use CNN's to recognise faces and animals which are made up of complex image patterns.

Chapter 4

Results

In this section we have benchmarked our current algorithms against each other to see the difference in performance. This was done both before and after tuning to see the effect of the changes that were made. The benchmarks were performed by passing in 3 sizes of input to our algorithms, being 512×512, 1024×1024 and 2048×2048 respectively. We examined the run time with `Futhark Bench` which uses `Futhar Dataset` to generate binary data files that our programs are then benchmarked on. To do this we ran `set INCREMENTAL_FLATTENING=1 && futhark bench -backend=opencl bench.fut`. The program used for this benchmark can be found at `https://github.com/Kyediis/Futhark_Convolution/tree/master/Benchmarks`.

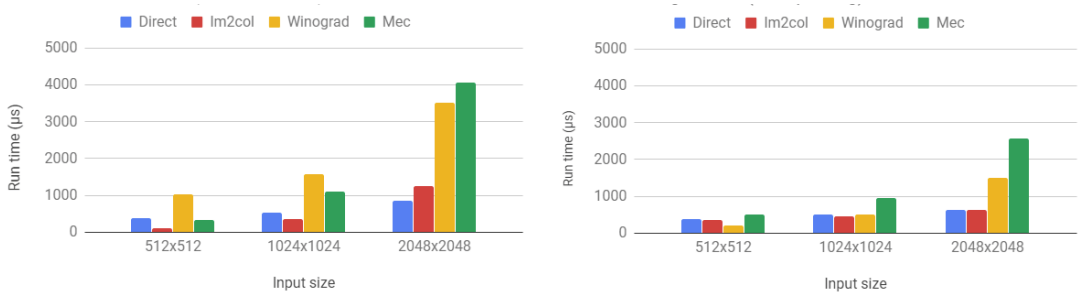
Another important thing to note here is that we set the environmental variable for incremental flattening before our test. This unlocks an unfinished optimizer in the Futhark compiler, that extends the current flattening algorithms for regular nested data parallelism[9]. This is a relatively new addition to the Futhark compiler and ensures that we get the best possible results that Futhark is currently able to produce. Finally, we also make sure to set the backend to opencl so that the programs are run on our GPU. For the CPU benchmarks we chose the library function from Section 3.3. We then used the same input sizes as passed into futhark test and timed the run time with Python's `timeit` library.

As `Futhark Bench` does not have a way to benchmark memory usage, we instead called each of our programs with the -D option. Here we had to use futhark dataset to manually create the three input sizes used in the other benchmarks. We were then able to pass these to each algorithm and analyze the peak memory usage.

The benchmarks were run on a Windows 10 machine using a NVIDIA GTX 1070 GPU with 8GB of memory, INTEL i5 4690K CPU and 16GB of DDR3 memory, with programs all using 32 bit floating points. The benchmarks listed in this chapter can be found in the appendix A.

4.1 Benchmarks

The first benchmark we see below in Figure 4.1 shows the difference in run time between our non-optimized and optimized algorithms.



(a) Before tuning (μs)

(b) After tuning (μs)

Figure 4.1: Benchmark results for algorithms

The results here show that we were able to almost halve the total run time of our algorithms for input sizes of 2048×2048. For the most part this is due to the dynamic padding, locality optimizations and unrolled operations discussed in Chapter 3. This choice did come with a trade-off, due to there being both pros and cons of unrolling a computation as described in Subsection 3.2.4. As we see here, the trade-off between low efficiency loop tiling and parallelism is equal around the 1024×1024 input size. This is why we see direct convolution and Im2col becoming slightly slower for the smaller inputs after our improvements. This is interesting as as we now see Winograd having the best run time for the 512×512 input size, further showcasing the potential that this algorithms has.

By benchmarking the convolutional function from Python we are able to see how well our best implementation measures up against one run on the CPU.

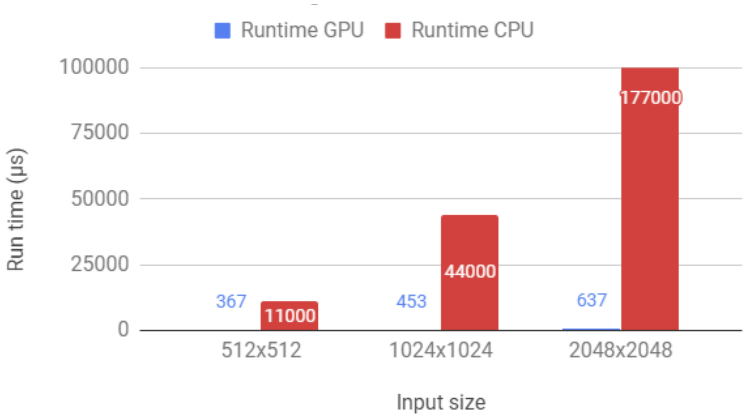


Figure 4.2: Benchmark results for GPU vs. CPU convolution

It is to be expected that the GPU implementation would be much faster, and the fundamental differences in GPU and CPU architecture makes it a unfar comparison. This being said, Figure 4.2 still gives us proof of precisely how much better a parallel approach is in contrast to a sequential one, when performing convolution.

Our final benchmark in Figure 4.3 shows how memory efficient each of our programs are.

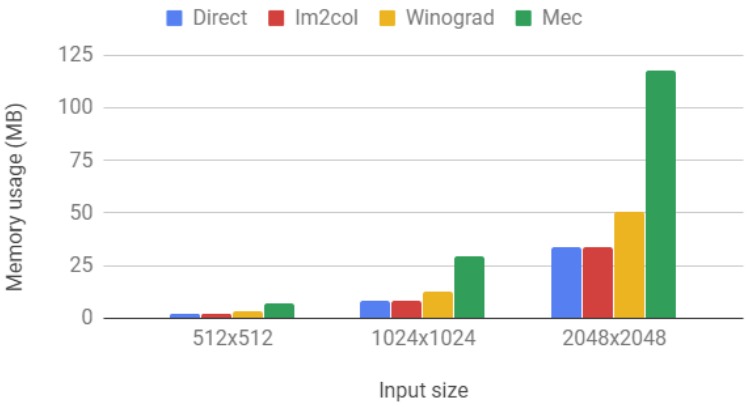


Figure 4.3: Benchmark results for peak memory usage

The interesting thing here is that direct convolution and Im2col use the exact same amount of memory. This is likely due to how the compiler is able to reuse allocated memory as it fuses operations together. This means that Im2col does not have to allocate the lowered matrix that direct convolution does not have. This same reason is why we see such a large efficiency gap when comparing an optimal implementation Im2col with the currently sub-optimal MEC.

4.2 Practical example

In a real world application, we would use our implementation we have provided an edge detection convolution in Appendix D. We first constructed a couple of python wrapper function which makes us able to load images as matrices and output them as as png files. We then wrote a Futhark program that imports one of our convolutional algorithms and uses this convolve the image with a kernel of our choosing. In this example we chose two convolutions with the sobel x and sobel y filters, that find vertical and horizontal lines respectively. By taking the gradiant magnitude of these two outputs we are able to get the resulting image seen in Appendix D. The programs used for this example can all be found in `https://github.com/Kyediis/Futhark_Convolution/tree/master/image_processing`[1]

4.3 Reflection

When reflecting on the result of our implementation, it is important to first compare the abstract theory from Chapter 2 with our concrete implementations from Chapter 3. By doing this, it becomes clear what a mathematical formulation defines as efficient, is rarely in correspondence with what the compiler does. We discovered that implementing mathematical optimal pseudo code directly into programs without taking the compiler into consideration, was the biggest mistake we made in this project. It left us with the most naive algorithm (direct convolution) as one of the best performing implementations. This is clearly proof that, just because an algorithm has mathematical benefits on paper, does not necessarily mean these come to life in code. This was the case with Winograd convolution, as the savings in multiplications cannot be expressed the same way on a GPU. Another example is MEC and the problematic way of lowering its input data, causing it to become less memory efficient. As a result, if a programmer was to implement computational algorithms in Futhark, focusing on understanding the compiler is paramount. The compiler is what gives Futhark programs such good performance. Therefore, being able to write tailor-made programs for it, is of higher priority than saving multiplications or finding new ways to allocate and traverse data-structures.

From a design point of view, we can also state that Futhark was viable for implementing these algorithms. It had many tools for handling the different array operations that we required, and could even use its compiler to make executing these incredibly fast. The only downside of this is the learning curve regarding the compiler in order to achieve this. However, the Futhark compiler is getting smarter for every day, and it is reasonable to expect that this learning curve becomes less steep as the compiler gets more intelligent.

When looking at our results, it becomes clear that they could be further optimized. By either porting the BLAS library or using cuBLAS[2] we might have been able to make

[1] To run the example use the batch file provide (Windows only)

[2] `https://docs.nvidia.com/cuda/cublas/index.html`

Im2col and MEC faster, due to the low-level optimizations that these offer. For Winograd we could experiment with how to lay out its neighboring tiles in memory and how we could most efficiently transform these. With this being said, we can still say that we have developed a set of fast algorithms for performing convolutional operations. By the relatively short amount of time we have worked on this project, we have been able to increase the performance of most of these algorithms by a large amount. We can therefore be confident that we can make these implementations even faster in the future.

Chapter 5

Conclusion

5.1 Future work

There are several important things that were left unimplemented in this project as described in Section . Arguably the most important of these is implementing a way to use the BLAS library, as it is the main reason why Im2col and MEC are lucrative in the first place. In doing this we could have determined if this approach was more suitable that Futhark's operations. This could have further increase the performance of our algorithms, as well as the CNN they would be integrated into[10].

The other two mentioned shortcomings both rely on extensive knowledge regarding the compiler. As we have explored in the project, we did not have sufficient knowledge to make the most of our implementations. Gaining more knowledge about the Futhark compiler is therefore also a relevant place to continue this project. This new found knowledge would be the foundation on which we would be able to further improve our implementations. It would not only make our algorithms faster, but also less conservative.

5.2 Evaluation

Throughout this thesis we have explored several algorithms that all performs convolution. We have implemented these in Futhark while documenting the design and implementation process. This showed us that Futhark is indeed capable of providing the abstraction needed in order to implement each algorithm. The flexibility of the language was also a main reason why we were able to improve our implementations as much as we did. It showed that faster and faster run times were achievable, given the necessary knowledge regarding the compiler. This is especially important looking forward, as we try to make our algorithms even faster, in spite of them already running in a matter of milliseconds.

Each of the algorithms came with pros and cons, which on paper dictated how well they would perform. Surprisingly enough these mathematical concepts did not come to

life the way we expected them to. After an in depth analysis, we came to the conclusion that the theoretical and practical implementations diverged from each other regarding what is most efficient. In turn, this meant that we got results that were not in accordance with our expectations. We found that this was mainly due to the lack in knowledge about the compiler and its abilities. As we begin to delve deeper into Futhark and its compiler, we expect to achieve even greater results looking forward.

5.3 Acknowledgements

Firstly, I would like to express my gratitude towards the University of Copenhagen for making this project possible. Secondly, I would also like to thank my supervisor Troels Henriksen for overseeing this project, and providing help in order to understand Futhark and the project itself. Finally, I am grateful to M. Tran for adding motivation to this project based on his own thesis [13], and answering questions related to his work.

Appendix A

Benchmarks

A.1 Futhark benchmarks results

A.1.1 Futhark programs before tuning

```
1  Results for bench.fut:direct:
2  dataset [2048][2048]f32 [3][3]f32:     845.50µs (avg. of 10 runs; RSD: 0.13)
3  dataset [1024][1024]f32 [3][3]f32:     535.70µs (avg. of 10 runs; RSD: 0.15)
4  dataset [512][512]f32 [3][3]f32:       377.10µs (avg. of 10 runs; RSD: 0.14)
5  Results for bench.fut:im2col:
6  dataset [2048][2048]f32 [3][3]f32:    1243.00µs (avg. of 10 runs; RSD: 0.02)
7  dataset [1024][1024]f32 [3][3]f32:     348.50µs (avg. of 10 runs; RSD: 0.02)
8  dataset [512][512]f32 [3][3]f32:       113.40µs (avg. of 10 runs; RSD: 0.07)
9  Results for bench.fut:winograd:
10 dataset [2048][2048]f32 [3][3]f32:    3511.80µs (avg. of 10 runs; RSD: 0.04)
11 dataset [1024][1024]f32 [3][3]f32:    1582.80µs (avg. of 10 runs; RSD: 0.09)
12 dataset [512][512]f32 [3][3]f32:      1019.40µs (avg. of 10 runs; RSD: 0.13)
13 Results for bench.fut:mec:
14 dataset [2048][2048]f32 [3][3]f32:    4072.10µs (avg. of 10 runs; RSD: 0.01)
15 dataset [1024][1024]f32 [3][3]f32:    1098.10µs (avg. of 10 runs; RSD: 0.02)
16 dataset [512][512]f32 [3][3]f32:       330.50µs (avg. of 10 runs; RSD: 0.04)
```

Listing A.1: Result of futhark test

A.1.2 Futhark opencl (GPU)

```
Results for bench.fut:direct:
dataset [2048][2048]f32 [3][3]f32:     637.60µs (avg. of 10 runs; RSD: 0.18)
dataset [1024][1024]f32 [3][3]f32:     514.20µs (avg. of 10 runs; RSD: 0.18)
dataset [512][512]f32 [3][3]f32:       373.50µs (avg. of 10 runs; RSD: 0.25)
Results for bench.fut:im2col:
dataset [2048][2048]f32 [3][3]f32:     637.20µs (avg. of 10 runs; RSD: 0.12)
dataset [1024][1024]f32 [3][3]f32:     453.20µs (avg. of 10 runs; RSD: 0.37)
dataset [512][512]f32 [3][3]f32:       367.10µs (avg. of 10 runs; RSD: 0.21)
Results for bench.fut:winograd:
dataset [2048][2048]f32 [3][3]f32:    1499.60µs (avg. of 10 runs; RSD: 0.02)
dataset [1024][1024]f32 [3][3]f32:     513.70µs (avg. of 10 runs; RSD: 0.09)
dataset [512][512]f32 [3][3]f32:       203.10µs (avg. of 10 runs; RSD: 0.11)
Results for bench.fut:mec:
dataset [2048][2048]f32 [3][3]f32:    2581.80µs (avg. of 10 runs; RSD: 0.09)
dataset [1024][1024]f32 [3][3]f32:     954.60µs (avg. of 10 runs; RSD: 0.10)
dataset [512][512]f32 [3][3]f32:       516.70µs (avg. of 10 runs; RSD: 0.19)
```

Listing A.2: Result of futhark test

A.1.3 Python (CPU)

```
512x512: 11138.7883905µs
1024x1024: 44449.5351914µs
2048x2048: 177550.039995µs
```

Listing A.3: Futhark benchmark

Appendix B

Programs

B.1 Interpret data no prior padding

```
let noPadInterpret [rows][cols]
                   (data: [rows][cols]f32): [][9]f32 =
  let res =
    unsafe
    map (\flat ->
        unsafe
        let i = flat / (cols)
        let j = flat % (cols)
        in [if (i != 0 && j != 0) then data[i-1,j-1] else 0,
            if (i != 0) then data[i-1,j] else 0,
            if (i != 0 && j != cols-1) then data[i-1,j+1] else 0,
            if (j != 0) then data[i,j-1] else 0,
            data[i, j],
            if (j != cols-1) then data[i,j+1] else 0,
            if (i != rows-1 && j != 0) then data[i+1,j-1] else 0,
            if (i != rows-1) then data[i+1,j] else 0,
            if (i != rows-1 && j != cols-1) then data[i+1,j+1] else 0
           ])
        (iota ((rows) * (cols)))
  in res
```

Listing B.1: Dynamic padding

Appendix C

Tests

C.1 Test of correctness

```
let transformData (tile: [][]f32): [4][4]f32 =
import "../Convolutional_algorithms/direct"
import "../Convolutional_algorithms/im2col"
import "../Convolutional_algorithms/mec"
import "../Convolutional_algorithms/winograd"

-- ==
-- entry: direct_test
-- compiled input @ pyarray.in
-- output @ pyresult.out

entry direct_test = direct.main

-- ==
-- entry: im2col_test
-- compiled input @ pyarray.in
-- output @ pyresult.out

entry im2col_test = im2col.main

-- ==
-- entry: mec_test
-- compiled input @ pyarray.in
-- output @ pyresult.out

entry mec_test = mec.main

-- ==
-- entry: winograd_test
-- compiled input @ pyarray.in
-- output @ pyresult.out

```

```
33 entry winograd_test = winograd.main
```

Listing C.1: Test of algorithm correctness

```
1 (0 failed, 0 passed, 1 to go).
2 Started testing correctness.fut (0 failed, 0 passed, 1 to go).
3 Finished testing correctness.fut (0 failed, 1 passed, 0 to go).
```

Listing C.2: Result of futhark test

Appendix D

Image convolution example

D.1 Edge detection

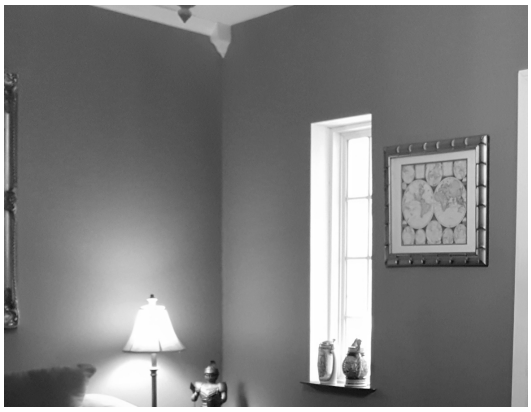


Figure D.1: Original image

Figure D.2: Edge detected image

Bibliography

[1] Basic overview of convolutional neural network (cnn). `https://medium.com/@udemeudofia01/basic-overview-of-convolutional-neural-network-cnn-4fcc7dbb4f17`. (Accessed on 06/02/2019).

[2] Convolutional neural networks (cnn): Step 1- convolution operation, Aug 2018.

[3] Going beyond full utilization: The inside scoop on nervana's winograd kernels, Dec 2018.

[4] Richard E. Blahut. *Fast Algorithms for Digital Signal Processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1985.

[5] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010.

[6] Minsik Cho and Daniel Brand. MEC: memory-efficient convolution for deep neural network. *CoRR*, abs/1706.06873, 2017.

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

[8] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. *SIGPLAN Not.*, 52(6):556–571, June 2017.

[9] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. Incremental flattening for nested data parallelism. pages 53–67, 02 2019.

[10] H. Kim, H. Nam, W. Jung, and J. Lee. Performance analysis of CNN frameworks for GPUs. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 55–64, April 2017.

[11] Andrew Lavin. Fast algorithms for convolutional neural networks. *CoRR*, abs/1509.09308, 2015.

[12] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[13] Duc Minh Tran. Implementation of a deep learning library in futhark, 2018.

[14] S. Winograd, Society for Industrial, Applied Mathematics, Conference Board of the Mathematical Sciences, and National Science Foundation (Estats Units d'Amèrica). *Arithmetic Complexity of Computations*. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1980.