Ingebrigt Kristoffer Thomassen Hovind
Erling Sung Sletta

# Parallelization of Local Learning Rules

Bachelor’s thesis in Computer Science
Supervisor: Ole Christian Eidheim
May 2022

Bachelor’s thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Ingebrigt Kristoffer Thomassen Hovind
Erling Sung Sletta

# Parallelization of Local Learning Rules

Bachelor’s thesis in Computer Science
Supervisor: Ole Christian Eidheim
May 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

# Abstract

This thesis is a piece of further work based on an article written by Ole Christian Eidheim. There, he describes "a novel unsupervised learning rule based on Gaussian functions that can perform online clustering without needing to specify the number of clusters prior to training" [6]. The specific focus has been to improve the computational speed of the Python algorithm released together with Eidheim's article, by porting it to C++ and parallelizing it on both CPU and GPU.

We have developed five different, well-optimized implementations[1]. The first is a standard C++ implementation using only one thread, the second and third implementations are then extensions of the first, both utilizing several threads but with different strategies. The final two implementations utilize the third party libraries ArrayFire and Boost Compute in order to GPU-accelerate the algorithm.

We have gathered data on the execution time of the various algorithms on two different systems, giving valuable information on exactly what strategies pay off, and how much one can expect to benefit from parallelism. We have found that with the right implementation running on the right system, GPU-acceleration can be faster than ordinary CPU implementations, even when running only 16 calculation loops in parallel. For less optimal implementations, the GPU-accelerated programs scale better than most of the CPU implementations, and are able to achieve a faster execution time on our hardware setups when utilizing 169 calculation loops.

[1]The source code for all our implementations, along with information about necessary dependencies for execution, can be found at https://github.com/ikhovind/Parallelization-of-Local-Learning-Rules

# Sammendrag

Denne bacheloroppgaven er et arbeid basert på en artikkel skrevet av Ole Christian Eidheim, som beskriver en "novel unsupervised learning rule based on Gaussian functions that can perform online clustering without needing to specify the number of clusters prior to training" [6]. Mer spesifikt så handler oppgaven om å oversette en Pythonalgoritme utgitt sammen med Eidheims artikkel til C++, og så parallellisere den på både CPU og GPU.

Vi har produsert fem forskjellige, godt optimaliserte implementasjoner [2]. Den første er en standard C++ implementasjon som kun bruker én tråd. Den andre og tredje er utvidede versjoner av den første, og bruker flere tråder men med forskjellige strategier. De siste to implementasjonene bruker to forskjellige tredjepartsbibliotek, ArrayFire og Boost Compute, for å parallellisere algoritmen på GPU.

Vi har samlet data på flere forskjellige plattformer om hvordan forskjellige implementasjoner av samme algoritme, som gir oss verdifull informasjon om nøyaktig hvilke strategier det er som lønner seg og hvor mye man kan forvente å tjene på parallellisering. Vi har funnet at med riktig implementasjon på riktig system, så kan GPU-parallellisering kjøre raskere enn CPU-versjoner, selv når man bare kjører 16 parallelle løkker. Også med mindre optimale implementasjoner så skalerer de GPU-parallelliserte implementasjonene som oftest bedre enn de fleste CPU-implementasjoner, og oppnår lik eller raskere kjøretid med 169 parallelle utregninger på våre maskiner.

[2]Kildekoden til alle våre implementasjoner og samt oversikt over avhengigheter nødvendig for kjøring finnes i vårt GitHub-repo: https://github.com/ikhovind/Parallelization-of-Local-Learning-Rules

## Preface

This bachelor thesis was written by two computer science students at The Norwegian University of Science and Technology in the spring of 2022, on behalf of supervisor and client, Ole Christian Eidheim; an associate professor at the department of computer science. The thesis is an extension to a machine learning article written by Eidheim [6], and further explores the viability of an algorithm he had developed and written in Python. The groups first assignment was therefore to implement these algorithms in a faster, lower level language, specifically C++, and later to utilize GPU-parallelization to further improve performance at scale.

The task was picked due to the exciting prospect of working with GPU-parallelization, which neither member had any experience with. The team also wanted to work in a lower level language such as C++, and wanted more experience with optimization.

The process itself has been successful, both from the perspective of the tasks and that of personal growth. This thesis has involved working with many fields that the group was not familiar with beforehand, which has proven to be challenging. There were also many external aspects that provided the group with additional challenges throughout the process, for example the installation of external dependencies, or the finer parts of a CMake script. We are overall proud of the work done in the process of writing this thesis, and hope that it may be useful to others in the future.

We would like to thank our supervisor Ole Christian Eidheim for both an interesting task and good assistance in times of need. We would also like to thank the developers of every open source library that this thesis depends on, such as ArrayFire [28], Boost-Compute [17] and matplotlib-cpp [8].

Ingebrigt Hovind

Ingebrigt Kristoffer Thomassen Hovind
Trondheim, 20.05.2022

[illegible]

Erling Sung Sletta
Trondheim, 20.05.2022

## Task

The task was originally given in Norwegian, but has been translated by the group into:

### Purpose of task:

The goal of this task is to explore local learning rules as an alternative to end-to-end optimization of machine learning models through back-propagation, where the adjustment of the weighting of each layer is only dependent on the input to the layer and the layer itself. These are learning rules that are more biologically plausible, but such algorithms are not well supported in frameworks such as PyTorch.

### Short description of task:

Local learning rules have the potential to achieve larger machine learning models than the ones that exists today. One of the reasons for this is that models trained with local learning rules can be trained layer by layer, and the training can then be done more efficiently, for example through parallelization, than end-to-end optimization with back-propagation.

Examples of one or more tasks one can focus on in this project:

- Parallelization of general calculations on GPUs such that one can parallelize for example local learning rules in a simpler way.

- Parallelization of specific calculations done in local learning rules. In some cases these algorithms can be parallelized better to the detriment of accuracy, and it is necessary to examine if the loss of accuracy if acceptable.

- Comparison of models trained with local learning rules with models trained with back-propagation.

### Project focus:

After a startup meeting, the group was assigned the first task listed above as the point of focus for the thesis. The goals of the project were then decided to be to measure performance of different parallelization methods, and to find the crossover point where the GPU implementations became faster than the CPU implementations.

# Contents

## List of Figures

## List of Tables

## Acronyms

**CPU** Central Processing Unit. i, ii, iv, viii, 3, 4, 8, 10, 11, 15, 16, 18, 19, 29, 30, 33, 35

**CUDA** Compute Unified Device Architecture. vii, 8, 10, 11, 21, 29–31, 35, 36

**GPU** Graphics Processing Unit. i–iv, viii, 1, 3, 4, 8, 10–17, 29, 30, 32, 33, 35, 36

**OpenCL** Open Computing Language. vii, 8, 10, 11, 16, 19, 21, 30, 31, 36

## Glossary

**Batch** A collection of patches that are to be processed by the algorithm. 15, 30

**Cluster** *See* neuron . 12

**Context switching** The act of storing the state of a thread and suspending its execution to allow another thread to run. 3, 29

**Device** In the context of GPU acceleration; device refers to the GPU and it's memory. 4, 11, 16, 31

**Filter** A pattern found by the algorithm; able to represent a specific part of the input, for example the edge of a line when using handwritten numbers. viii, 11, 12, 15, 17, 30

**GPU-acceleration** The act of taking an algorithm normally ran only on the CPU and adapting it to run on the GPU, which will lead to a great acceleration when ran at scale. i, 13–16, 19, 30, 33–35

**Host** In the context of GPU acceleration; host refers to the CPU and it's memory. 4, 11, 16, 31

**Neuron** A maximum number of neurons is set ahead of training; each neuron may or may not have an attractive term towards a given sample, and thus may or may not find a filter during training. vii, viii, 12, 13, 15, 17, 19–21, 23, 30, 31, 35

**OpenMP** Open Multi-Processing, an API that provides parallel programming support for environments with a shared memory. 8

**Operator overloading** Writing methods that allow you to use pre-defined operators, such as +, -, *, and so on, for custom data structures. 19

**Patch** Used by the team to refer to a part of an image; a 5x5 patch is a 5 pixel wide, 5 pixel long sample from one of the images in the dataset.. vii, viii, 15, 17, 19–23, 31

**PyTorch** A popular open source Python framework used for machine learning purposes. iv, 1

**Sample** *See* patch; more specifically a patch that is to be processed by the algorithm. viii, 30, 31

# 1 Introduction

As an alternative to end-to-end optimization of machine learning models through backpropagation, local learning rules allow the adjustment of the weighting of each layer to be only dependent on the input to the layer and the layer itself. Such models are more biologically plausible, but they are not well supported in existing frameworks, such as PyTorch.

This bachelor's thesis was written as an extension to an article discussing the viability of local learning rules, written by our supervisor. The purpose is to examine the computational performance of an implementation of an algorithm used in local learning rules. This process was divided into three phases; to translate the algorithm from Python to C++, to parallelize the algorithm, and then finally to analyze the results. The questions we has tried to answer with this piece of work are; What effect can one expect from parallelizing a general purpose algorithm, and at what scale of calculations does the increased overhead of the GPU become insignificant?

## 1.1 Revisiting Gaussian Neurons for Online Clustering with Unknown Number of Clusters

> Despite the recent success of artificial neural networks, more biologically plausible learning methods may be needed to resolve the weaknesses of backpropagation trained models such as catastrophic forgetting and adversarial attacks. A novel local learning rule is presented that performs online clustering with a maximum limit of the number of cluster to be found rather than a fixed cluster count.[6]

In these first couple sentences of the previously mentioned article written by our supervisor, Eidheim describes a need for alternative training algorithms due to a number of weaknesses to those that are currently in use. More specifically, he mentions models trained backpropagation, and states that they have been getting more popular in the last decade. As an alternative, he presents a couple methods that be believes to be more biologically plausible.

Biological plausibility and some weaknesses of backpropagation networks will be explained in a later section, but for now the main takeaway is that the novel learning rule and algorithms presented in the article also allow for a higher degree of parallelization. This was mentioned as one of the possible opportunities for investigation for further work in Eidheim's article, which is what created the basis for this thesis.

## 1.2 Structure

The structure of this thesis is:

**Section 1** - Gives an introduction to the thesis by describing the origin of the problem as well as actualizing what the group is trying to solve.

**Section 2** - Gives a theoretical background not only for the work done in this thesis, but also why it is important. In addition it presents the statistical formulas used for the later analyses.

**Section 3** - Presents the technologies used by the group throughout the project. This involves both technical choices made to support the group administratively, and choices made that directly affected the different implementations.

**Section 4** - Presents how the group worked to develop the implementations that were necessary. It describes how the different versions were optimized, as well as how data was gathered in order to draw the necessary conclusions.

**Section 5** - Presents the results gathered, both scientific and those related to the project itself.

**Section 6** - Discusses the results gathered and the decisions that the team has made throughout the process, in addition to the potential societal impact of parallelism and efficient computing.

**Section 7** - States the team's final conclusions regarding the problems described in section 1 based on the data gathered, and potential future work aspects.

# 2 Theory

## 2.1 Background

Before getting into the finer details, some context to the problems at hand is useful to have. This chapter will serve to explain some of the key concepts that were used over the course of the project.

### 2.1.1 Moore's law

Moore's law was a prediction made by Intel co-founder Gordon Moore in 1965, which states that transistor counts in computer chips would double every two years [7]. Moore's law has been valid for many years, and this trend has meant that major increases in clock speeds could be expected in relatively short periods of time. However, this trend has recently been stagnating, as technology is reaching the physical limits of the size of transistors. Real world delays in transistor technology, such as when Intel delayed the release of their 10-nanometer technology in 2015, have indisputable implications for users of computer hardware, who cannot expect the same kind of increase in clock speed that they once did [18].

Instead of increasing the number of cores on a chip, one possibility would be to speed up the existing ones. However, clock speeds have largely hit a wall, as heat generation has reached critical levels. This problem with heat generation is in large part the reason that the max clock speed of computer chips has been essentially capped since 2004 [26].

### 2.1.2 Parallelization

While, in theory, four cores working at 250 MHz are computationally equivalent to one core working at 1 GHz, this is often not the case in practice. The most essential problem is that many algorithms are not perfectly parallelizable, that is to say that all parts of the program cannot be run at the same time, while having the same outcome as a sequential implementation. This is a large reason for why the speedup gained when working with multi-threaded programs are not linear in relation to the number of threads used. Even if the program is possible to parallelize, the work load for each thread will likely not be exactly equal, meaning that the computational power is not used as efficiently as possible.

There are also several other factors that play a large part in the diminishing returns of several cores. For example, when working with several threads, any operating system running the program will have to perform more context switching, as there will be more threads competing for the same computing power. This is a source of overhead which only grows along with increased parallelism.

Spreading executions over several cores can also divide memory into several caches. This means that if the program is parallelized in a way such that each thread does not work in a continuous memory region, then each thread will likely experience a higher amount of cache misses than a single-threaded program would.

Despite all the downsides, parallel computing is now a necessity, not an option, and due to the slowdown of Moore's law, it is only becoming more relevant. There are many strategies when it comes to CPU-parallelization, but the most important ones involve keeping threads as independent as possible, and keeping their work loads relatively equal. The specific strategies implemented in this thesis is discussed further in section 4.

### 2.1.3 GPU-Acceleration

GPUs are computer hardware commonly used to render images that will be displayed on a screen [20]. In order to do this most efficiently, their structure is highly parallel, which allow them to process large amounts of data at the same time. In addition to being used to render images, GPUs can also be used for general purpose programming. Although individual operations are slower, the increased parallelism allows for greater speed on tasks that can be ran in parallel.

The structure of a GPU means that all the problems from working with parallelism mentioned in the previous subsection are also relevant when working with GPUs. In addition, GPUs face some computational limitations, as the cores are simpler than the those on a CPU. For example, some older GPUs do not support double precision floating point numbers, and even on modern GPUs, operations using these can take up to 32 times as long to perform as those using single precision [3].

An additional bottleneck faced by GPUs is the transfer of data between the host and the device. This means that for small calculations, the transfer speed to the GPU would outweigh the computational benefits gained.

In return for these downsides, a modern GPU allows for running thousands of calculations in parallel. This is the reason GPUs are frequently used in fields such as machine learning, as the ability to run numerous calculations at the same time more than outweighs the downsides and lower clock speeds. Any algorithm, especially ones that feature many independent general calculations, will run faster on the GPU when ran at scale.

### 2.1.4 Training Algorithms

Since machine learning is not the main aspect of this thesis, these topics will not be explained entirely in-depth, but some context could be nice to have to understand the importance and motivation behind the project. This section will provide a surface level context, while the cited articles can be referred to for more comprehensive explanations.

In his article, Eidheim mentions that models trained with backpropagation can be found to have several weaknesses, and that more biologically plausible methods may be needed to circumvent them [6]. This term, biological plausibility, is often brought up in machine learning context due to the fact that Artificial Neural Networks are designed and called as such, because they are meant to mimic the neurons one can find in a biological brain. Neuroscientists, however, often criticise existing algorithms due to their inconsistency with current day knowledge of neurobiology [2]. This biological implausibility has spawned many attempts and research projects to explore alternative algorithms, including the article that this thesis extends upon.

The weaknesses mentioned in Eidheim's article are catastrophic forgetting and adversarial attacks. The former is also known as catastrophic interference, and refers to instances where machine learning models spontaneously and completely forgets all previously learned information during training, in favour of new information. Roger Ratcliff mentions another issue with sequential learning, specifically that "discrimination between studied items and new items either decreases or is nonmonotonic as a function of learning" [21], and addresses several attempts at solving the problem, without success.

Adversarial attacks are instances where a person with malicious intent makes use of known weaknesses or information about a machine learning model to launch attacks on it. An example could be supplying inputs that have been tampered with, for example a modified street sign, to a classifier model in an attempt to deceive it. These attacks are not necessarily visible to the human eye, as we tend to filter out tiny bits of noise that could prove disastrous for machine learning models. A research paper conducted by Szegedy *et al.* uncovered that this property was not a random result; a model trained on a different dataset would misclassify the same input [25].

### 2.1.5 Regression

Regression is a method used within statistics in order to characterize the relationship between one or more explanatory variables, and one response variable. The two forms of regression relevant to this thesis are linear regression, where a straight line is fitted to best estimate the observed data, and polynomial regression, where a polynomial to the $n^{th}$ degree is best fitted to observed data.

For the initial linear regression, two assumptions will have to be made about the data in order for the regression line to be accurate. The first is that that the two variables are linearly dependent. This means that the formula for the estimated value can be written like so:

$$E(Y|X = x) = \beta_0 + \beta_1 x \tag{1}$$

Where $E(Y|X = x)$ is the estimated $Y$ at a given $x$ value, $\beta_0$ is the crossover point of the x-axis and $\beta_1$ is the gradient.

The second assumption made for a linear model is that the value of Y, is normally distributed with the same standard deviation $\sigma$, for any value of x. This means that for the model to be accurate, the standard deviation of the response variable cannot vary.

The first assumption, the linear dependence, can be shown by calculating the covariance of the data pairs. A covariance not equal to zero signals that the variables are linearly dependent. The value of the covariance is hard to interpret, the correlation coefficient can therefore be calculated in order to give a number that is easier to grasp.

In order to find the correlation coefficient, the covariance has to be found first. If x is an explanatory variable and y is responsive, the covariance can be estimated by:

$$\hat{Cov}(X, Y) = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})$$

Where $x_i$ and $y_i$ are pairs of data, $\bar{x}$ and $\bar{y}$ are the average values, and $n$ is the total number of observed pairs.

Before the correlation coefficient can be found, the standard deviation of each variable needs to be found:

$$\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2}$$

The correlation coefficient is then found by dividing the estimate on the factor of the standard deviations:

$$R = \frac{\hat{Cov}(X, Y)}{s_x \cdot s_y}$$

This is a number between -1 and 1, where -1 indicates a strong negative correlation, 1 indicates a strong positive correlation, and 0 indicates no linear correlation.

Even data with high correlation coefficients does not need to be perfectly linear, and can often be slightly polynomial, meaning a better equation for the estimated value would be:

$$E(Y|X=x) = \beta_0 + \beta_1 x + \beta_2 x^2 \tag{2}$$

Therefore, a representation of the modelled polynomial can be written as:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \epsilon \tag{3}$$

[19]

Where $\epsilon$ is the error at the given point, that is, the distance away from the chosen regression model.

If we let $x_2 = x^2$, the formula can be rewritten as:

$$y = \beta_0 + \beta_1 x + \beta_2 x_2 + \epsilon$$

And then be solved as a normal linear regression with two variables. This is most easily done in matrix form:

$$\mathbf{y} = \mathbf{X}\beta + \epsilon$$

Where:

$$y = \begin{bmatrix} y_1 \\ \vdots \\ n \end{bmatrix}, X = \begin{bmatrix} 1 & x_{1,1} & x_{2,1} \\ 1 & x_{1,2} & x_{2,2} \\ \vdots & \vdots & \vdots \\ 1 & x_{1,n} & x_{2,n} \end{bmatrix}, \beta = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_n \end{bmatrix}, \epsilon = \begin{bmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

[19]

In order to find the curve of least squares, we want to find the $\beta$ values that minimize the following expression:

$$S(\beta) = \sum_{i=1}^{n} \epsilon^2 = \epsilon^T \epsilon = (y - X\beta)^T (y - X\beta)$$

[19]

Multiplying out the parenthesis gives:

$$y^T y - y^T X\beta - X^T \beta^T y + X^T \beta^T X\beta$$

Because $\beta^T$ and $X^T y$ are a row and column vector respectively, $\beta^T X^t y$ is a scalar. It is therefore equal to its own transpose $y^T X\beta$ and they can be added together:

$$S(\beta) = y^T y - 2\beta^T X^T y + X^T \beta^T X\beta$$

Critical points can be found by setting the derivative equal to zero:

$$\begin{aligned}
\Rightarrow \quad & \left.\frac{\partial S}{\partial \beta}\right|_{\hat{\beta}} = -2X^T y + 2X^T X\hat{\beta} = 0 \\
\Rightarrow \quad & X^T X\hat{\beta} = 2X^T y \\
\Rightarrow \quad & \hat{\beta} = (X^T X)^{-1} 2X^T y
\end{aligned} \tag{4}$$

[19]

Solving (4) gives the coefficients of the least square polynomial (3). These calculations have been implemented in the least_squares.py, hosted alongside the rest of the source code.

## 2.2 Related Work

This section will discuss and describe some articles concerning topics that are relevant to our thesis. This is to give some background on our foundation, and give some additional perspectives on the issues at hand.

### 2.2.1 GPU Acceleration of 3D Agent-Based Biological Simulations

Much like our thesis, this article talks about taking steps to optimize and GPU-accelerate an already existing algorithm [14]. The steps taken in this article can be generalized to most use cases. Among the ones most relevant ones, is being open to adapt the algorithm in order to parallelize it better. This does not only consist of making sections of the algorithm independent of each other, but also making sure that as many memory reads as possible are sequential and shared between different threads, thereby maximizing the amount of useful data kept in cache. Memory optimizations such as this has been especially useful for the group's own progress, as this has been the source of the largest speedups.

### 2.2.2 A Parallel Algorithm of Image Mean Filtering Based on OpenCL

"With the increasing amount of image data, the single-processor or multiprocessor computing equipment has been unable to meet the requirements of real-time data processing" [27]. In a research project conducted by H. Xiao *et al., (2021)*, an attempt was made to utilize OpenCL to parallelize their weighted mean filtering algorithm. As the earlier citation might suggest, their goal was to study the performance of their parallel algorithm compared to the CPU-based serialized one, much like what is being done in this thesis.

The article also covers the usage multiple different technologies aside from OpenCL, like OpenMP and CUDA. The latter two were used as baselines along with the serialized CPU version; the OpenCL algorithms reached speedups ranging from 5.00 to 20.88 times faster depending on the image size and platform used. The paper concludes that it's algorithm has shown itself to be effective, and that the results "solves the problem of hardware dependence of CUDA parallel acceleration" [27] among other things.

# 3 Technology

Several pieces of technology have been utilized to assist the team throughout the process of writing this thesis. Some have only been used to organize the work-flow, while others are part of the developed implementations themselves. This section will describe the technological choices that were made throughout the process and the reason for them, while the relative success of these choices are discussed in section 6.

## 3.1 Administrative

The team has used various administrative tools in order to make the project easier to manage, while minimizing the amount of time spent on non-development tasks. These have varied from technologies chosen in order to make the development itself easier, to those used for mandatory parts of the project, such as time tracking and report writing.

### 3.1.1 Non-development technology

The most important piece of technology that has been used that is not directly connected to the development process itself is Discord. Discord is an online messenger application that has played a large part in organizing the team and it's structure. The team has created a custom server, which made it possible to sort communications by topic, allowing previous solutions to be easily available when the similar problems arose later on.

Another important piece of technology has been Toggl. Toggl is time tracking software that the team has used in previous projects in order to easily and accurately track the amount of time spent in different areas of the thesis work. Toggl was chosen because the team already had experience with using it, the free version of the software was adequate for our needs, and because it is very easy to manage time entries, both to create new ones and edit any existing.

Last but not least, the final important piece of non-development technology has been Overleaf. Overleaf is an online LaTeX editor, which the team has used to write the final report as well as several other mandatory documents. Overleaf was chosen because both team members has had good experiences with it in previous projects, and it allowed the team to simultaneously write on the same sections without facing versioning errors, which would have been a problem if the writing had been handled locally.

### 3.1.2 Directly assisting technology

This section will talk about technologies used in order to assist the development directly. The project is not inherently reliant on these technologies, as it is with for example C++, but they have still been important for the productivity of the team as a whole.

Firstly, the IDE of choice for both team members has been CLion, which is developed by JetBrains. CLion was chosen because both team members have had good experiences with the program previously, and there are several key features which have been beneficial to the teams productivity. For example, Code With Me, which has allowed the team to have live, collaborative programming sessions while still being able to work on their desktop computers.

The versioning software that has been used is Git, and the online repositories have been hosted on GitHub. Both technologies were chosen due to the team's previous experience. The link to the associated GitHub repository can be found in the Abstract.

## 3.2 Testing Environment

The testing environments used during development has been the group's personal computers. For the sake of consistency, all benchmarking have been done on the same two platforms. Both platforms are desktop computers belonging to the team members, and the technical specifications of each one can be seen below.

| Specification \ Platform | Platform 1 | Platform 2 |
|---|---|---|
| CPU | AMD Ryzen 3 3200G 3.6GHz (4 cores) | Intel Core i7 7700k 4.2GHz (4 cores) |
| # of threads | 4 threads | 8 threads |
| System memory | 16.0 GB | 32.0 GB |
| GPU | AMD Radeon RX 580 | NVIDIA GeForce RTX 2070 |
| GPU cores | 2304 | 2304 |
| GPU core frequency | 1257 MHZ | 1410 MHz |
| Video memory | 8 GB GDDR5 | 8 GB GDDR6 |
| Video memory bandwidth | 256 GB/s | 448 GB/s |
| Video memory bit width | 256 bits | 256 bits |
| Single-precision floating-point computing capacity (FP32) | 6.2 TFLOP/s | 7.48 TFLOP/s |
| Double-precision (FP64) | 385.9 GFLOP/s | 233.3 GFLOP/s |

**Table 1:** Technical specifications of the two benchmarking platforms

Due to CUDA only being able to be run on NVIDIA GPUs, platform 1 has only been able to test CPU implementations as well as the OpenCL-based GPU implementations.

The algorithm has been developed on various Linux platforms, but the majority of the development process and all benchmarking has been conducted on Manjaro. This distribution was chosen due to the ease of installation for required drivers and other dependencies that were necessary for the libraries used.

## 3.3 Python

The Python implementation was developed by Ole Christian Eidheim in order to conduct preliminary testing on his algorithm, described in the previously mentioned article [6]. The algorithm itself has not been changed by the group and all benchmarking has been performed in its original state. It is written in Python 3.

## 3.4 C++

The C++ version of the algorithm was developed to be an optimization of the Python algorithm. The C++ version that was chosen was C++20. This was chosen since Manjaro is rolling-release, it is easy to find support for the newest C++ versions. Working with modern C++ would also give valuable experience, while giving access to useful tools, such as std::filesystem, which was introduced in C++17.

### 3.4.1 Third party libraries

The only third party library that was used before development on the GPU-implementations begun was matplotlib-cpp [8]. This is a C++-wrapper for the popular Python library matplotlib, which is used to create and visualize data. In this project, it was used for plotting the filters, which naturally made the C++-version dependent on Python, albeit only for this visualization of data.

**GPU libraries**
In order to GPU-accelerate an existing program, interoperability between the GPU and CPU is needed. A GPU library provides such interoperability, while lowering the required effort to program for the GPU. The two libraries which were used in this thesis were ArrayFire [28] and Boost Compute [17].

**ArrayFire**
ArrayFire is a high level library for parallelism, allowing the user to choose whether to use a CPU-, OpenCL- or CUDA-backend. This allows the user of the library to write a general parallelized application and then pick the backend that performs the best in each case. The group initially chose to use ArrayFire based on advice from the project supervisor. Additionally, the ability to develop a singular application that could test both OpenCL- and CUDA back-ends was quite enticing.

ArrayFire works by providing a single data structure, the ArrayFire array. Manipulation of data in this array can be vectorized using overloaded operators or member functions of the array itself. ArrayFire also provides several separate functions to parallelize otherwise sequential code snippets such as for-loops.

**Boost Compute**
Boost compute is a part of the the Boost organization's libraries [4]. It is a parallel computing library that provides a thin wrapper over OpenCL, which simplifies the compilation of a kernel, and the interactions between the host and the device, while still allowing the use of custom OpenCL kernels.

Boost Compute was chosen as a second library after the team had gained experience with ArrayFire, but before a functional version had been completed with it. The team desired a way to write code in a more conventional way, which would make it easier to port the algorithm. Boost Compute was chosen over using OpenCL directly, due to its seemingly good documentation and combination of high- and low-level functionality when needed. Additionally, the team had previous experiences with other boost C++ libraries, which influenced the decision. Working with Boost Compute gave the team greater freedom by allowing them to write a custom kernel. Since it was easier to port the algorithm to the custom flavor of C99, which is used in OpenCL, the team was able to create a working version in Boost Compute in less time than it took with ArrayFire.

# 4 Method

The goal of the project has been to gather data on several different implementations of the same algorithm. The algorithm itself is used to find a series of filters that can be, for instance, used as edge detectors for classifiers. The images that have been used for testing throughout this thesis is the MNIST dataset [5], and the clusters shown in later examples are all found through this dataset. An in-depth explanation of how the filters are found and the math behind them can be found in Eidheim's article [6]. Attempts were also made to apply the algorithms on the CIFAR-10 dataset [15], however time constraints led the group to discontinue the project.

The process of developing these implementations and gathering the data itself will be discussed in the following sections.

**Figure 1:** An example of some handwritten numbers from the MNIST dataset

Source: Josef Steppan [24]

## 4.1 Research method

The method that will be used in this thesis in order to draw conclusions based on the data found will be a quantitative analysis of the response variable execution time and the explanatory variable maximum number of neurons.

In order to ensure the validity of the data gathered, all timings were to be done using built-in software clocks. This ensured that the timers in question would record only the relevant parts of the program. It also made it so that any timer data could be made independent of human interaction. This allowed the team to write a script that would automatically perform the necessary testing when the computers were not needed for other purposes. The software clocks were also helpful since the team wanted to make a program that could be used with other datasets that would require varying amounts of I/O, although for consistencies sake only the MNIST dataset has been used for benchmarking.

The original plan was to gather data on only one of the team members' computers, but a decision was later made to test on two separate platforms. This would allow the team to see if the trends present on one platform was present on the other, allowing them to draw conclusions based on a larger set of results. The platforms used also consist of one computer with AMD branded parts and one with Intel/NVIDIA parts. This means that the conclusions drawn will be more independent of the underlying hardware technology. The specifications of each computer can be found in section 3.2, Table 1.

The decision to gather data on the team's own computers was made despite the fact that the team had been offered to use a dedicated GPU-server, if they thought it necessary. The personal computers were chosen because the group wanted the data to be as generalizable as possible. It would be a big bonus if this thesis was able to draw conclusions on the run time of general software ran on normal computers, instead of only on high performance systems. By using personal computers instead of a GPU-server, the group limited the amount of memory and GPU cores that could be used, which made the data gathered more representative, as most people only have access to consumer-grade GPUs.

Through a regression analysis based on the relationship between the maximum number of neurons and the execution time, the group has attempted to create a model that can be used to compare the scaling execution time of the different implementations.

## 4.2 Process

Early on, the team divided the project into a set number of tasks. For planning purposes, these tasks were then assigned a time frame. A Gantt chart was drawn to visualize the process, where each day represents roughly six hours of work per person. This means that the estimated amount of work before the initial C++ implementation was functional, was estimated to be $22 * 6 * 2 = 264$ hours at most.

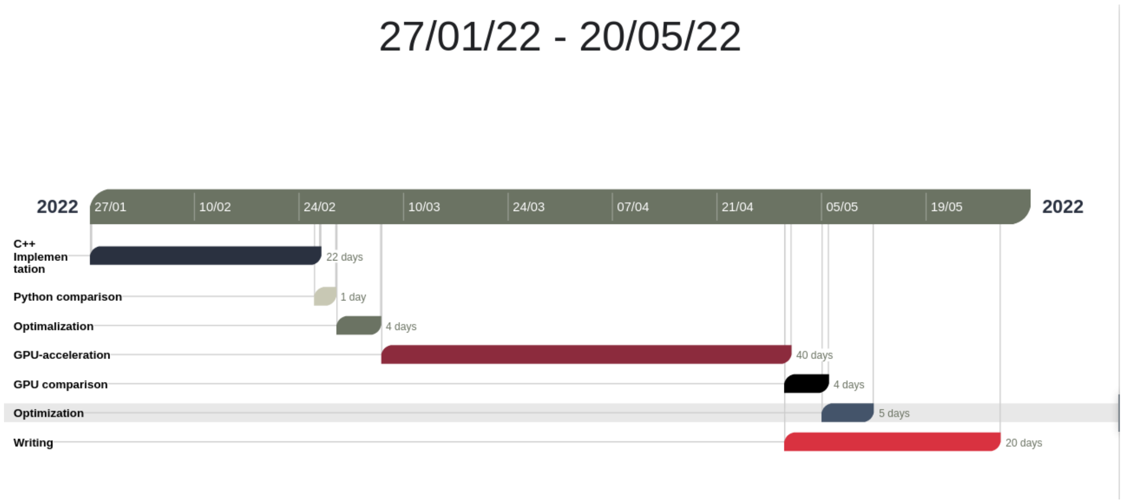


**Figure 2:** Gantt chart from the "Project Handbook" attachment

1. **C++ Implementation (27/01 - 26/02)**
   The first period was focused on getting a deeper understanding of the inner workings of the Python algorithm, as well as starting the translation to C++.

2. **Python comparison (26/02 - 28/02)**
   The next period was to be used to compare the working C++ implementation with the Python algorithm. In reality, the C++ implementation was tested extensively during development, and no dedicated day to compare ended up being needed.

3. **Optimization (01/03 - 06/03)**
   When the C++ implementation was found to be functional, the team then moved onto optimizing said implementation. In reality, the optimization of the single threaded program took around ten days, but these ten days also involved different developments happening simultaneously.

4. **GPU-acceleration (07/03 - 30/04)**
   This period was the most important and therefore had the longest dedicated time-frame. After a meeting with the team's supervisor, it was decided that it would be better for the project as a whole, in addition to the groups own learning, to develop multi-threaded C++

implementations before starting the GPU-acceleration. The first ten days in this period were therefore spent developing two different implementations of multi-threading, extending upon the original implementation.

5. **GPU-Comparison (30/04 - 05/05)**
During the development of the GPU implementations, the team conducted extensive testing in the same way that was done when developing the first implementation; this means that this period was largely superfluous and was therefore omitted as a dedicated period.

6. **Optimization (05/05 - 11/05)**
As the previous period was omitted, this period was shifted up by around one week. Here the group focused on optimizing the different GPU-versions and achieved around a 3x speedup on the fastest one. The period originally planned for this process was instead used for the final formal data gathering.

7. **Writing (30/04 - 20 /05)**
This period was allocated to primarily focus on the project thesis, while still allowing other tasks to be done in parallel. In reality this happened along with the final developments and data gathering.

As this is a research thesis, the results derived from the developed implementations are more important than the developed product itself. This means that there is no ”final release”, and the period spent working on the program was more dynamic than that of a normal systems development task. The team therefore knew from the beginning that the planned dates for finishing each section were advisory, not absolute. The team also knew that there were likely changes that had to be made to older versions a ways out into the project, but the goal was to be done with large scale changes within the assigned time frames.

### 4.2.1 Roles

The team consists of two good friends with large amounts of experience working together from previous group projects. This means that the group already had good communication, and no time needed to be spent on team building. Both members in this group are self starters, able to organize work alone or in unison with an equal, and therefore chose not to appoint a leader. This is also the reason for there being no defined roles at the outset, apart from choosing the member responsible of the documents related to this thesis as well a secretary.

### 4.2.2 Development

The team decided early on that strict system development methodology, such as scrum, would be a hindrance to the naturally exploratory manner in which this research thesis would be written. The team did however bring along experience from earlier scrum projects, and wanted to preserve as many positive aspects as possible.

The aspects that the team wanted to preserve the most from scrum methodology, was the open form of communication. This was especially important when the development of the algorithm took place on the team’s desktop computers, as this meant that the team members would not be sitting in the same room. Through frequent voice calls and text channels on a dedicated server, the team was able to keep each other updated on personal progress. Frequent meetings also took place online, facilitating for example pair programming sessions.

The development tasks themselves were handed out on a first-come-first-serve basis. Any larger decisions were discussed verbally, and both team members were easily able to keep track of what the other person was doing due to the group’s communication style; avoiding any instances of overlapping work. As the team members already had experience working together, they also felt comfortable enough to ask about anything they might have felt unsure about.

## 4.3 Optimization strategies

All but one of the major strategies used for optimization on the single-threaded C++ version were derived from the book *Optimized C++: Proven Techniques for Heightened Performance* by Kurt Guntheroth [12]. The first applied strategy was to focus only on the "hot" statements of the program. When optimizing a program within a limited time frame, it is important to focus on the code that is actually affecting the execution time. There are several ways to find out what sections of a program are "hot"; the strategies used by the group has been to examine the content of loops, and thereby find out which parts of the program are performed most frequently. A secondary more sophisticated strategy has been to use a profiler. A profiler is a tool that allows you to see how much of a given programs execution time was spent in a given portion of code. This allows you to see what function calls are the most intensive. Combining these two strategies allowed the group to focus on the parts of the program with the most speedup potential.

Another strategy mentioned by Guntheroth is to reduce memory allocation and copying. This strategy has been implemented by transferring large data structures as references instead of by value. This means that instead of copying the entire structure, the memory address of the existing structure is sent instead. The group has also strived to move large data structures from within functions, to member variables, thereby the program does not have to allocate new space for the variable each time the function is called, and can overwrite the existing values instead.

The only major strategy not described in the book was to implement all multi-dimensional C++ vectors as linear one-dimensional vectors, so that a vector with the dimensions 2x3 would be represented by a single vector with six elements. This was the single change that lead to the greatest speedup for the single threaded version of the program, and led to a 3.3x speedup. The significant speedup can be explained by several factors; the matrices are not sparse, which means that every single column would need be allocated a new vector, and that the columns in the vectors are all uniform so that they all have the same number of elements. These two factors together mean that the same number of actual elements are allocated with a one-dimensional solution as a two-dimensional, but for the one-dimensional vector only one vector has to be allocated.

### 4.3.1 Multi-threading strategies

The strategies used for the later parallelizations were divided into two main categories. The first involved taking each batch of patches, dividing them over the available threads and running these segments in parallel. Once the threads have finished their respective segments of the batch, the average value of all computations are calculated and assigned to the neurons. While this method was found to be generally faster, this increase in performance would come at a cost of accuracy. This loss of accuracy is discussed in subsection 6.2.

The second strategy was based around the way filters are found in a model's neurons. A number of samples are processed a given number of times, and in each of these calculations, the neurons are given an updated value based on previous neuron values. This means that they can all be run in parallel, and forms the basis of the second parallelization strategy used, where the new values for each neuron were calculated and assigned in parallel, and everything else is run sequentially.

The GPU-accelerated implementations were exclusively based on the second mentioned strategy. This is because a large part of the thesis is to find out how many neurons have to be present in parallel before the GPU, which has greater overhead but greater parallelization potential due to it's large number of cores, is faster than the ordinary CPU implementations.

### 4.3.2 GPU-strategies

The strategies used for the GPU-accelerated version were first focused on optimizing the transfer of memory objects from the host to the device. For example, as you cannot generate random numbers on the OpenCL device, the 3 million or more random numbers that are needed by the program are generated at program start and transferred in its entirety, instead of continuously throughout the program. This was done in order to limit the number of transfers, which would be significantly higher if the random numbers were generated when needed, as in the non-GPU-accelerated implementations.

There was also significant speedups, roughly 2x, found by adjusting the memory type of the OpenCL-kernel parameters from global to private, constant or local. This allows the OpenCL-device to store the memory closer to each core, as read-only memory and local memory can use a lower level cache without worrying about other threads. Local and private memory is used for data that is created individually on each compute object. They can therefore not be used for transferring data, but are useful for performing the computations themselves. The difference in execution time between using global and constant memory was minor, while transforming data into private variables led to a large speedup. This is despite of the fact that the variable transformed from global to constant was the same size as, and accessed an equal number of times as the local variable.

The final strategy used to optimize the Boost Compute version, was to set as many OpenCL-kernel variables as possible to compile time constants. The group found that when setting the loop variables as parameters, the execution time of the entire program was significantly slower when compared to using raw integers. With the help of online forum posts, the group theorized that this was due to the compiler being unable to unroll the loops when they were limited by dynamic variables. By defining all the variables as constants in the compile time options of the kernel, the compiler was able to unroll the loop, and the execution time became equivalent to using values directly.

The ArrayFire iterations were developed in a somewhat different manner, as the library's simplification of "the process of software development for the parallel architectures found in CPUs, GPUs, and other hardware acceleration devices" [28] came at the cost of some freedom. Large portions of time were spent trying to map out the intricacies of the library, by for example figuring out when host-device transfers were taking place. Optimization strategies were then mostly based around looking through relevant documentation and discussions to find methods with correct functionality, as these were meant to be optimized already.

## 5 Results

Two different benchmarks have been performed on each implementation of the algorithm; one using 5x5 pixel patches, and the other 9x9. The given parameters used and their effects in each test can be found in Eidheims article, where the algorithm is originally described [6].

For the 5x5 benchmark, $10^6$ samples were used, which is the same amount as in the original article. The 9x9 benchmarks, however, used $10^7$ samples in the original article, but only $10^6$ were used for the benchmarks made in this thesis. This is due to the limitations of the hardware used for testing, as well as the fact that this thesis is aimed primarily towards performance and GPU acceleration. Since the filters found at the end of each experiment were not as relevant as relative execution time, the group did not deem it necessary to run the longer experiments. Due to further time constraints within the project, some of the testing was also not ran at full scale, instead processing $10^5$ samples and scaling up to $10^6$. This was deemed to be necessary on the Python when utilizing above 25 neurons, as well as the Boost Compute implementation on system 2 above 81 neurons.

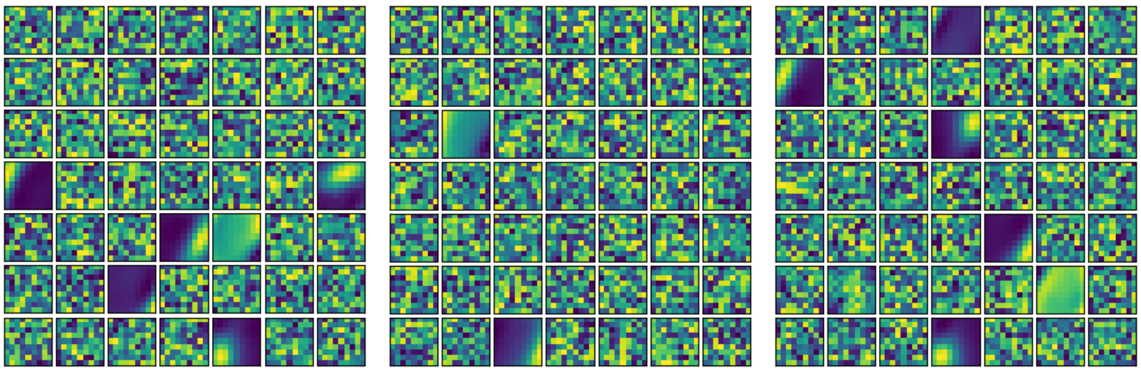

**(a)** Single-Threaded **(b)** Parallelizing batches **(c)** Parallelizing neurons

**Figure 3:** Examples of filters with C++ implementations processing $10^6$ 9x9 patches

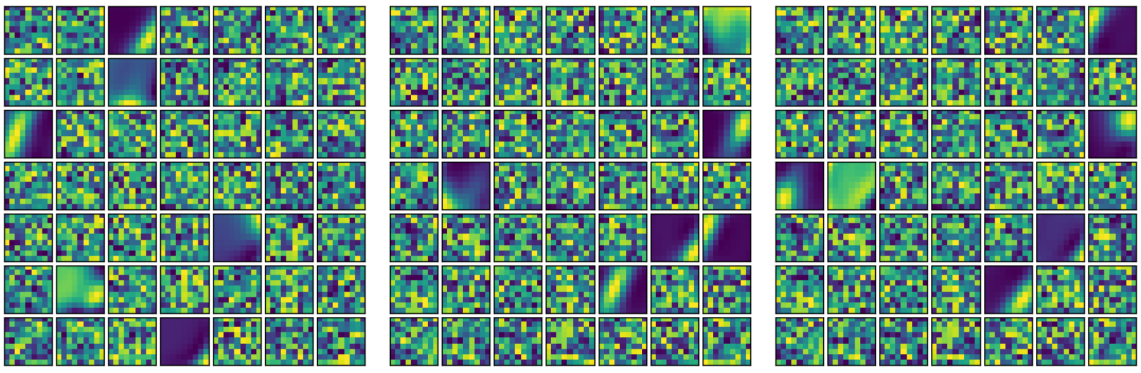

**(a)** Boost.Compute **(b)** ArrayFire using OpenCL **(c)** ArrayFire using CUDA

**Figure 4:** Examples of filters found with GPU implementations processing $10^6$ 9x9 patches

## 5.1 Engineering

### 5.1.1 Functional demands

As this task was a research task, there was only one functional goal created at the outset. In order to be able to finish this thesis, the group would have to be able to come to a sound conclusion based on correct data. For the data to be correct, the implemented algorithms need to perform the same calculations as the original Python script, i.e. they need to be mathematically correct. If an implementation is not mathematically correct, the results of the calculations may be similar, but it would be impossible to know to what degree mathematical error affects the execution time. For the sake of data integrity, the group has therefore spent a great deal of effort ensuring that each script performs the correct calculations.

One of the ways the correctness has been controlled throughout the process has been by comparing the results produced by each algorithm in each implementation with the results from Eidheim's orignal article [6]. Here, every implementation used to draw a final conclusion has been shown to produce similar results as the ones presented, when given the same parameters. Every implementation has also gone through thorough code review, and any error found in one implementation has also been automatically fixed in others, if the error had propagated outwards.

### 5.1.2 Non-functional demands

Non-functional demands are not absolutely necessary for finishing the project, but are required in order to improve the quality of the conclusions drawn. The only such demand set for this thesis was therefore to have a well-optimized program, formalized as having a faster execution speed than the Python implementation. Having a well optimized program means that the analysis of the execution time will be more accurate. Trends could still be seen with major optimizations lacking, but it would be more difficult to draw a conclusion, as it would be uncertain whether the difference in execution time stems from one implementation being better optimized, or from the strategy and technology used. Fast execution times have been a priority from the start, and early optimizations have therefore propagated outwards to all implementations.

## 5.2 Administrative results

In the initial progress plan found in section B.2 of the attached project handbook, three result goals were laid out; to achieve a good end result that satisfies all of the task's and client's requirements, to leave no large or obvious improvement potentials in the final iteration of the programs, and to write good, concise documentation for the work process, result and code that is used over the course of the project.

In this thesis, Ole Christian Eidheim has taken the role as a client, and the team has achieved the requirements set to a good degree, confirmed in the frequent meetings throughout the process, as noted in the meeting minutes. For example, the requirement to develop a multi-threaded version on the CPU, specified in meeting number 3, as seen in section C.3 of the project handbook.

The final demand of good and concise documentation has also been achieved well. Through a thorough README, references to the original article, and comments within the code describing the general structure. It would be entirely possible for someone else to take over the project and continue working on it, or implement the algorithm within their own codebase, with some minor modifications.

In addition to the result goals, team also set out to improve their own skills as software developers. The team has gained experience with large group projects and the requirements around them, and how good and bad decisions propagate outwards. For example, the data structure initially used to simply implement operator overloading, became very essential to every CPU implementation. The team has also learned how to best gather scientific data, and has learned that more data points with less well-tested data can be more valuable than fewer data points that are more thoroughly tested, because of the inherent time limits in the project. The team has also realized that a completely flat organizational structure works well in many contexts, as the entire project has been completed without a leader.

Another goal for the team was to learn new technologies. The team had previous experience with both C++ and multi-threading, but had not completed any large scale project utilizing either one. The team has also learned to work with previously unfamiliar technologies, such as ArrayFire, Boost Compute and OpenCL. This will allow the team to identify situations in the future where GPU-acceleration will be a valuable tool, as well as the experience to implement such a solution.

### 5.2.1 Time spent

As mentioned previously, the time spent on this thesis has been tracked using Toggl. The total time spent on this project amounts to roughly 970 hours. A total number of 482 hours was spent developing the implementations used to gather data, 307 has been spent writing the thesis and the remaining hours was used for either administrative work such as meetings, or registering and analyzing the data gathered.

This project has been worked on from around the end of January to the end of May. Both team members had to attend a mandatory 10 ECTS course from the start of the semester until the beginning of March, but for all remaining parts of the semester, this project has been the sole focus.

## 5.3 Scientific results

### 5.3.1 Python

The initial Python version was developed by Ole Christian Eidheim, and tested on the computers of both team members. The technical specifications of each platform can be seen in subsection 3.2.

| Platform 1 | | |
|---|---|---|
| Neurons | 5x5 patches | 9x9 patches |
| 16 | 2 650 565 ms | 2 737 724 ms |
| 25 | 6 437 066 ms | 6 648 232 ms |
| 36 | 13 312 537 ms | 13 760 701 ms |
| 49 | 24 677 038 ms | 25 349 946 ms |

| Platform 2 | | |
|---|---|---|
| Neurons | 5x5 patches | 9x9 patches |
| 16 | 1 813 033 ms | 1 818 632 ms |
| 25 | 4 298 988 ms | 4 247 180 ms |
| 36 | 9 278 680 ms | 9 011 840 ms |
| 49 | 16 934 697 ms | 16 244 405 ms |

**Table 2:** Performance with Python

### 5.3.2 C++

#### Single-threaded

| Platform 1 | | |
|---|---|---|
| Neurons | 5x5 patches | 9x9 patches |
| 16 | 76 651 ms | 215 115 ms |
| 25 | 184 007 ms | 515 915 ms |
| 36 | 379 806 ms | 1 076 480 ms |
| 49 | 701 879 ms | 1 979 643 ms |
| 64 | 1 196 984 ms | 3 421 655 ms |
| 81 | 1 911 744 ms | 5 546 995 ms |
| 100 | 2 915 642 ms | 8 570 675 ms |
| 121 | 4 269 327 ms | 12 359 130 ms |
| 144 | 6 039 914 ms | 17 611 875 ms |
| 169 | 8 325 103 ms | 22 823 320 ms |

| Platform 2 | | |
|---|---|---|
| Neurons | 5x5 patches | 9x9 patches |
| 16 | 69 732 ms | 172 754 ms |
| 25 | 168 173 ms | 417 821 ms |
| 36 | 346 383 ms | 867 953 ms |
| 49 | 639 332 ms | 1 605 309 ms |
| 64 | 1 091 759 ms | 2 770 353 ms |
| 81 | 1 746 511 ms | 4 388 921 ms |
| 100 | 2 667 910 ms | 6 663 123 ms |
| 121 | 3 917 530 ms | 9 804 227 ms |
| 144 | 5 515 671 ms | 13 815 898 ms |
| 169 | 7 623 748 ms | 19 274 632 ms |

**Table 3:** Performance with single-threaded C++

#### Multi-threaded

| Platform 1 | | |
|---|---|---|
| Neurons | 5x5 patches | 9x9 patches |
| 16 | 25 180 ms | 62 559 ms |
| 25 | 57 471 ms | 141 925 ms |
| 36 | 113 423 ms | 291 733 ms |
| 49 | 206 278 ms | 508 043 ms |
| 64 | 348 228 ms | 892 133 ms |
| 81 | 552 554 ms | 1 431 909 ms |
| 100 | 841 410 ms | 2 180 864 ms |
| 121 | 1 223 487 ms | 3 175 432 ms |
| 144 | 1 729 883 ms | 4 496 862 ms |
| 169 | 2 384 139 ms | 6 760 592 ms |

| Platform 2 | | |
|---|---|---|
| Neurons | 5x5 patches | 9x9 patches |
| 16 | 24 050 ms | 55 263 ms |
| 25 | 51 471 ms | 130 059 ms |
| 36 | 105 107 ms | 255 054 ms |
| 49 | 193 190 ms | 458 927 ms |
| 64 | 320 283 ms | 739 380 ms |
| 81 | 497 736 ms | 1 173 892 ms |
| 100 | 747 988 ms | 1 747 473 ms |
| 121 | 1 066 183 ms | 2 508 401 ms |
| 144 | 1 488 654 ms | 3 544 592 ms |
| 169 | 2 036 425 ms | 4 892 153 ms |

**Table 4:** Performance when parallelizing batches

| Platform 1 | | |
|---|---|---|
| Neurons | 5x5 patches | 9x9 patches |
| 16 | 70 860 ms | 118 674 ms |
| 25 | 99 401 ms | 398 325 ms |
| 36 | 239 056 ms | 722 396 ms |
| 49 | 515 039 ms | 1 349 758 ms |
| 64 | 808 441 ms | 2 176 564 ms |
| 81 | 1 323 283 ms | 2 331 682 ms |
| 100 | 1 912 450 ms | 3 523 758 ms |
| 121 | 3 026 058 ms | 5 298 635 ms |
| 144 | 4 005 134 ms | 7 317 840 ms |
| 169 | 6 432 302 ms | 10 441 780 ms |

| Platform 2 | | |
|---|---|---|
| Neurons | 5x5 patches | 9x9 patches |
| 16 | 71 287 ms | 204 858 ms |
| 25 | 94 561 ms | 235 603 ms |
| 36 | 145 548 ms | 371 029 ms |
| 49 | 220 615 ms | 579 572 ms |
| 64 | 344 565 ms | 885 321 ms |
| 81 | 509 307 ms | 1 340 472 ms |
| 100 | 753 693 ms | 1 932 619 ms |
| 121 | 1 077 206 ms | 2 789 002 ms |
| 144 | 1 499 428 ms | 3 808 864 ms |
| 169 | 2 052 956 ms | 5 213 884 ms |

**Table 5:** Performance when parallelizing neurons

### 5.3.3 GPU

#### Compute

| Platform 1 | | |
|---|---|---|
| Neurons | 5x5 patches | 9x9 patches |
| 16 | 63 144 ms | 158 079 ms |
| 25 | 93 223 ms | 241 602 ms |
| 36 | 132 484 ms | 343 296 ms |
| 49 | 178 314 ms | 463 905 ms |
| 64 | 253 346 ms | 617 874 ms |
| 81 | 323 028 ms | 1 516 084 ms |
| 100 | 366 560 ms | 1 868 778 ms |
| 121 | 440 841 ms | 2 266 042 ms |
| 144 | 501 394 ms | 2 746 786 ms |
| 169 | 1 116 573 ms | 4 899 879 ms |

| Platform 2 | | |
|---|---|---|
| Neurons | 5x5 patches | 9x9 patches |
| 16 | 1 045 121 ms | 4 708 489 ms |
| 25 | 1 632 165 ms | 7 340 942 ms |
| 36 | 2 335 674 ms | 10 559 880 ms |
| 49 | 3 532 411 ms | 28 059 068 ms |
| 64 | 4 593 578 ms | 36 651 985 ms |
| 81 | 7 377 023 ms | 62 969 273 ms |
| 100 | 9 102 675 ms | 86 030 968 ms |
| 121 | 14 288 221 ms | 138 462 441 ms |
| 144 | 17 019 378 ms | 164 892 804 ms |
| 169 | 25 624 691 ms | 242 081 747 ms |

**Table 6:** Performance using Boost.Compute

#### ArrayFire

| Platform 1 | | |
|---|---|---|
| Neurons | 5x5 patches | 9x9 patches |
| 16 | 931 571 ms | 892 201 ms |
| 25 | 1 364 428 ms | 1 226 402 ms |
| 36 | 2 049 285 ms | 1 949 413 ms |
| 49 | 2 572 142 ms | 2 257 010 ms |
| 64 | 3 293 285 ms | 2 994 205 ms |
| 81 | 3 673 857 ms | 3 685 627 ms |
| 100 | 4 463 857 ms | 4 590 808 ms |
| 121 | 5 336 285 ms | 5 498 480 ms |
| 144 | 6 499 142 ms | 6 604 830 ms |
| 169 | 7 451 666 ms | 7 484 011 ms |

| Platform 2 | | |
|---|---|---|
| Neurons | 5x5 patches | 9x9 patches |
| 16 | 1 822 547 ms | 1 727 863 ms |
| 25 | 2 318 574 ms | 2 222 246 ms |
| 36 | 3 400 078 ms | 3 370 134 ms |
| 49 | 4 362 561 ms | 4 271 132 ms |
| 64 | 5 919 551 ms | 5 782 362 ms |
| 81 | 7 148 837 ms | 7 127 715 ms |
| 100 | 8 609 692 ms | 8 534 563 ms |
| 121 | 10 447 142 ms | 10 385 684 ms |
| 144 | 12 482 989 ms | 12 215 096 ms |
| 169 | 14 200 633 ms | 14 274 554 ms |

**Table 7:** Performance using ArrayFire with OpenCL

| Platform 2 | | |
|---|---|---|
| Neurons | 5x5 patches | 9x9 patches |
| 16 | 968 598 ms | 983 132 ms |
| 25 | 1 225 078 ms | 1 244 612 ms |
| 36 | 1 869 148 ms | 1 845 334 ms |
| 49 | 2 291 062 ms | 2 287 017 ms |
| 64 | 3 197 064 ms | 3 189 014 ms |
| 81 | 3 913 858 ms | 3 901 281 ms |
| 100 | 4 655 803 ms | 4 639 993 ms |
| 121 | 5 686 032 ms | 5 548 128 ms |
| 144 | 6 586 284 ms | 6 624 989 ms |
| 169 | 7 697 583 ms | 7 867 086 ms |

**Table 8:** Performance using ArrayFire with CUDA

### 5.3.4 Comparison

H. Xiao *et al.* included a table containing the "acceleration ratio" of each platform's performance on the different computing platforms in their research paper [27]. This gave a simpler and clearer overview than comparing each iteration's runtime in ms did, which inspired this thesis to include similar tables. These acceleration ratios represent each iteration's speedup, relative to the optimized C++ version represented in Table 3. 1.08 means an 8% increase in performance, while 0.93 means a 7% decrease. Using the columns listed under Platform 2; these represent the relative performance of the implementations represented by tables 5, 4, 7, 6, and 8, respectively.

| | Platform 1 | | | | Platform 2 | | | | |
|---|---|---|---|---|---|---|---|---|---|
| # of filters | Filter | Batch | OpenCL | Boost | Filter | Batch | OpenCL | Boost | CUDA |
| 16 | 1.08 | 3.04 | 0.08 | 1.21 | 0.98 | 2.90 | 0.04 | 0.07 | 0.07 |
| 25 | 1.85 | 3.20 | 0.13 | 1.97 | 1.78 | 3.27 | 0.07 | 0.10 | 0.14 |
| 36 | 1.59 | 3.35 | 0.19 | 2.87 | 2.38 | 3.30 | 0.10 | 0.15 | 0.19 |
| 49 | 1.36 | 3.40 | 0.27 | 3.94 | 2.90 | 3.31 | 0.15 | 0.18 | 0.28 |
| 64 | 1.48 | 3.44 | 0.36 | 4.72 | 3.17 | 3.41 | 0.18 | 0.24 | 0.34 |
| 81 | 1.44 | 3.46 | 0.52 | 5.92 | 3.43 | 3.51 | 0.24 | 0.24 | 0.45 |
| 100 | 1.52 | 3.47 | 0.65 | 7.95 | 3.54 | 3.57 | 0.31 | 0.29 | 0.57 |
| 121 | 1.41 | 3.49 | 0.80 | 9.68 | 3.64 | 3.67 | 0.37 | 0.27 | 0.69 |
| 144 | 1.51 | 3.49 | 0.93 | 12.05 | 3.68 | 3.71 | 0.44 | 0.32 | 0.84 |
| 169 | 1.29 | 3.49 | 1.12 | 7.46 | 3.71 | 3.74 | 0.54 | 0.30 | 0.97 |

**Table 9:** Performance comparison for 5x5 patches

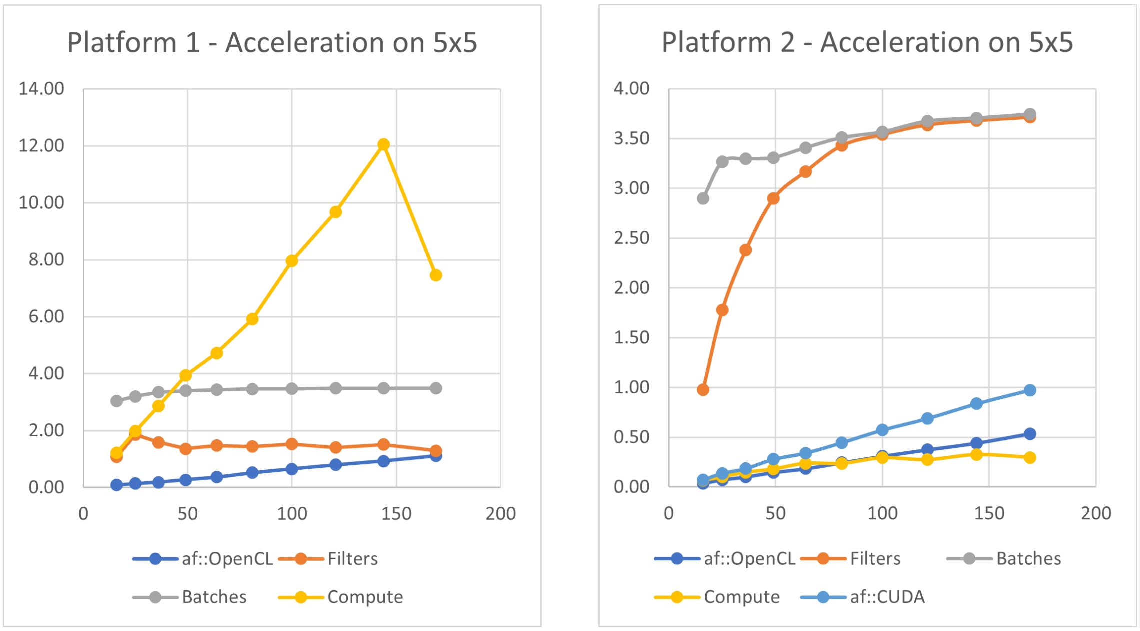
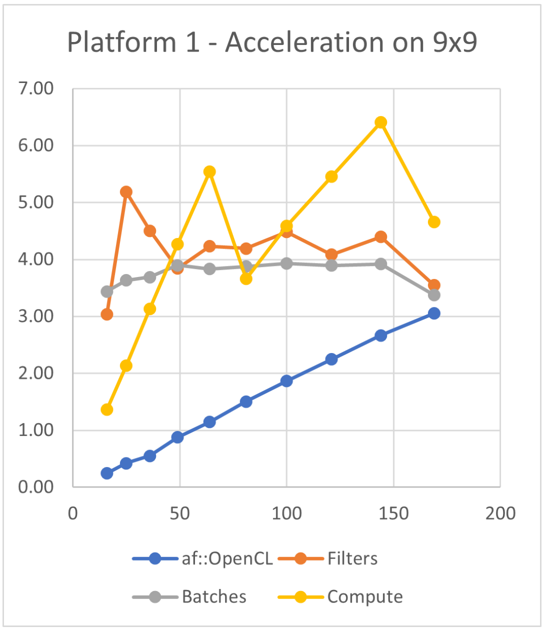


**Figure 5:** Graphical representation of numbers from Table 9

| # of filters | Platform 1 Filter | Batch | OpenCL | Boost | Platform 2 Filter | Batch | OpenCL | Boost | CUDA |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 3.04 | 3.44 | 0.24 | 1.36 | 0.43 | 2.68 | 0.04 | 0.07 | 0.07 |
| 25 | 5.19 | 3.64 | 0.42 | 2.14 | 1.03 | 3.18 | 0.08 | 0.10 | 0.14 |
| 36 | 4.50 | 3.69 | 0.55 | 3.14 | 1.55 | 3.22 | 0.10 | 0.15 | 0.19 |
| 49 | 3.84 | 3.90 | 0.88 | 4.27 | 2.06 | 3.20 | 0.15 | 0.18 | 0.28 |
| 64 | 4.23 | 3.84 | 1.14 | 5.54 | 2.46 | 3.37 | 0.19 | 0.24 | 0.34 |
| 81 | 4.19 | 3.87 | 1.51 | 3.66 | 2.84 | 3.45 | 0.25 | 0.24 | 0.45 |
| 100 | 4.48 | 3.93 | 1.87 | 4.59 | 3.04 | 3.47 | 0.31 | 0.29 | 0.57 |
| 121 | 4.08 | 3.89 | 2.25 | 5.45 | 3.22 | 3.57 | 0.38 | 0.27 | 0.70 |
| 144 | 4.40 | 3.92 | 2.67 | 6.41 | 3.34 | 3.64 | 0.45 | 0.32 | 0.83 |
| 169 | 3.55 | 3.38 | 3.05 | 4.66 | 3.39 | 3.63 | 0.53 | 0.30 | 0.96 |

**Table 10:** Performance comparison for 9x9 patches



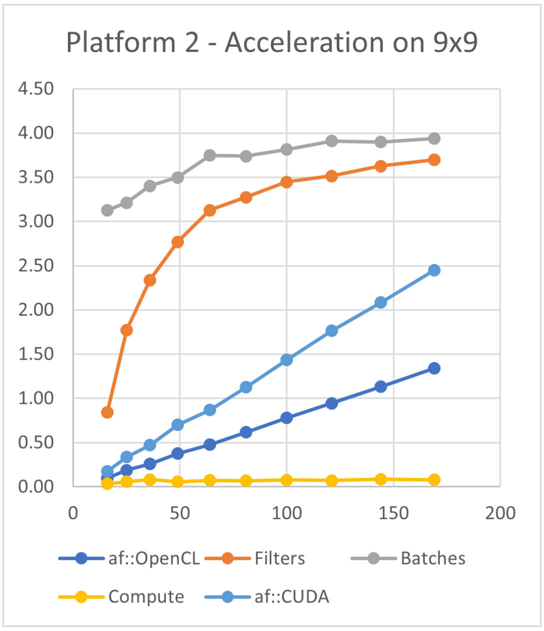


**Figure 6:** Graphical representation of numbers from Table 10

### 5.3.5 Regression

It can be seen that in all cases, the execution time is very strongly correlated with the maximum number of neurons to be found, which is the intuitive conclusion.

In order to analyze to what degree a general purpose algorithm will benefit of parallelization, the team has calculated regression lines for the number of neurons and execution time according to the formulas described in section 2. A polynomial model was chosen, because it is clear from looking at the data, not all the relationships are linear. A polynomial model can model both linear and non-linear relationships well.

| Statistics for 5x5 on platform 1 | | | | |
|---|---|---|---|---|
| Version | Avg. execution time | Covariance | Standard deviation | Correlation coefficient |
| Python | 11 769 306 ms | 136 276 632 | 9 670 838 | 0.989 |
| Single threaded | 2 600 106 ms | 141 900 847 | 2 802 275 | 0.973 |
| Parallel batches | 748 205 ms | 40 552 697 | 800 647 | 0.973 |
| Parallel filters | 1 843 202 ms | 103 756 253 | 2 081 615 | 0.958 |
| Boost compute | 346 891 ms | 14 542 157 | 308 051 | 0.907 |
| ArrayFire OpenCL | 3 763 552 ms | 112 755 855 | 2 170 837 | 0.998 |

**Table 11:** Statistical calculations for 5x5 performance on platform 1

| Statistics for 9x9 on platform 1 | | | | |
|---|---|---|---|---|
| Version | Avg. execution time | Covariance | Standard deviation | Correlation coefficient |
| Python | 12 124 151 ms | 139 966 326 | 9 927 918 | 0.990 |
| Single threaded | 7 412 080 ms | 399 937 410 | 7 858 219 | 0.978 |
| Parallel batches | 1 994 205 ms | 111 121 485 | 2 216 401 | 0.963 |
| Parallel filters | 3 544 625 ms | 170 895 123 | 3 355 342 | 0.979 |
| Boost compute | 1 512 233 ms | 74 816 056 | 1 505 883 | 0.955 |
| ArrayFire OpenCL | 3 718 299 ms | 118 103 593 | 2 271 295 | 0.999 |

**Table 12:** Statistical calculations for 9x9 performance on platform 1

| Statistics for 5x5 on platform 2 | | | | |
|---|---|---|---|---|
| Version | Avg. execution time | Covariance | Standard deviation | Correlation coefficient |
| Python | 8 081 350 ms | 94 021 941 | 6 668 668 | 0.990 |
| Single threaded | 2 378 675 ms | 129 900 659 | 2 565 413 | 0.973 |
| Parallel batches | 653 109 ms | 34 701 433 | 682 785 | 0.977 |
| Parallel filters | 676 917 ms | 34 057 896 | 672 571 | 0.973 |
| Boost compute | 8 655 094 ms | 406 597 927 | 8 037 416 | 0.972 |
| ArrayFire OpenCL | 7 071 260 ms | 223 485 179 | 4 296 797 | 1.000 |
| ArrayFire CUDA | 3 809 051 ms | 120 436 780 | 2 315 932 | 0.999 |

**Table 13:** Statistical calculations for 5x5 performance on platform 2

| Statistics for 9x9 on platform 2 | | | | |
|---|---|---|---|---|
| Version | Avg. execution time | Covariance | Standard deviation | Correlation coefficient |
| Python | 7 830 514 ms | 89 678 301 | 6 355 368 | 0.990 |
| Single threaded | 5 978 099 ms | 327 008 594 | 6 463 626 | 0.972 |
| Parallel batches | 1 550 519 ms | 82 833 492 | 1 706 721 | 0.974 |
| Parallel filters | 1 736 122 ms | 86 548 675 | 1 633 751 | 0.975 |
| Boost compute | 78 175 760 ms | 4 060 232 460 | 79 955 040 | 0.976 |
| ArrayFire OpenCL | 3 718 299 ms | 118 103 593 | 2 271 295 | 0.999 |
| ArrayFire CUDA | 3 813 059 ms | 121 643 166 | 2 339 034 | 0.999 |

**Table 14:** Statistical calculations for 9x9 performance on platform 2

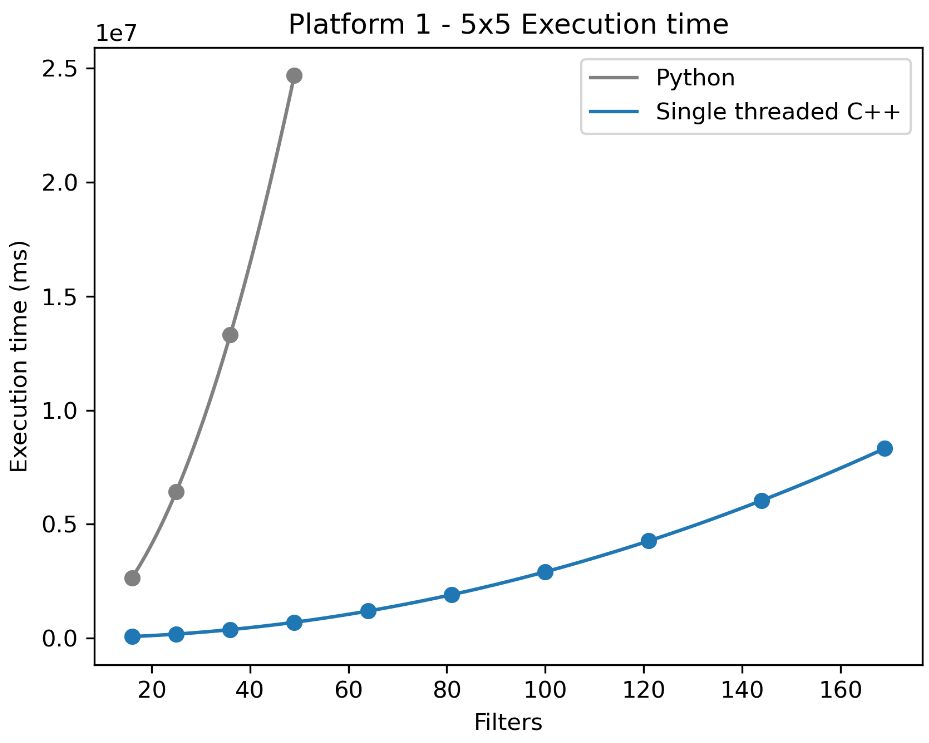


**Figure 7:** Least square estimates of Python and C++ implementation with 5x5 resolution on platform 1

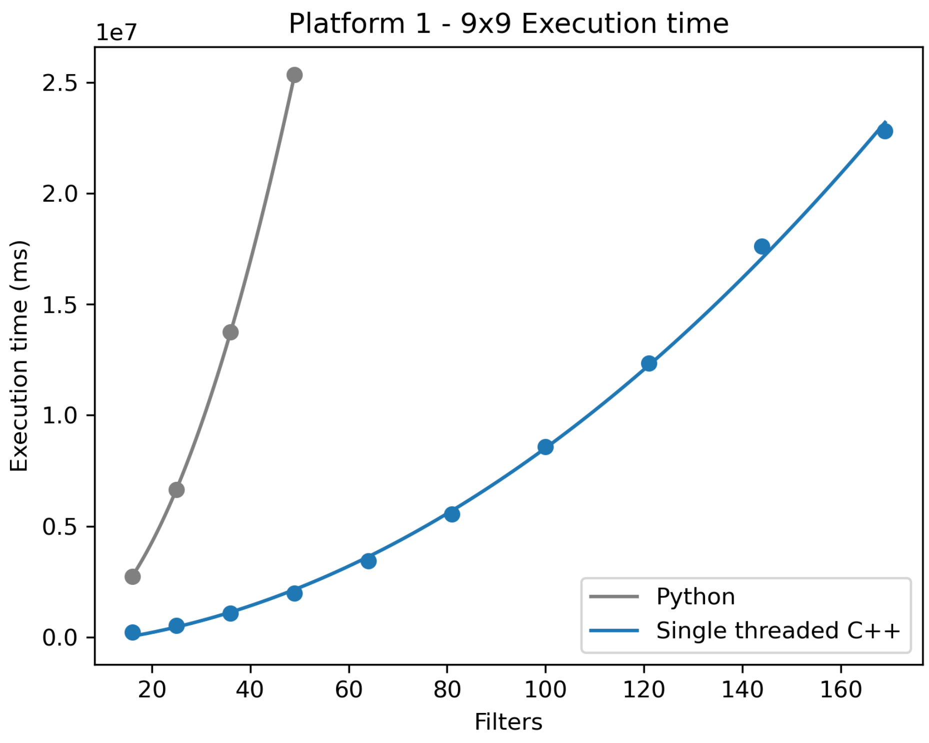


**Figure 8:** Least square estimates of Python and C++ implementation with 9x9 resolution on platform 1

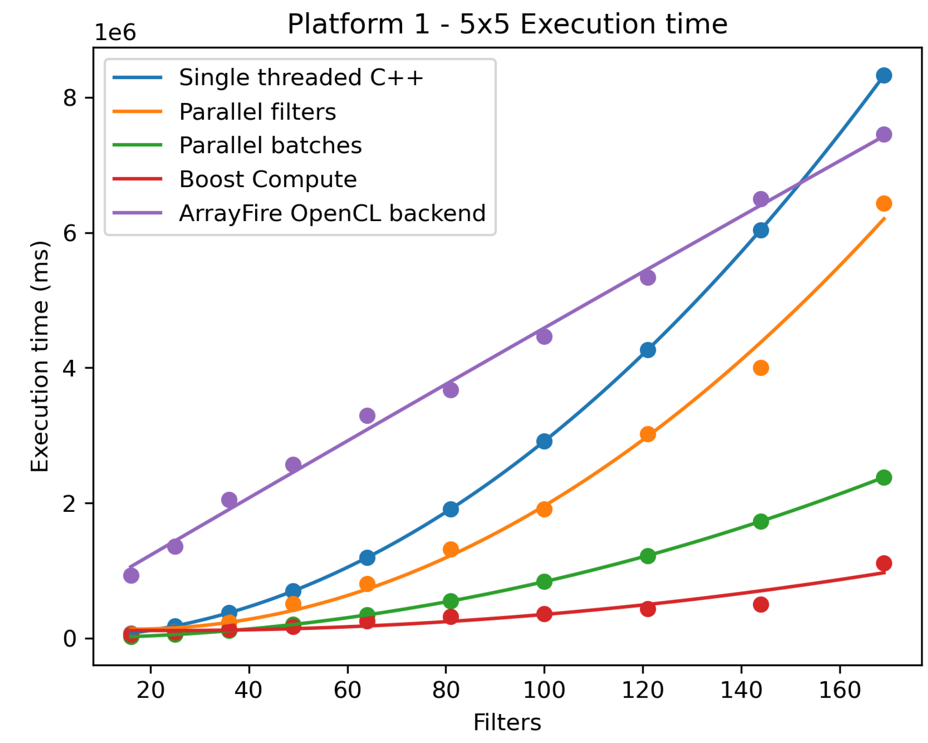


**Figure 9:** Least square estimates of all C++ versions with 5x5 resolution ran on platform 1

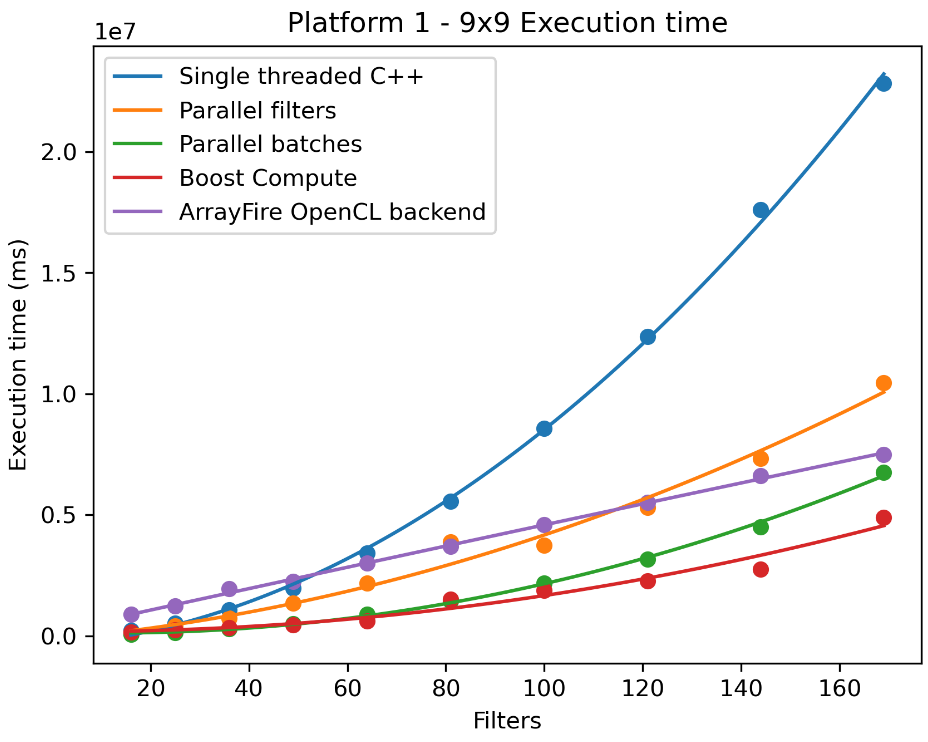


**Figure 10:** Least square estimates of all C++ versions with 9x9 resolution ran on platform 1

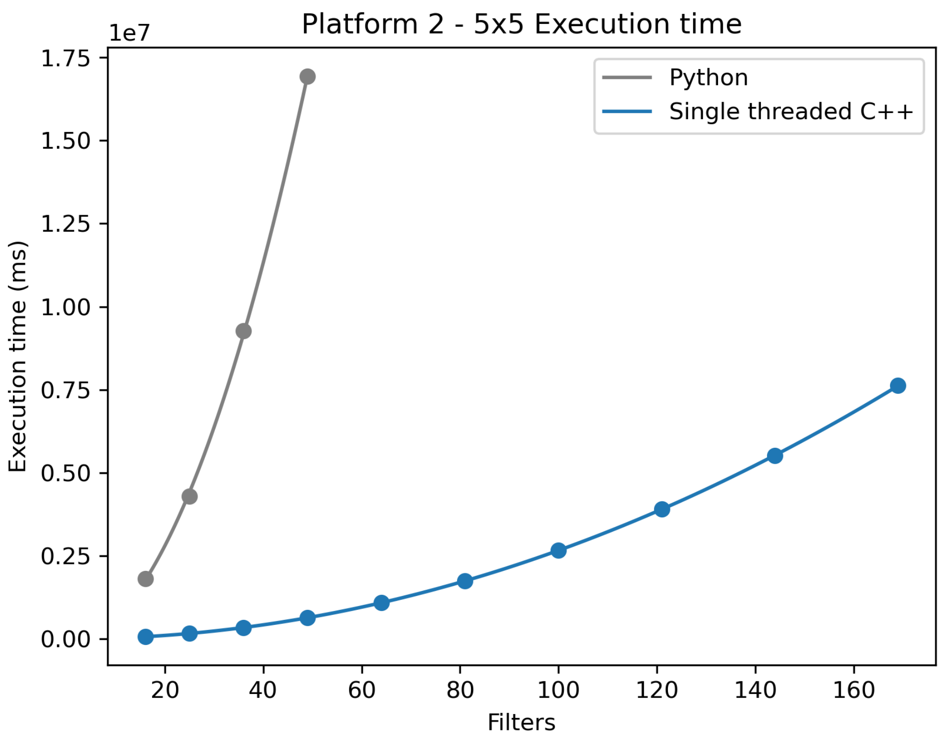


**Figure 11:** Least square estimates of Python and C++ implementation with 5x5 resolution on platform 2

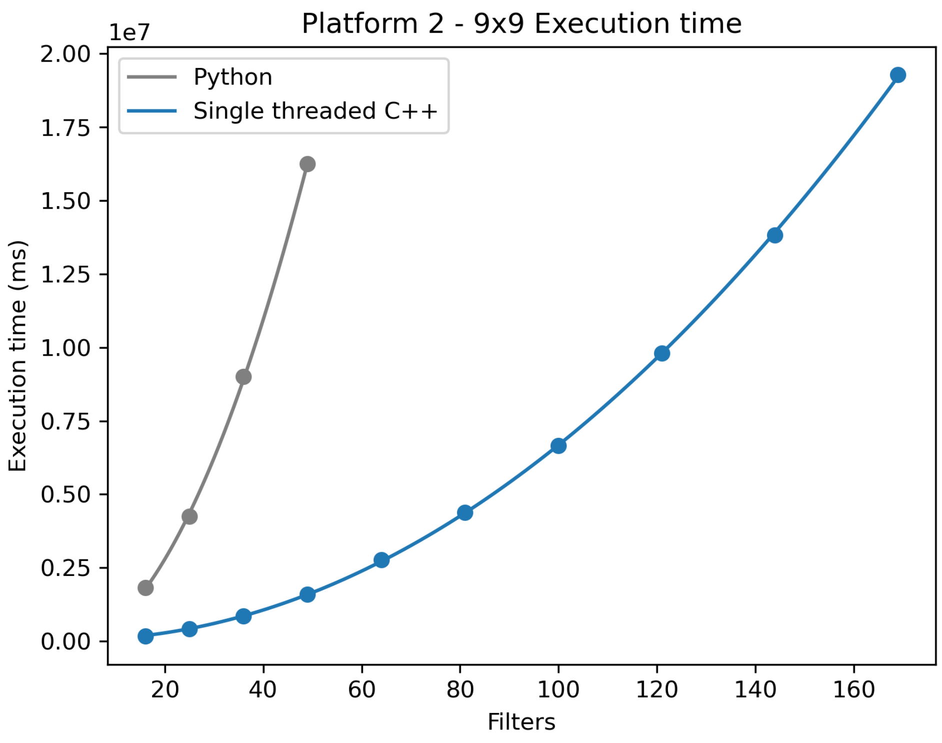


**Figure 12:** Least square estimates of Python and C++ implementation with 9x9 resolution on platform 2

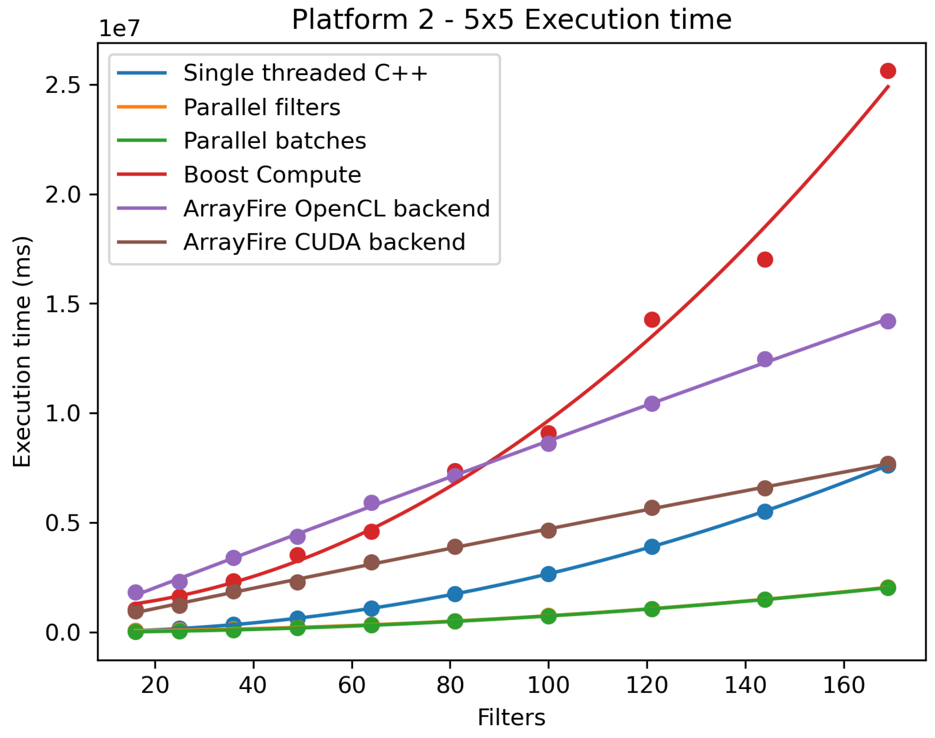


**Figure 13:** Least square estimates of all C++ versions with 5x5 resolution ran on platform 2, parallel filters is only slightly visible under parallel batches

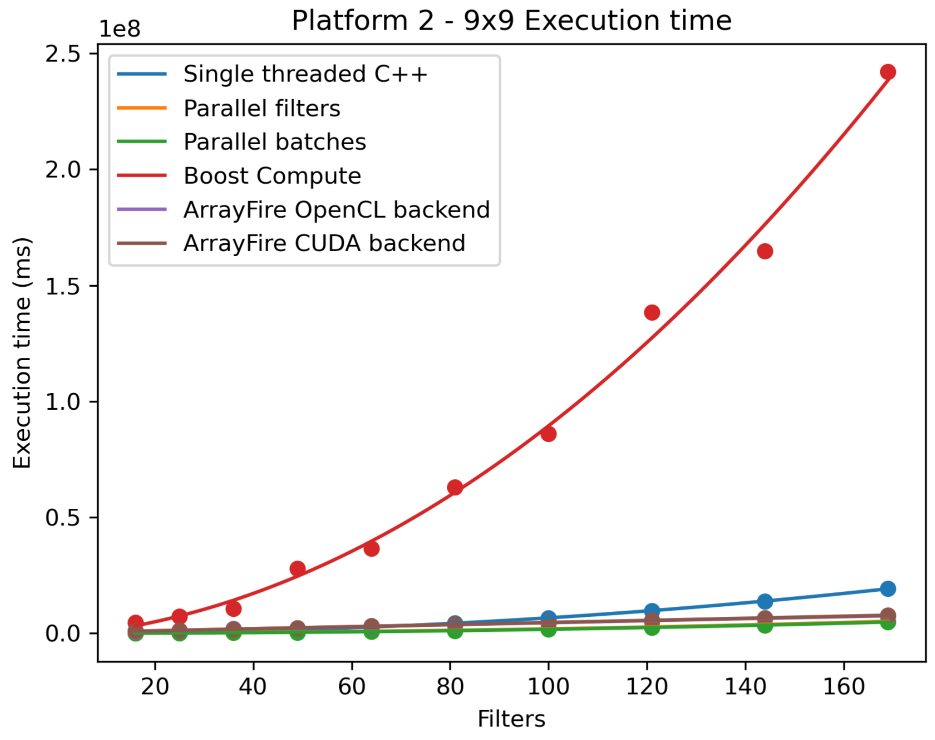


**Figure 14:** Least square estimates of all C++ versions with 9x9 resolution ran on platform 2, parallel filters is overlapped by parallel batches

# 6 Discussion

## 6.1 Sources of Error

There are many sources of error to consider, especially so when measuring small differences in multi-threaded performance. For example, the scheduling of the operating system plays a large part, which makes the whole test susceptible to randomness. However, as all of the tests take place over a minimum of 100 seconds, the randomness associated with context switching remains a small factor for the whole ordeal, due to the law of large numbers [23]. There are also several other sources of error, which will be discussed in the following sections:

### 6.1.1 Hardware

The hardware used for testing is not a dedicated system, but the team members' personal computers. This means that the test data is more affected by external factors, such as programs running in the background, than it would be on a dedicated system. However, the absolute execution time of the program is not what is interesting. This thesis is meant to compare the execution time with the original solution, as well as between the different versions themselves. Even though the results found are not as optimal as they possibly could be, the relative execution time between each program is likely accurate as they all have been tested in similar conditions. This means that the results that the group will be drawing conclusions from are accurate.

Another problem that stemmed from the fact that the data was gathered on the group's private computers, is that the drivers used may not have been optimal. As a caveat of conducting development and testing on Linux, support for required drivers is not as readily available as it would have been on Windows, which may negatively affect the performance the systems. The group believes driver support to be the deciding factor in system 1 having better performance than system 2 in certain implementations, even though system 2 is theoretically better. This is naturally not ideal, as having a large bottleneck in the driver used would make the data gathered on this system less reliable. However, this driver issue likely only affected the GPU implementations relying on OpenCL, meaning that only two implementations were majorly affected.

Additionally, one of the platforms used to gather data has an AMD GPU, which means that it cannot run the CUDA-backend of the ArrayFire implementation. This means that this specific backend can only be accurately compared to the other measurements on made on the same platform, which gives the group a smaller basis to draw conclusions from.

With CUDA being NVIDIA specific, the differences in performance may also be because of NVIDIA-hardware having biases towards CUDA programs. However, as the group is interested in making categorical conclusions based on the performance differences, it is not important whether or not the difference in execution speed is due to CUDA-biases. The important part is that the performance difference is there and will affect real world programs.

### 6.1.2 Code

As with most programs, the code is likely not as optimized as it possibly could be. None of the group members have extensive experience with optimizing code, and there are likely large parts that could be optimized further. A possible problem here would be that if a given version is unoptimized, then it will not give accurate results when compared to another, better optimized version. When comparing very similar implementations, such as the single-threaded and the multi-threaded CPU versions, this is mitigated by the fact that the difference between the versions is minuscule, meaning that any sub-optimal parts in one implementation is very likely present in both, making the comparison accurate.

A problem arises when two versions are substantially different, such as the single-threaded and the GPU-accelerated implementations. While the same calculations are performed, the data structures used are different, and there may be hidden inefficiencies that the team was not able to uncover. For example, the team suspects that the data structures used in order to implement operator overloading for a more direct port of the Python algorithm are more inefficient than using vectors directly. This data structure is present in all CPU versions, but not within and GPU version. The GPU versions either use the dedicated ArrayFire array, or lower level data structures compatible with OpenCL. The team has not had the capacity to develop a CPU version using lower level data structures, meaning that the comparison between the CPU and GPU implementation might be skewed towards the GPU due to design choices and not objective differences. It may therefore be harder to draw sound conclusions.

Another possible source of error would be that the algorithm is implemented incorrectly in one of the versions, leading to unnecessary slowdowns or an inaccurate speedup. The same concept applies here as for the missed optimizations; any math error existing in one version is very likely to exist in every version, which makes the comparisons accurate. All versions and configurations used for the later benchmarks have also been manually tested and proven to produce filters as desired.

## 6.2 Parallel batches

As mentioned previously, the method implementing parallelization of batches cannot guarantee the same level of accuracy as the other versions. This is due to the fact that calculations that are normally run sequentially, are ran in parallel. Since updating a neuron is dependent on its previous value, this type of parallelism cannot guarantee the same results as a sequential calculation would give. This loss of accuracy manifests as fewer filters found, when compared with the more accurate methods, as seen in figure 3. Here, the implementation parallelizing batches was only able to find two filters, compared to six filters found by the the other implementations.

The benefit of using the parallel batches implementation is that it is very fast, and scales well. In every tested case it is the fastest, or second fastest implementation, even when finding many filters. It is only beat in certain cases by the GPU implementation using Boost Compute [17]. The fast execution speed comes from several factors, but the largest one is likely the decrease in overhead, as a result of creating threads less frequently when compared to the strategy employing parallel neurons. In a normal execution with $10^6$ samples, the parallel neuron strategy would create $4 * 10^6$ threads, while the parallel batches implementation would create 4000. In addition to the lessened overhead, the parallel batches strategy is also able to parallelize a larger part of the program.

## 6.3 Technology

### 6.3.1 ArrayFire

Looking back at the project with the experience gained, the team would not choose to develop an ArrayFire implementation of the algorithm again. There are a lot of idiosyncrasies and weaknesses in the documentation that forces a developer to write code in a very specific way, which makes it harder to be productive from the very start. ArrayFire claims to be a high level library, but the team still had to adapt to oddly specific parts, such as having to use gfor loops and being unable to specify how and when the calculations should be parallelized. The biggest merit that is derived from the ArrayFire library is the ability to run a singular application both in OpenCL and in CUDA. However, with the development time the ArrayFire implementation demanded, the team believes a raw CUDA application could have been written instead. The OpenCL backend is not of much use, as the Boost Compute implementation is better performing and was written in less time. The team believes that the difference in execution time stems from having been unable to perform the same memory optimizations in ArrayFire as in Boost Compute.

Writing a CUDA application, instead of using ArrayFire, would have given a more representative performance for the technology, as the team would not have to take into consideration how well optimized the ArrayFire-backend is. Even though CUDA likely has it's own idiosyncrasies, it would be easier to adapt to and create a working application, as the availability of support and useful examples online are much greater in magnitude, due to the larger user base.

### 6.3.2 Boost Compute

The team is overall happy with the outcome of the Boost Compute version. In hindsight, however, it is clear that the documentation for Boost Compute was not ideal, despite initially looking very good. At the time of writing this thesis, it was around 8 years old. This lead to the team being overly reliant on example code, when implementing badly documented functionality, such as local variables for the kernel. Because the team had no experience with OpenCL, it was difficult to know which aspects were general for OpenCL and which were specific for Boost Compute. Luckily, the provided examples were extensive, and covered the team's desired use cases.

Due to writing a custom kernel, most of the functionality used from Boost Compute was for transferring data from the host to the device, something that the team could have done without much more effort in OpenCL. Writing OpenCL directly would have made it easier to find up-to-date documentation, while still being able to find good examples for the desired problems. This would have also made the application dependent on one less external factor.

## 6.4 Scientific results

Overall, the results found were valuable for the team's ability to answer the questions asked at the beginning of the project. All versions of the end product are well optimized within the capacity of the team, and the group feels comfortable making conclusion based on the data gathered. This confidence is strengthened by the fact that the group has gathered data on two different platforms, which gives a wider base and a clearer trend to draw conclusions from.

While the standardized sample size of $10^6$ for all tests gave stability to the results, there were also weaknesses associated with such large scale testing. The most obvious aspect is that some of the results had to be simulated, where only a portion of the standardized sample size was used, since the entire test would be too time-consuming. This means that in theory, these simulated tests are not as accurate as the ones ran in full scale. This decrease in accuracy is however minimal, as the timings for each sample were found to be very constant, and a sample size of $10^5$ is likely enough for any large scale randomness to disappear through law of large numbers [23]. Another weakness that stems from the tests being too time-consuming was the team's inability to gather additional data points. For example, the team would like to further measure performance when using different patch resolutions. This would've allowed for a more comprehensive, multiple regression model that also used the patch size as an explanatory variable, which would have given a more detailed view of the data. A lower standardized sample size would have allowed the team to increase the tests' scale in other aspects such as the number of neurons; measuring up to several thousands of parallel calculations, instead of just up to 169. While the team would have conducted testing differently if the project was to be ran again, the results found are still ample, and the group believes they could prove valuable to others pursuing parallelism.

When reading the results from this thesis it needs to be kept in mind that the algorithm used was parallelizable to a large degree without having to rely on mutexes, which means that the results gathered here cannot be assumed to be valid for less parallelizable problems.

## 6.5 Administrative results

Overall the initial Gantt-chart proved to be quite accurate, especially with the earlier parts of the project. The major differences were that the borders between each section were far more blurred than indicated, and that the optimizations took longer than expected. The GPU development also split into two separate undertakings, as the ArrayFire implementation was more of a roadblock than initially thought.

Another deviation from the Gantt-chart was that there were several issues discovered within different implementations late in the process that needed to be fixed, both small and large. Although these fixes generally did not take much time, it led to the writing being delayed. These errors also led to the gathering of data being a couple of weeks behind schedule, however, most of the thesis could be written without majorly relying on this data, and since the team members had performed large amounts of informal testing, and were able to reflect upon the progress before the final formal data was gathered.

### 6.5.1 Pre-project Goals

One of the primary goals of the group set early in the project was for the end product to have no significant room for improvement that was within the group's capacity. There are currently some likely improvements to be had, such as changes in design or data structures, which could lead to a faster execution time or a more well structured code. These would, however, require significant amounts of time and restructuring for an uncertain increase in performance, and as such, the group has decided that they are not worth exploring within the limited time frame given. The group therefore believes to have achieved all the goals that were set in the project plan, despite there being potential for improvement. Interesting further work that it could be worth looking into in the future is described in section 7.1.

### 6.5.2 Group Dynamics

As mentioned earlier, the group agreed early on not to choose a leader. For future groups considering the same structure, the most important aspect is to have an already established group dynamic and experience from previous group projects. The decisions that would normally have been taken by a group leader has instead been taken in unison. In these cases, the group has been absolutely reliant on good communication and a healthy dynamic, as a major disagreement in the team, for example regarding a major design decision, would have been detrimental to the project as a whole. Luckily, there have been no major conflicts within the group, and overall the process has gone smoothly. Having no clearly defined roles allowed the group to adapt to the needs of the project dynamically.

### 6.5.3 Development

Choosing to not follow strict agile methodology has had both up- and downsides. On one hand, less time has been spent on administrative work, and therefore the time spent developing solutions and gathering data has been maximized. On the other hand, the lack of splitting large tasks into several smaller ones has led to a large degree of ownership over certain parts of the project. This, in turn, made it harder for a team member to work on something that had been primarily developed by the other member. This downside has, however, been largely negated by the team's good communication. Having a dedicated chat server has proven to be useful, as it has given the team members a way to inform each other about potential difficulties they may face in the future.

Not using a predefined workflow also made it possible for each member to work on the aspects of the project where they felt that they would be the most valuable, not simply the aspects of the project that were most prioritized on a task manager. By allowing team members to chose tasks dynamically, it has also made it easier to have a gradual transition into other parts of the development process, as it has not been necessary to have an administrative meeting to create tasks before being able to move onto the later parts of the process. This lack of predefined tasks also led to the members being free to perform the sidestep necessary in order to develop the Boost Compute version, when realizing that the ArrayFire implementation was more a larger undertaking than expected.

## 6.6 Societal implications

The societal implications of more efficient code have been and always will be large. In a world with increasing amounts of energy crises, computer software is one of the very few areas where energy demand can be cut without sacrificing quality of life or productivity. This is increasingly important in a world that is unwilling to sacrifice economic growth, but still needs to transition away from non-renewable energy sources.

Prioritizing economic growth over environmental impact does not stem from malice, but from a desire to move large swaths of the population out of poverty. Richard H. Adams found that a ten percentage point increase in GDP led to a 25.9 % decrease in the proportion of people living on under $ 1 per day [1].

With such strong incentives to continue growth, nations cannot be expected to voluntarily forgo it for the benefit of the environment. Therefore, the pragmatic effort is the one attempting to decouple the strong link between GDP and $CO_2$ emissions, something that has recently happened to a certain degree in industrialised countries [13]. Growth facilitated by the internet and increases in computing efficiency likely plays a large part in this. For example, the share of global energy consumed by data centres has not increased since 2010, despite a 15-fold increase in internet traffic [22]. This is the kind of energy efficient growth that is possible when efficient programming is strongly incentivized, such as in large data centres.

By furthering the knowledge of parallel programming, the necessary threshold for implementing it and GPU-acceleration is lowered; dramatically increasing potential computing efficiency. It is impossible to estimate how much, but simple small scale programs no doubt use large amounts of energy on a global basis. If the threshold for efficiency is lowered enough, then energy used by global computer usage can be cut drastically.

For future green growth not only energy use needs to be taken in mind, but also resource use. This is relevant for this thesis as; by extracting all possible computing power from hardware, it increases their lifespan, which in turn reduces the consumption of consumer hardware and rare earth metals. This is in line with the United Nation's sustainability goal #12, as it would both reduce the consumption and waste of hardware that is currently not being adequately recycled [11]. When analysing computer hardware from cradle to grave, the number of total use cases handled compared to the amount of emissions is very dependent on the type of software it runs. Higher levels of parallelism and optimized software will lead to better outlooks for the computer sector as a whole.

GPU-acceleration is also vital in many areas where heavy computing power is required. For example in areas of research, such as Folding@Home [9]. This is a distributed network of computers, all calculating the folding of proteins. By utilizing Graphics Processing Unit (GPU)s, they have seen a 20-30x speedup compared to their CPU counterparts, allowing them to perform significantly more calculations to the benefit of medical research [10]. By calculating protein folding, diseases that are caused by misfoldings can be understood better, which is vital for future treatment [9].

Computing has allowed many medical breakthroughs, and becomes increasingly more important for the future of medicine as more and more complex systems can be studied through simulation. As Moore's law slows down, this growth in complex simulations will be more reliant on parallelism and efficient computation. By getting out early and educating the masses on the benefits and its applications, the innovations in the medical field will be able to continue, without being reliant on increasing transistor counts.

However, the medical field and parallelism does not have a perfect history. Due to the ethical implications, there are few fields where the authors of software should be more aware of the possible consequences of their implementations. For example, the infamous Therac 25 is a horror story of multi-threaded computing going fatally wrong. Between 1985 and 1987 the Therac 25, which was a radiation therapy machine, gave massive radiation doses to six people; leading to several deaths [16]. This was due to software race conditions, which were not uncovered during development. Computer programmers cannot view the programs they are creating as simple calculations, especially in a field where it may have fatal consequences. Parallel programming adds more pitfalls to an already complicated profession, and it needs to be wielded carefully. Without proper knowledge, the likelihood of rare but serious consequences increases. Because of the scale and speed at which a single person's solutions can be applied, programmers face a special ethical consideration when developing solutions that may affect millions of people.

As a whole, the increased complexity of parallel programming is an important to keep in mind when developing software. Increased sustainability goals should not take precedent over correctness or real world safety, even if it's easy to distance oneself from the actual applications of one's programs. Despite this increased complexity, there is no getting around the fact that parallel computing and GPU-acceleration has a place in a sustainable future. Without programs that scale efficiently without increasing resource consumption, it will be more difficult to decouple economic growth from energy usage and $CO_2$ emissions, which is a necessity if the world as we know it today is going to tackle climate change while still improving the lives of as many people as possible.

# 7 Conclusion

When looking at the data and the statistical analysis, the benefits of multi-threading for an application such as ours is clear. Even in the simplest multi-threaded implementation that perfectly preserved the accuracy of the original program, the execution speed was on average between 1.4 and 3.5 times faster, depending on the scale of the test and the hardware being used. We were also able to find that the crossover point for when the GPU version became faster than the CPU version, was between less than 16 neurons on Platform 1 using Boost Compute and less 169 neurons on Platform 2 using ArrayFire with a CUDA backend. The algorithm used in this thesis is unique in the fact that parts of it was completely independent of each other, meaning that large parts could be parallelized without having to rely on any mutexes. Many real world applications will not see the same scale of benefits as the algorithms seen in this thesis, but the potential is still large.

The data gathered also shows, that in some cases, programs utilizing GPU-acceleration can be faster even at a small scale, despite the larger overhead. This is, however, very dependent on the strategy being used, the hardware available, and the GPU-implementation itself.

Many of the less efficient GPU implementations were also shown to clearly scale much better than the CPU-implementations. Both systems had one of the ArrayFire implementations beating the single-threaded CPU version, within the scale of our testing. The gap between the GPU and CPU would likely grow even larger when scaled up further. Even when the CPU utilized the faster, but less accurate strategy of parallelizing batches, the GPU version could often catch up, even when calculating less than 169 neurons, while still preserving the accuracy of the algorithm.

Overall this has clear implications for future software development. While using multi-core software is challenging, it is more important than ever to minimize the environmental impact of our field, by maximizing the lifetime of hardware and minimizing the electricity used. Even today as CPUs gain more and more cores, a GPU's ability to scale will remain relevant, especially when it comes to generic calculations such as the ones performed in this project.

## 7.1 Further work

There are several aspect of the project that would benefit from being further explored. Firstly, if the CPU version was to be worked on, then an attempt should be made to further optimize it. There are likely several ways to do this, but namely, the data structures should be simplified, as these could be a potential slowdown.

In real world applications, such as in the machine learning models that were a basis for the development of the original algorithm, the CPU-implementation would not be used, in favor of one of the GPU-implementations. If the algorithm is to be used in a real world setting, then the most important change needed to the codebase is to make it adaptable to several different datasets. Currently, large changes are needed if the format of the data being read changes. A second important aspect is how the final output from the program is used. Currently, the neurons are saved to a file at end of each experiment, and can thus be loaded again at a later time whenever needed. This may not be optimal for the desired use case, and an option should be implemented to directly transfer said output into another process.

This process could for example be one that's used for training machine learning models, which would then make it possible to gauge accuracy and performance in a more complete environment. One of the possible tasks that were given in the initial assignment was to compare models trained with local learning rules with those trained with backpropagation, and this would be an interesting exploration if given ample time and effort.

Another recommendation on further work is to move away from the Boost Compute library in favour of directly utilizing OpenCL. This is not because of errors with the library, but due to the fact that the benefit that Boost Compute provides is not worth the risk connected to being dependent on a third party library. Using raw OpenCL would also make getting support for many problems easier, as it is software with a large online presence, unlike Boost Compute. We also recommend creating a dedicated CUDA implementation as well, as this would likely allow a much faster execution speed on NVIDIA GPUs, something that we struggled with throughout the process.

Further recommendations for those completing similar projects in the future are; if you have no experience with GPU programming, it is better to create an initial version using OpenCL or CUDA, instead of directly jumping into a wrapper library. This is due to the fact that idiosyncrasies in the underlying software are likely better documented and easier to get around. Additionally, you will get a better understanding of what your application does and does not need. Using underlying software directly instead of abstracting libraries will lead to a steeper learning curve, but will often lead to a better solution. During the process of working on this thesis we has spent large amounts of time working with libraries that were not strictly needed, and this time could have been better spent working with the underlying technology instead.

## References

[1] Richard H. Adams. ‘Economic Growth, Inequality, and Poverty : Findings from a New Data Set’. In: *Policy Research Working Paper* 2972 (2003). URL: http://hdl.handle.net/10986/19109.

[2] Yoshua Bengio et al. *Towards Biologically Plausible Deep Learning.* 2015. DOI: 10.48550/ARXIV.1502.04156. URL: https://arxiv.org/abs/1502.04156.

[3] Kevin Carbotte. *Nvidia’s new Titan V pushes 110 Teraflops from a single chip.* Visited 2022-05-10. Dec. 2017. URL: https://www.tomshardware.com/news/nvidia-titan-v-110-teraflops,36085.html.

[4] Beman Dawes. *Boost Libraries - 1.64.0.* Feb. 2015. URL: https://www.boost.org/doc/libs/1_64_0/libs/libraries.htm#Concurrent (visited on 30th Apr. 2022).

[5] Li Deng. ‘The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]’. In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142. DOI: 10.1109/MSP.2012.2211477.

[6] Ole Christian Eidheim. *Revisiting Gaussian Neurons for Online Clustering with Unknown Number of Clusters.* 2022. DOI: 10.48550/ARXIV.2205.00920. URL: https://arxiv.org/abs/2205.00920.

[7] The Editors of Encyclopaedia Britannica. *Moore’s law.* July 2011. URL: https://www.britannica.com/technology/Moores-law.

[8] Benno Evers. *matplotlib-cpp.* https://github.com/lava/matplotlib-cpp. 2014. (Visited on 25th Apr. 2022).

[9] Folding@home. *Diseases.* URL: https://foldingathome.org/diseases/?lng=en (visited on 6th May 2022).

[10] Folding@home. *Does folding@home run on my graphics chip or GPU?* URL: https://foldingathome.org/faqs/running-foldinghome/foldinghome-run-graphics-chip-gpu/ (visited on 11th May 2022).

[11] *Goal 12 — UN Department of Economic and Social Affairs.* URL: https://sdgs.un.org/goals/goal12.

[12] Kurt Guntheroth. *Optimized C++: Proven Techniques for Heightened Performance.* 1st. O’Reilly Media, Inc., 2016.

[13] Helmut Haberl et al. ‘A systematic review of the evidence on decoupling of GDP, resource use and GHG emissions, part II: synthesizing the insights’. In: *Environmental Research Letters* 15.6 (June 2020), p. 065003. DOI: 10.1088/1748-9326/ab842a. URL: https://doi.org/10.1088/1748-9326/ab842a.

[14] Ahmad Hesam et al. ‘GPU Acceleration of 3D Agent-Based Biological Simulations’. In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* IEEE, June 2021. DOI: 10.1109/ipdpsw52791.2021.00040. URL: https://doi.org/10.1109%2Fipdpsw52791.2021.00040.

[15] Alex Krizhevsky. *Learning multiple layers of features from tiny images.* Tech. rep. 2009.

[16] N. G. Leveson. ‘The Therac-25: 30 Years Later’. In: *Computer* 50.11 (Nov. 2017), pp. 8–11. ISSN: 1558-0814. DOI: 10.1109/MC.2017.4041349.

[17] Kyle Lutz. *Boost Compute.* https://www.boost.org/doc/libs/1_66_0/libs/compute/doc/html/index.html. Last accessed 2022-05-03. 2022.

[18] John Markoff. ‘Smaller, Faster, Cheaper, Over: The Future of Computer Chips’. In: *New York Times* (Sept. 2015). URL: https://www.nytimes.com/2015/09/27/technology/smaller-faster-cheaper-over-the-future-of-computer-chips.html.

[19] Douglas Montgomery, Elizabeth Peck and G. Vining. ‘Multiple linear regression’. In: *Introduction to linear regression analysis, 5th Edition*. John Wiley & Sons, 2012, pp. 67–73.

[20] PCMag. *Definition of GPU*. URL: https://www.pcmag.com/encyclopedia/term/gpu (visited on 10th May 2022).

[21] Roger Ratcliff. ‘Connectionist models of recognition memory: constraints imposed by learning and forgetting functions.’ In: *Psychological review* 97 2 (1990), pp. 285–308. DOI: /10.1037/0033-295X.97.2.285.

[22] Timothy Rooks. *Big data centers are power-hungry, but increasingly efficient*. 2022. URL: https://p.dw.com/p/45cO4 (visited on 5th May 2022).

[23] Richard Routledge. *Law of large numbers*. June 2005. URL: https://www.britannica.com/science/law-of-large-numbers.

[24] Joseph Steppan. *File:MnistExamples.png - Wikimedia Commons*. https://commons.wikimedia.org/wiki/File:MnistExamples.png. [Online; accessed 26-03-2022]. 2017.

[25] Christian Szegedy et al. *Intriguing properties of neural networks*. 2013. DOI: 10.48550/ARXIV.1312.6199. URL: https://arxiv.org/abs/1312.6199.

[26] M. Mitchell Waldrop. ‘The chips are down for Moore’s law’. In: *Nature* 530.7589 (Feb. 2016), pp. 144–147. DOI: 10.1038/530144a.

[27] Han Xiao et al. ‘A Parallel Algorithm of Image Mean Filtering Based on OpenCL’. In: *IEEE Access* 9 (2021), pp. 65001–65016. DOI: 10.1109/ACCESS.2021.3068772.

[28] Pavan Yalamanchili et al. *ArrayFire - A high performance software library for parallel computing with an easy-to-use API*. Atlanta, 2015. URL: https://github.com/arrayfire/arrayfire (visited on 25th Apr. 2022).

## List of Attachments

Below is a list of files contained within the attached project folder.

1. Project Handbook
   - A Contract
   - B Progress Plan
   - C Meetings
   - D Weekly Reports
2. Link to repository