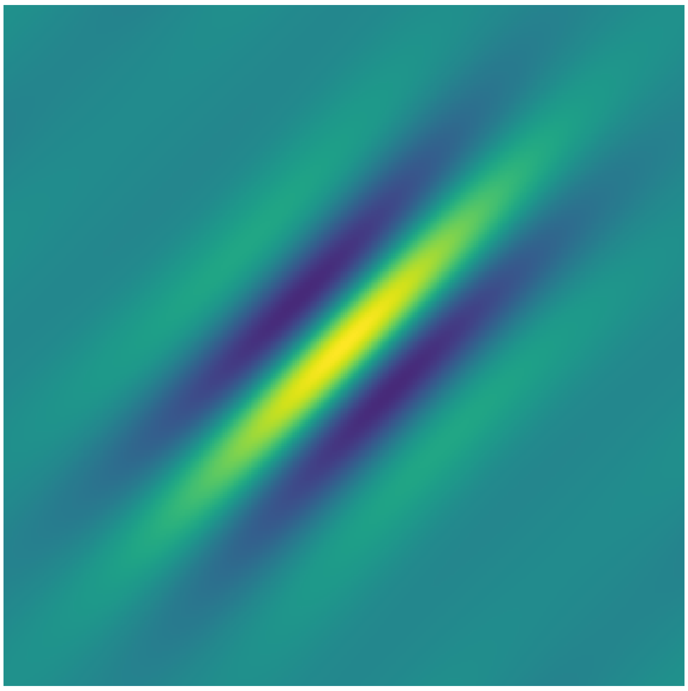
Bachelor's thesis

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

Edvard Schøyen
Jarand Jensen Romestrand
Vetle Ålesve Nordang

Parallelization of Gabor-like Filter Application on Different Hardware Architectures

Bachelor's thesis in Computer Science
Supervisor: Ole Christian Eidheim
May 2024



NTNU
Norwegian University of
Science and Technology

Edvard Schøyen
Jarand Jensen Romestrand
Vetle Ålesve Nordang

Parallelization of Gabor-like Filter Application on Different Hardware Architectures

Bachelor's thesis in Computer Science
Supervisor: Ole Christian Eidheim
May 2024

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 inspired by a project conducted in the course ”Applied Machine Learning with Project” (IDATT2502) which involved using Gabor filters for image classification and exploring the effect of classifying these images with different levels of noise A. The focus of this thesis is on the parallelized application of the new Gabor-like filters. These filters are developed and implemented using the algorithms described in the article by Devakumar and Eidheim (Devakumar & Eidheim, 2024).

Different parallel hardware options, including Graphics processing unit (GPU), Field Programmable Gate Array (FPGA), and Application-specific integrated circuit (ASIC), for parallelizing the filter application have been explored. However, due to time constraints, implementations have been made only for Central Processing Unit (CPU) and GPU. Multiple versions have been developed for both the CPU and the GPU, for a thorough comparison of the implementations and their performance on different hardware. For the CPU, there are three implementations: a sequential version in Python using NumPy, another sequential version in Rust, and a parallelized version in Rust utilizing Rayon. The GPU implementations have been written in both Rust and C++. Different shaders, which are code for GPU execution, have been developed and tested[1].

The processing time for the implementations has been collected from the platforms to determine which was more effective. This was achieved using the hardware clocks on each platform to measure the time taken for each run of the filter application. The results show that the GPU outperforms the CPU as long as the time of the memory transfer between the CPU and GPU is shorter than the computation time of the CPU. Another indication from the results is the substantial variation in computational performance observed when implementing diverse parallelization strategies.

[1]The source code for all implementations, along with information about dependencies for execution and tests, can be found at https://github.com/s24-idatt2900-072/parallelization

Sammendrag

Denne bacheloroppgaven er inspirert av et prosjekt gjennomført i emnet ”Anvendt maskinlæring med prosjekt” (IDATT2502), som omhandlet bruk av Gabor-filtre for bildeklassifisering og utforsking av effekten av støy på disse bildene, se vedlegg A. Fokuset i denne oppgaven er på den parallelliserte anvendelsen av de nye Gabor-lignende filtrene. Disse nye filtrene er utviklet og implementert ved å bruke algoritmen som finnes i artikkelen av Devakumar og Eidheim (Devakumar & Eidheim, 2024).

De ulike parallelle maskinvarealternativene GPU, FPGA og ASIC for å parallelisere filteranvendelsen er utforsket. På grunn av tidsbegrensninger er implementeringer kun gjort for CPU og GPU. Flere versjoner er utviklet for hver av disse for en grundig sammenligning av deres ytelse. Versjonene for CPU er én sekvensiell versjon i Python ved bruk av NumPy, en annen sekvensiell versjon i Rust, og en parallelisert versjon i Rust ved bruk av Rayon. GPU-implementeringene er skrevet i Rust og C++. Forskjellige shaders, som er kode for GPU, er utviklet og testet [2].

Prosesseringstiden for versjonene er samlet inn fra plattformene for å fastslå hvilken som var mer effektiv. Dette ble oppnådd ved å bruke maskinvareklokkene på hver plattform for å måle tiden som ble brukt for hver kjøring av filteranvendelsen. Resultatene viser at GPU-en overgår CPU-en så lenge tiden for minneoverføringen mellom CPU-en og GPU-en er kortere enn prosesseringstiden til CPU-en. En annen indikasjon fra resultatene er den betydelige variasjonen i ytelsen til beregningene som observeres ved implementering av de ulike paralleliseringsstrategiene.

[2]Kildekoden for alle versjoner, sammen med informasjon om avhengigheter for utførelse og tester, kan finnes på https://github.com/s24-idatt2900-072/parallelization

Preface

Three computer science undergraduates at the Norwegian University of Science and Technology (NTNU) authored this bachelor's thesis in the spring of 2024. The project was on behalf of Ole Christan Eidheim, who also supervised during the period. The thesis stems from a project done in the course Applied Machine Learning with project (IDATT2502). The project tested Ole Christian Eidheim's idea of a new image recognition model. It used Gabor filters to classify numbers and tested its resilience against noise. The future research for this previous project was to improve the training of the model to make it scale with larger data sets and more filters. This thesis builds upon future work by researching and testing different approaches to optimize the training.

With the desire to learn more about optimizing and low-level language, this project suited the objectives of the group, given the complexity of GPU programming and low-level language use with Rust and C++.

Working on the project has been both educational and challenging. It has allowed us to improve our understanding of different hardware as well as low-level implementation skills. The ability of the group to communicate and share their thoughts has improved as well, making them better individual members who can contribute effectively to other projects. With this in mind, the overall project has been successful and has shown promising results. The group is proud to present their findings.

Firstly, we would like to thank Ole Christian Eidheim for his guidance and for providing us with the task. Stian Wilhelmsen and Jonathan Andre Vedang for helping us with organizing the room schedule and allowing us to work in quiet environments. Trine Welle for proofreading and feedback on the structure of the thesis.

Edvard Schøyen	Jarand Jensen Romestrand	Vetle Ålesve Nordang
Edvard Schøyen	Jarand Romestrand	Vetle Ålesve Nordang
Trondheim, 20.05.2024	*Trondheim, 20.05.2024*	*Trondheim, 20.05.2024*

Assignment Details

The task was originally given in Norwegian and has been translated by the group:

Description of the assignment and problem

This task involves parallelizing different methods on CPU, GPU, and/or other specially designed hardware. The focus is on highly parallel methods that use a lot of data and can require a lot of memory. It is interesting how these methods can be implemented in C, C++, and/or Rust, and find out which frameworks or libraries that are most appropriate to use.

The project is particularly interesting in relation to new machine-learning methods that are being developed in our research environment. A possible direction is therefore to try out heavier methods for image analysis which, for example, may be better suited in relation to noise and adversarial attacks than today's solutions.

Project focus

The main focus of the thesis was looking at the parallelization of a new machine-learning method on the CPU and GPU using Rust as the main programming language. A C++ implementation was also necessary to compare today's Rust against a more evolved programming language. Early in the project period, the team investigated the potential of using specially designed hardware, but this was deemed to be too challenging. The potential in specially designed hardware will therefore be discussed from a theoretical point of view instead.

Contents

Figures

Tables

Algorithms

Glossary

artifact An artifact in Design Science Research is a constructed solution, specifically designed to address and solve the described problem within research. These artifacts are both products of research and tools used for making further progress in the research. . 22, 23, 25–27, 60, 63

bus A bus, in computer architecture, is a system that transfers data between components inside a computer, or between computers. 8

busy wait a synchronization technique where the process waits and continuously checks for a condition to be satisfied before going ahead with its execution. . 11

context switching Switching of a thread's execution to another by storing its state. 7, 68

Gabor Gabor filter is a filter used in image processing to extract features and detect edges. Named after Nobel prize winner in physics Dennis Gabor, an electrical engineer, and physicist. . i, iii, v, 1, 15, 16, 20, 29, 30, 80, 83, 84

hyperthreading Hyperthreading is a technology used by some Intel processors that allows each CPU core to handle two threads at the same time Patterson and Hennessy, 2014, p. 564-568 . 72

mean A numeric quantity representing the center of collected data, also called the expected value.. 18

MNIST A large database of handwritten images that is commonly used for training image processing systems. The size of the images is 28×28 pixels. . 32

NumPy A Python library that is written in mostly C and C++ for numerical computing. . i, iii, 29, 38, 40, 45, 58, 66, 72, 79

NVIDIA NVIDIA is a technology company known for its powerful GPUs. . 10, 66, 67, 72, 74

race condition A situation where the behavior of a program depends on the relative timing of multiple threads or processes.. 11, 12

shader A program executed on the GPU for parallel processing tasks. i, iii, 8, 11, 28, 29, 31–36, 38, 40, 41, 45, 47, 49–56, 58, 66–71, 74, 75, 79

spline A spline is a function defined piecewise by polynomials.. 18

wgpu WGPU is a graphics library for Rust based on the WebGPU API. It is suitable for general-purpose graphics and computing on the GPU. . 28, 32, 65–69, 74, 75, 80, 81

Acronyms

AI	Artificial Intelligence.
ALU	Arithmetic Logic Unit.
ASIC	Application-specific integrated circuit.
CNN	Convolutional Neural Network.
CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture.
DSR	Design Science Research.
FPGA	Field Programmable Gate Array.
GPGPU	General-purpose computing on graphics processing unit.
GPU	Graphics processing unit.
HDL	Hardware description language.
HPC	High-performance computing.
IDE	Integrated Development Environment.
NTNU	Norwegian University of Science and Technology.
RAM	Random Access Memory.
SIMD	Single Instruction, Multiple Data.
TPU	Tensor Processing Unit.
UN	United Nations.
VRAM	Video Random Access Memory.
VS Code	Visual Studio Code.
WGSL	WebGPU Shading Language.

1. Introduction

Machine learning models for image recognition, such as Convolutional Neural Network (CNN), have become increasingly popular in recent years. However, CNN has still some issues. Firstly, they have shown problems with image recognition on images with noise and are prone to adversarial attacks (Momeny et al., 2021). Additionally, CNNs have been criticized for not being biologically plausible (Eidheim, 2022).

In the fall of 2023, a project was presented to a group of four students in the course IDATT2502 - Applied Machine Learning with Project. The supervisor was Ole Christian Eidheim, and the project built upon the use of Gabor filters to recognize images in the MNIST dataset without backward propagation. It was created to provide a proof of concept based on an idea by Eidheim. The model uses Gabor filters to better simulate a more biologically plausible machine learning model and be more robust against noise. The paper is included as appendix A.

The results of the project in IDATT2502 were promising but had some limitations. One significant issue was the time complexity of training the model, which was implemented in Python using only the CPU. This restriction limited the number of filters that could be used in the simple model. The future work concluded that optimizing the calculations to better scale with more Gabor filters would improve the performance of the model. Since the calculations to apply a Gabor filter are independent, it is theoretically possible to calculate every filter on every image simultaneously.

The power of the CPU is not sufficient in the field of machine learning, while the GPUs has been used to train models because of its computational power (Jeon et al., 2021). This led to the thought that the optimization of training the model made in IDATT2502 could be done on specialized hardware for parallel algorithms such as GPU, FPGA, and ASIC. This raised the question of how one can implement independent machine-learning calculations that use no backward propagation.

This thesis investigates how independent calculations in machine learning models that do not use backward propagation can be optimized for specialized hardware designed for parallelization. It explores designing an implementation that utilizes the hardware in the best possible manner and evaluates how much faster it is compared to existing implementations made on the CPU.

1.1 Structure

2 Theory and relevant literature:

Present relevant theory and literature for understanding the development of algorithms designed to run on parallel hardware such as a GPU.

3 Methodology and process:

This section describes how the project was planned, the work methodology, the choice of technology, and the scientific approach.

4 Results:

The results from all of the tests are presented with tables and figures for clear visualization and easy interpretation. Non-scientific results such as administrative details, are also included.

5 Discussion:

This section provides a discussion and reflection on the results presented in the previous section. The results are explained based on the current theoretical understanding.

6 Conclusion and further work:

This section evaluates how well the suggested approaches address the problem presented in the introduction. It also discusses potential future work that could lead to a better understanding of and higher computational power for independent machine learning calculations.

7 Societal Impact:

Discusses the potential environmental and societal impact of the results of the project.

2. Theory

This chapter will present the necessary knowledge required to go further in-depth on the topics of parallel computing on different hardware, and their application in machine learning. The theory described here will lay the foundation for the knowledge required to answer the problem described in the task description, introduced in Section Assignment Details.

2.1 Central Processing Unit

The CPU is the main component of a computer and is responsible for most of the processing in the system. It does so by interpreting signals from both the software and hardware of the system (ARM, 2024b).

The CPU is made up of several components, with one of the most crucial ones being the Arithmetic Logic Unit (ALU). The ALU is responsible for performing all arithmetic and logical operations, from basic operations such as addition to more complex operations such as bit manipulation. This component is central in most operations executed by the CPU, as it processes both register-to-memory and register-to-register instructions. It cooperates closely with the other components of the CPU, receiving instructions from the control unit and utilizing data stored in the registers (Tanenbaum, 2013, p. 55-60).

The control unit is responsible for fetching and decoding instructions from the memory of the computer and directing them to the appropriate components. It handles the execution flow of the CPU by managing the flow of instructions and data between the ALU, registers, and memory (Tanenbaum, 2013, p. 55).

Registers in the CPU are responsible for fast access to data and instructions that the CPU needs immediately. There are several registers but they can often be generalized into two types of registers. General-purpose registers are responsible for storing key local variables and intermediate results of calculations. The other kind of generalized registers is the special-purpose registers, including important features such as the program counter and the stack pointer (Tanenbaum, 2013, p. 349-351).

The definition of an instruction set is a collection of all the operations the CPU can execute. These operations might include arithmetic or logical tasks performed by the ALU. This set of instructions varies between the dif-

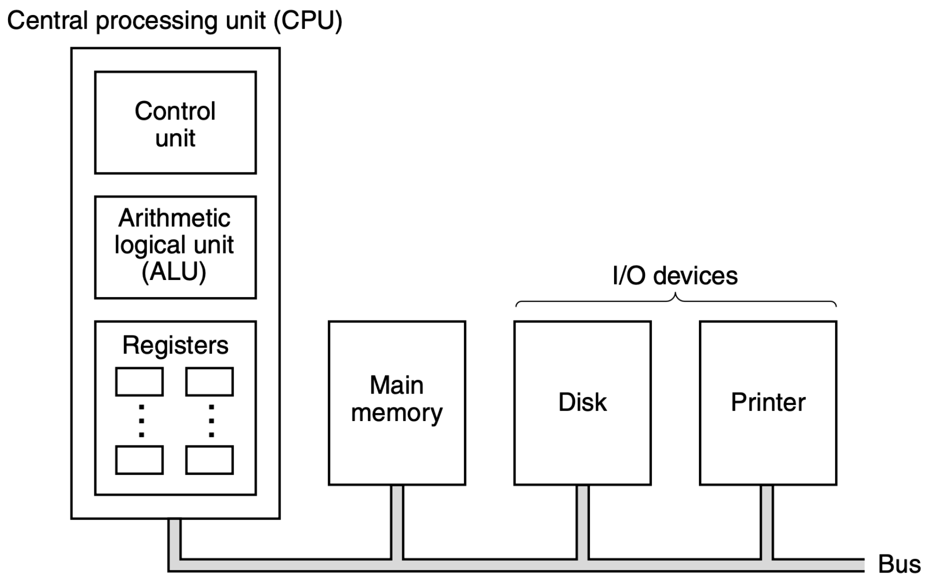


Figure 2.1: The organization of a simple computer with one CPU and two I/O devices. Owner (Tanenbaum, 2013, p. 56)

ferent CPU architectures and defines how the CPU handles the instruction (Tanenbaum, 2013, p. 344-351).

The rate at which the CPU can execute these instructions is determined by the clock frequency of the CPU. The clock frequency is measured in gigahertz (GHz) and determines the number of operations the CPU can perform each second. Therefore, a higher clock count frequency will result in faster execution speed, but will also consume more power and generate higher temperatures, which can lead to potential throttling (‘Understanding CPU Clock Speed’, 2024).

While the clock frequency indicates the execution speed of a single core, multi-core processors address processing efficiency in a fundamentally different manner. A multi-core processor can schedule tasks independently, which enhances system performance without the need to increase the clock speed. These extra cores can utilize pipelining, which is a technique where multiple instructions are overlapped in execution (Patterson & Hennessy, 2014, p. 272-286).

2.2 Graphics Processing Unit

The GPU is another important component of the computer primarily handling tasks such as image processing and 3D rendering. Beyond these functions, the GPU is also a specialized electronic circuit designed to process complex mathematical calculations (Tanenbaum, 2013, p. 582).

One of the key attributes of the GPU is its parallel processing architecture based on Single Instruction, Multiple Data (SIMD), which consists of numerous processing units each performing an operation each cycle (Tanenbaum, 2013, p. 583). This allows the GPU to process multiple operations concurrently, making it efficient in the computation of complex calculations. While this allows for efficient graphics display output, it also enhances the overall system performance by offloading the CPU (Tanenbaum, 2013, pp. 70–73).

The main components the GPU is made up of include its cores, the Video Random Access Memory (VRAM), and the bus interface. The cores, which can be general-purpose or specialized depending on the architecture, are primarily responsible for computational tasks. The VRAM serves as temporary storage for data being processed on the GPU. Lastly, the bus interface provides the connection between the GPU and the rest of the system and is responsible for the data transfer between GPU and system memory.

Over the years the GPU has evolved from fixed-function units to programmable units. This evolution allowed GPUs to not only manage graphics but also made it possible for it to handle tasks normally performed by the CPU. This is the fundamentals of General-purpose computing on graphics processing unit (GPGPU) (Tanenbaum, 2013, pp. 582–585).

The GPUs range of use has evolved as GPGPU became available. This evolution opened up other fields of computation to utilize the processing powers of the GPU. The GPU can now perform a more broad range of computational tasks in parallel. This makes it faster at running highly parallel algorithms compared to the CPU when run at a certain scale (Tanenbaum, 2013, pp. 582–585).

2.3 FPGA and ASIC

FPGA is a type of integrated circuit that lets the user reconfigure the hardware itself after being manufactured. The internal connections and logic components of the FPGA can be adjusted to suit the user's needs, which allows for a variety of digital circuits and functions. This degree of flexibility makes FPGAs especially suitable for prototyping, education, and research (ARM, 2024c).

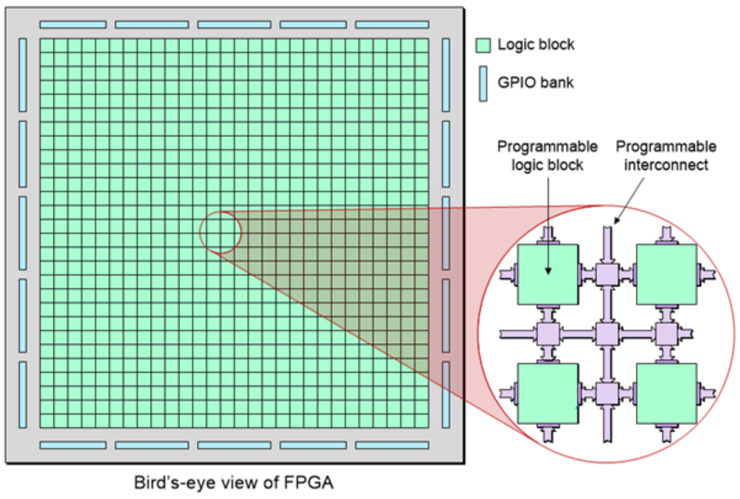


Figure 2.2: The reconfigurable architecture of an FPGA consisting of several logic blocks and interconnections. Owner (Semiconductor, 2024)

In contrast to FPGA, ASIC is designed with the purpose of a fixed functionality. An ASIC is an integrated circuit that is designed for performing a particular task. This specialization allows ASIC to perform exceptionally well for these tasks, making them ideal for repetitive tasks that need to be executed rapidly but also reliably (ARM, 2024a).

2.4 Parallelization

Parallelization is a technique used to increase efficiency by dividing a larger task into smaller, more manageable subtasks. These subtasks can then be executed simultaneously across multiple computing units. This approach is popular in modern computing, especially with the demand for fast processing of larger volumes of data (Robey & Zamora, 2021, pp. 2–6).

2.4.1 Distributed vs Shared Memory Architectures

Parallel programming can be implemented using either distributed memory or shared memory architectures. Distributed memory architectures typically consist of a collection of nodes, each with its local memory, connected with other nodes through a network. This architecture is scalable, as each node operates independently from the others (Robey & Zamora, 2021, pp. 22).

In contrast, in shared memory architectures, multiple nodes have a shared memory space. This simplifies the problem of data sharing by making the memory accessible to all nodes. Synchronization between processes is therefore important, to ensure correct program execution and data consistency (Robey & Zamora, 2021, pp. 22–23).

2.4.2 Parallelization on CPU

Parallelization on the CPU utilizes the multi-core structure of modern CPUs to execute tasks concurrently across the cores. This approach optimizes the use of resources, reduces idle times, and improves execution speed.

Attempting to parallelize a task beyond the number of available cores can result in frequent context switching, creating overhead and taking a toll on the benefits of parallelization.

Effective parallelization relies on balancing the workload to ensure all cores are optimally utilized without overwhelming the system (Tanenbaum, 2013, pp. 554–574).

2.4.3 Parallelization on GPU

The GPU excels at processing large amounts of data in parallel by taking advantage of the vast amount of cores. This parallel architecture makes GPUs well-suited for accelerating data processing. Unlike the CPU, where excessive parallelization beyond the available number of cores can lead to overhead and frequent context switching, the GPU benefit from dividing the task into as many subtasks as possible. The low cost of context switching on the GPU allows for rapid switching between tasks, thereby reducing the impact of latency from memory operations. This keeps the GPU busy, effectively hiding the latency caused by memory access and keeps it continuously computing without the need to wait for data. This makes the GPU exceptionally well-suited for data processing tasks (ENCCS, 2024).

2.4.4 Parallelization on FPGA and ASIC

FPGAs and ASICs offer specialized solutions for hardware acceleration in computing systems. They excel in handling tasks in parallel while ensuring high performance and low latency.

FPGAs are especially suited for parallelizable tasks due to their reconfigurable nature. By leveraging their customizable architecture data processing can be enhanced and one can potentially expect higher efficiency compared to processing the data on the CPU and possibly the GPU. The key to this performance boost lies in its array of configurable logic blocks and programmable interconnections, which allows for flexible data routing. Each of these blocks can individually be programmed to perform a task concurrently. Customized data paths can therefore be created, enhancing the parallel execution flow.

Furthermore, FPGAs often serve as prototyping platforms in the development of ASICs (Markovic et al., 2007). This allows for thorough testing and optimization of the ASIC before assembling the final circuit, which is crucial due to the non-reconfigurable characteristics and higher production cost compared to FPGA. The developed ASIC will generally offer better performance and power efficiency than their FPGA prototype counterpart. This stems from ASIC being specialized to perform specific operations, making it able to operate at higher speeds and lower latency. Therefore, ASIC's are ideal for tasks involving high-volume, repetitive data processing.

2.5 GPU Architecture

When a program is to perform calculations, it first initializes a function on the GPU known as a shader. Unlike typical CPU operations, the shader is not set to run sequentially, but in parallel utilizing thousands of threads that are distributed across its cores. This parallel execution model makes the GPU suitable for handling complex computations efficiently, increasing processing speed compared to the CPU.

2.5.1 Memory and data transfer

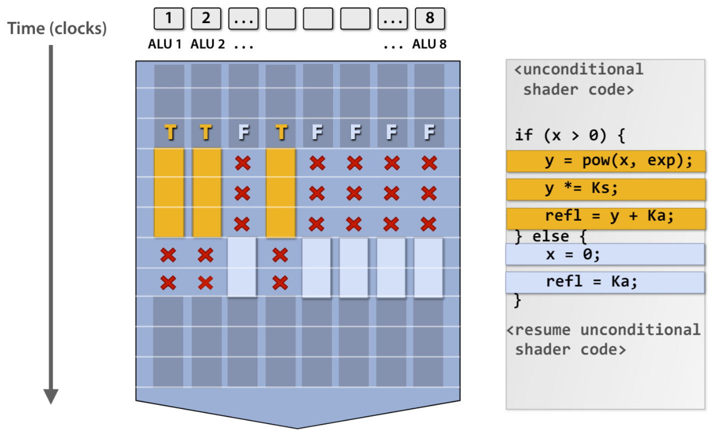
The GPU has its own dedicated memory separate from the host. Therefore, data must be transferred between the host and the GPU using buses. This approach is generally slower than the transfers between the CPU and the Random Access Memory (RAM). Therefore, for smaller calculations, this slower transfer rate overshadows the computational gains from computing on the GPU (Ansorge, 2022, pp. 318–319).

2.5.2 Threads

The threads in the GPU are much lighter than their CPU counterparts. To further organize and streamline this process, the threads are grouped in units.

These thread groups play an important role in the GPU architecture to achieve optimal performance. They run and execute the same instruction concurrently, ensuring effective management and processing of the data, especially for uniform operations. To maximize the performance of the thread group, its threads must access data that is adjacent in memory.

Even though threads in the same group execute the same task, this approach may also lead to divergence. Divergence occurs when there are conditional statements present in the instruction set, causing threads to follow different execution paths. This will cause the unit to diverge into different branches. When this happens, a branch will enter the conditional section, causing the others to be idle until the diverged branch has finished the section (ENCCS, 2024). This behavior is illustrated in Figure 2.3.



```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional
 shader code>
```

Figure 2.3: Visual representation of single-instruction multiple-thread. Owner (Fatahalian, 2011)

2.5.3 Block

A block consists of several thread groups and functions as a larger collection of threads that can run independently of other blocks. As a result of this, complex operations can be executed in a parallel manner over greater parts of the memory.

Blocks can also synchronize the threads they contain through barriers. A barrier is a synchronization technique that ensures all threads in a block have reached a certain point in the execution before advancing. This synchronization is necessary for maintaining data consistency and correct execution order among all the threads (ENCCS, 2024).

2.5.4 Cores

GPU cores are the computing units responsible for executing data operations. Each core on a GPU can handle multiple threads simultaneously, making the GPU highly effective for parallel computing. These cores are designed to effectively perform floating point operations, central to graphics rendering and scientific calculations.

Types of cores

There are several different cores within GPU hardware, each designed for a specific computational task. Commonly used in machine learning applications are Compute Unified Device Architecture (CUDA) cores and Tensor cores, both developed by NVIDIA. CUDA cores are versatile, and designed to handle general-purpose computing. Tensor cores, on the other hand, are specialized for matrix operation, making them suitable for machine learning algorithms. These cores sacrifice accuracy for speed as one of many ways to speed up the computation of matrix calculations (Ansorge, 2022, pp. 358–359).

2.5.5 Compute Unit

A Compute Unit is another critical unit of a GPU, functioning as a cluster of GPU cores that concurrently execute operations on data in parallel. Each Compute Unit is comprised of multiple cores, typically optimized for certain types of tasks to be run in parallel, enhancing the GPU's ability to perform complex computations.

In addition to the cores, each Compute Unit is equipped with its own memory resources, including registers and shared memory. This shared memory is essential for providing the threads with rapid data access, significantly reducing latency when compared to accessing the GPU's main memory.

The Compute Units also play a vital role in thread scheduling within the GPU. They manage how and when to run the threads, based on the resources available and the prioritization of the tasks. Each Compute Unit can have several blocks running at the same time, but blocks can not be split among Compute Units. Effective scheduling is essential to ensure high throughput and to balance the load across the GPU (ENCCS, 2024).

2.6 Foundations of Parallel Programming

Parallel programming is the practice of writing code that can execute independent tasks concurrently. In contrast to sequential programming, parallel programming introduces new problems that need to be addressed. These problems are related to concurrency, synchronization, and data sharing across different executions. If these concerns are not addressed the program will behave in unexpected ways. As the nature of each parallel hardware architecture is different, they also have different ways to address these concerns. This section will go into detail on how relevant hardware deals with parallel programming.

2.6.1 Parallel programming on the CPU

When introducing parallelism to CPU-programming, it is important to understand some concepts especially when it comes to shared data across different threads. To address these problems related to shared memory, some tools are introduced. A commonly used strategy to manage problems like race condition is to utilize synchronization variables. Instead of relying on techniques like busy wait, consuming CPU resources while waiting to synchronize one can utilize synchronization variables like mutex locks. These ensure that only one thread can access a critical section of the code at a time, especially read-write operations to shared variables. This synchronization of threads is what makes parallel execution on the CPU predictable (Anderson & Dahlin, 2015, p. 45-96).

2.6.2 Programming on the GPU

It is essential to know several key concepts to understand how the execution on the GPU will behave when writing code for the GPU. Execution is made up of a group of threads forming a block, and a group of blocks making the execution grid. The number of dispatches in each dimension for a shader determines the number of blocks within that execution grid for the respective dimensions. A dispatch is the process of executing the grid of blocks to run the shader on the GPU. The total number of threads for a dispatch is determined by the number of blocks within the execution grid, as visualized in 2.4.

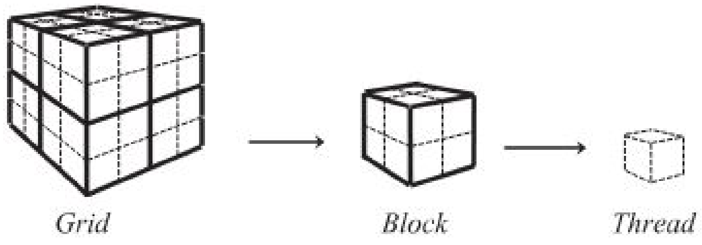


Figure 2.4: The three-level hierarchy of GPU programming. Owner (Navarro et al., 2021)

Each thread possesses its own set of private variables that indicate where it resides within the grid. One variable identifies which block the thread belongs to, and another specifies the position of the thread inside the block. These variables have three dimensions: x, y, and z. Depending on the algorithm, these variables are usually used to index into shared memory, allowing for computations on this memory area. Utilizing the unique combination of these variables can make the execution on shared memory race-safe, as each thread operates on its section of the shared memory.

Depending on the programming language used for GPU programming, there are various methods to synchronize the threads in the execution grid. Among these, synchronization barriers are a commonly used tool. While different types of barriers exist, it is important to note that they can only synchronize at block level. By utilizing the barriers appropriately, threads working on the same memory can be synchronized to avoid a race condition (ENCCS, 2024).

2.6.3 Designing FPGA and ASIC

FPGA and ASIC are not programmable in the same way as the CPU and GPU. Their design process, and the reconfiguration of an FPGA, is typically done with a Hardware description language (HDL). HDL is a computer language used to describe and structure the logic and behavior of electronic circuits. Tools exist for designing integrated circuits with HDL that also offer testing of the designs by simulating their behavior. This simulation is an essential part of the design process (Xilinx Inc., 2024).

The outcome of the design process is what determines the performance of the FPGA or ASIC in parallel processing. Designing a parallel solution for an integrated circuit involves the construction of a data flow that computes in parallel without introducing race conditions. Data races can occur when two rules, defined in the HDL implementation, match the same input data and try to update the same element. Critical errors can be introduced in many ways, as a circuit can be both synchronous and asynchronous. A synchronous circuit uses a clock signal to synchronize changes, while an asynchronous circuit can contain state elements that change at any time (Truong & Hanrahan, 2019). If these issues are addressed, the limit to how parallel one can design the integrated circuit is determined by the available hardware resources.

2.7 Machine learning

The primary objective of this thesis is not to improve an existing machine-learning model. However, it is motivated to address a challenge encountered in a type of machine learning model known as the CNN. These vulnerabilities include noise and adversarial attacks, which can degrade the performance of the model. With this in mind, some general insight into machine learning principles and terminologies will help to understand this thesis.

2.7.1 CNN model

CNN, is a machine learning model primarily used for image recognition. It uses back-propagation to train filters called kernels that are structured as a neural network for classification. A neural network is in simple terms a graph with weights that leads to the guess of the classification. Back-propagation is an algorithm to adjust these weights of the neural network for better classification (Wang et al., 2024).

A CNN is represented in Figure 2.5. The first layer uses 32 kernels to produce 32 new image matrices described through the kernel feature extraction. This is the process of finding the most descriptive attributes of the image.

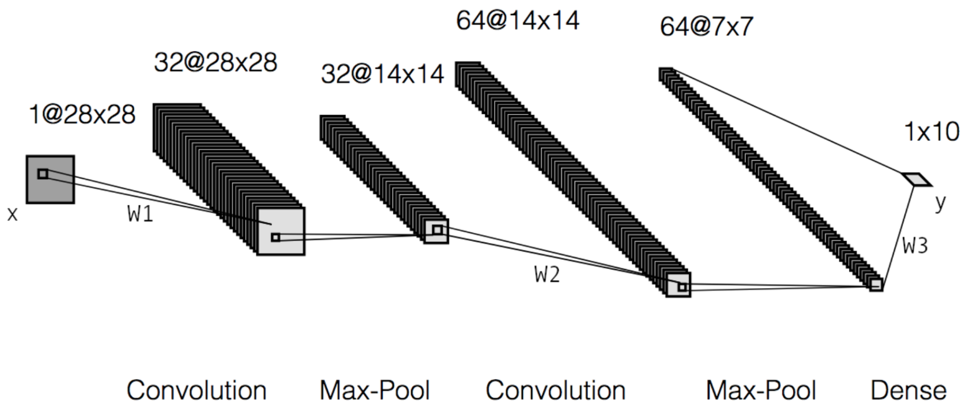


Figure 2.5: A visual representation of a simple CNN model with two convolution layers. The first layer has 32 kernels and the second has 64. Two Max-pool and one dense. Owner (Eidheim, 2023)

The initial kernels in a CNN model are typically trained to identify edges and lines(Stewart, 2019). However, the reliability of CNN models can be compromised by several issues. As mentioned in the introduction, one of these issues is how the model struggles to accurately classify images that contain noise or are subject to adversarial attacks (Momeny et al., 2021). Additionally, the model is criticized for not being biologically plausible.(Eidheim, 2022).

Applying a kernel in a CNN model involves performing the convolution operation over a small portion of the image, and then sliding the kernel over the entire image. The convolution operation involves multiplying the kernel with the pixel values in sub-regions of the image and summing the results. This produces a new matrix of the features, as illustrated below.

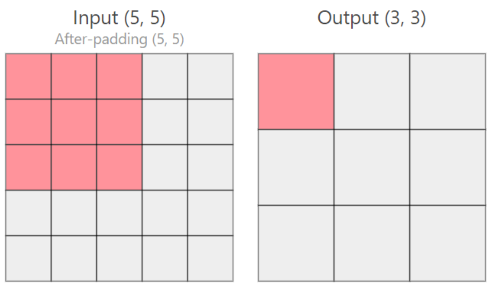


Figure 2.6: CNN dot product with filter in the top left corner of the image. Owner (Wang et al., 2024)

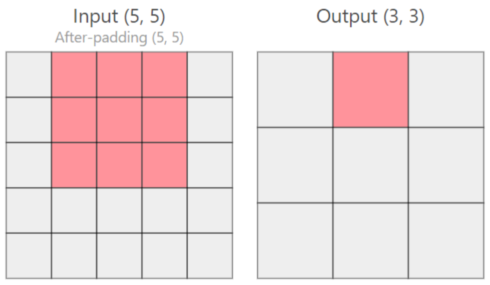


Figure 2.7: CNN dot product with the filter one stride to the right of the image. Owner (Wang et al., 2024)

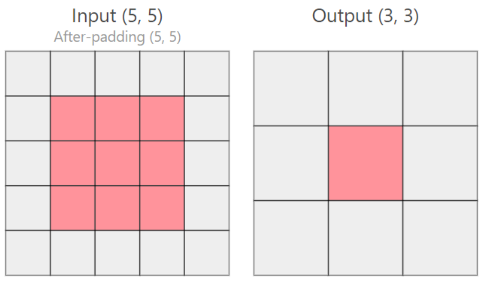


Figure 2.8: CNN dot product with the filter in the center of the image. Owner (Wang et al., 2024)

The feature matrix is then max-pooled. This means that the most descriptive features are stored in a new more dense matrix that reduces the spatial dimension. Max-pooling saves memory and achieves faster training time. This helps against overfitting the classification model and provides a broader perspective. This technique employs a grid that traverses the feature matrix and extracts the highest values as shown in Figure (Zafar et al., 2022). Since the calculations in the CNN do not need to be precise for an accurate model, there can be some deviation from the correct calculation. The key aspect of max-pooling is that the highest value within a region is greater than the others. This prioritizes computational speed over precision, which reduces training time, as f64 requires more clock cycles to calculate than f32 (Gupta et al., 2015).

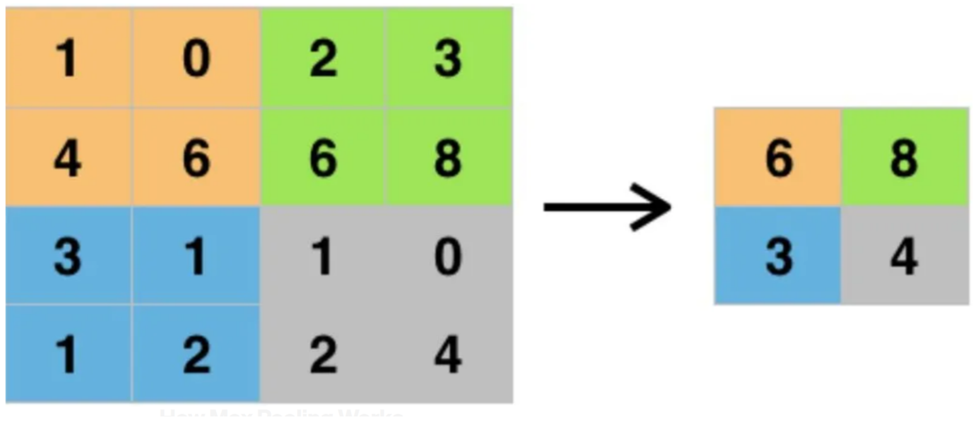


Figure 2.9: Illustration of how max-pooling in CNN is done. Each color represents a grid that selects the highest value Owner (Kılıç, 2023).

2.7.2 Gabor filters

A Gabor filter is a linear filter that excels at detecting edges and textures and is often used for feature extraction. The impulse response of the filter is characterized by the sinusoidal wave convoluted by a Gaussian function, and the filters evaluate the image around a fixed point. Two principal parameters of the Gabor filter are its orientation and wavelength. The orientation parameter affects the sensitivity of the filter to the angles in the image, while the wavelength parameter influences the scale at which the filter operates. The structural and functional capabilities of Gabor filters can resemble edge and width detection in the human visual system. Because of these properties, Gabor filters can be used to emphasize or remove certain features. An image of poor quality, such as one with noise and blurred edge separations, can be improved using Gabor filter (Dakshayani et al., 2022).

A research paper demonstrated that the fixed values of Gabor filters can be used to reduce training time and energy consumption in a CNN model. By replacing the kernels in the first and deeper layers with Gabor filters, the study observed improved training time ranging from 1.4 up to 2.23 times faster, where the performance degradation ranged from 0 to 3% compared to the baseline. Because the Gabor filters are only made once and do not need to be trained, time is saved by not adjusting the values of the kernels that consist of Gabor filters (Sarwar et al., 2017).

2.7.3 Gabor-like filters

At NTNU, a new type of filter has been developed that shares similarities with the Gabor filters. These filters are designed for detecting edges and simple shapes and can adjust their frequency and orientation to register different types of shapes. Derived from a mathematical formula, these filters do not need to be trained and only need to be generated once, similar to Gabor filters. The implementation of these filters in image recognition models can reduce the total time needed for training the model (Devakumar & Eidheim, 2024).

2.7.4 Photoreceptors

In the retina, the part of the eye responsible for registering light, there are two types of photoreceptors: cones and rods. Rods are light-sensitive nerves that are good at detecting shapes and movement, with about 125 million rods in each eye. Cones, of which there are about six million in each eye, are less sensitive but register the frequency of color (University of Oslo, Institute of Biosciences, 2019). Together, a total of 130 million photoreceptors work simultaneously to register light and convert it into electrical impulses sent to the visual cortex of the brain, and interoperated into vision. The visual field of the eye is strongest at the fixation point and diminishes with distance, but can still recognize larger objects (Høvding & Sandvig, 2020).

An optimal filter bank of Gabor filters has proven to be a good representation of the visual cortex (Daugman, 1985). This relates to the Gabor-like filter, which is based on the same principles. The new Gabor-like filters would simulate the connection between the photoreceptors in the human eye and the visual cortex because they can register many frequencies and work independently from each other.

A proof of concept created in a group project in a machine learning course at NTNU indicated that using Gabor filters can help with the classification of images with noise, Appendix A. They used Gabor filters instead of the new Gabor-like filter. However, the Gabor-like filter can be better suited for classification and resilience to noise, and the calculations for applying these filters are the same. These filters are applied to the entire image, primarily registering the center but also capturing strong features at the edges. In contrast, the filters of a CNN model register features in a small area at the time, as previously illustrated.

One benefit of this proof of concept is its flat structure, consisting of a single layer. All the filters can be applied to all images simultaneously.

The result consisting of all the features can then be max-pooled independently in parallel. This is an improvement over the training time for a CNN model where the image has to go through each layer and then undergo backpropagation. CNN models, such as AlexNet. have a limited number of kernels for feature extraction in each layer, with AlexNet having 96 kernels in the first layer (Khandelwal, 2020). A more biologically plausible model would require more filters than the current CNN models to simulate the millions of photoreceptors and visual cortex effectively.

2.7.5 Cosine Similarity

The Gabor-like filter consists of a two-dimensional matrix with size $X \times X$ containing complex numbers (real + imaginary). To apply the filter onto an image the filter matrix performs element-wise multiplication with the image values, as illustrated below

$$\text{Image} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \text{and} \quad \text{Absolute values} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$\text{Image} \times \text{Absolute values} = \begin{bmatrix} a_{11} \times b_{11} & a_{12} \times b_{12} \\ a_{21} \times b_{21} & a_{22} \times b_{22} \end{bmatrix} = \text{Filtered Image}$$

The matrix A is the image, while the matrix B represents the absolute values of the filter's complex numbers. The element-wise multiplication of these matrices produces a new matrix with the same dimensions as the image, describing the image through the filter.

For information retrieval, cosine similarity is used to measure how well the filter fits the image, as it indicates how similar the vectors are. Two vectors with the same direction have 0 degrees between them, resulting in $\cos(0) = 1$, describing a strong similarity between the two vectors. The formula for this is:

$$\cos\theta = \frac{\sum_{i=1}^{d} FilteredImage_i \times real_i}{\sqrt{\sum_{i=1}^{d} FilteredImage_i^2} \times \sqrt{\sum_{i=1}^{d} real_i^2}} \tag{2.1}$$

Where $\mathbf{FilteredImage}$ is the filtered image flattened, and $\mathbf{real}$ is the flattened version of the filter's real parts.

The calculation of the cosine similarity value is completely independent of each other. This means every picture with every filter can be calculated at the same time in any order. This makes it completely parallel and has the potential for computation with highly parallel hardware.

2.8 Data analysis

This project has resulted in a dataset containing a significant collection of data points of the processing speed of the experiments, presented in detail in Section 3.5.2. While individual data points may lack inherent visual appeal, improving data readability and facilitating comprehensive visualization and analysis is essential. Various techniques will be used on the collected data to achieve this goal. These techniques are explained in the sections below.

2.8.1 Standard deviation

Standard deviation is a way to measure the amount of variation for a given variable, and in this thesis, the variation in computation time across individual runs with the same set of instructions. A low standard deviation indicates that the values are usually close to the mean, while a high standard deviation indicates greater dispersion from the mean. The standard deviation is represented by the symbol σ and calculated by the equation below, where μ is the mean, x_i is each data point and N is the number of data points (Navidi, 2011, pp. 96–98):

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}}$$

2.8.2 Interpolation

Interpolation is the mathematical technique to estimate values between data points. It achieves this by constructing a function, $\boldsymbol{Q}$, that passes through the distinct data points, $(x_0, y_0) < (x_1, y_1) < \ldots < (x_n, y_n)$, in such a way that the interpolation requirements are satisfied (Levy, 2012).

$$\boldsymbol{Q}(x_j) = y_j, \quad 0 \leq j \leq n$$

An interpolant, denoted by the function $\boldsymbol{Q}$, is any function, linear, polynomial, or a spline, that satisfies the specified requirements.

2.9 Related work

In this section, articles with relevant topics to this thesis will be presented. This will provide a good background for other perspectives and findings related to the topics discussed in this thesis. The section will also draw parallels and distinctions between the articles and this thesis.

2.9.1 Accelerating Robot Dynamics Gradients on a CPU, GPU, and FPGA

Similar to the objectives of this thesis, the referenced article (plancherb1, 2024), explores methods to accelerate the execution of a specific algorithm tailored to different hardware, comparing the computation times of the implementations. In contrast to this thesis, the algorithm implemented in the article is an implementation of the gradient of rigid body dynamics. A review of the source code of the article reveals that the algorithms share similarities in their mathematical operations.

The article presents the design of three implementations tailored for three different computing platforms, the CPU, the GPU, and the FPGA. The article concludes that both the GPU and FPGA outperform the optimized CPU, achieving up to three times the speedup. Another interesting find by their experiments is that the FPGA seems to outperform the GPU at low numbers of parallel computations, while the GPU scales better to higher numbers of parallel computations (Plancher et al., 2021).

2.9.2 A comparative evaluation of the GPU vs. the CPU for parallelization of evolutionary algorithms through multiple independent runs

Another relevant article is (Syberfeldt & Ekblom, 2017), which compares the general performance, not hardware specific, of the CPU and the GPU by looking at multiple independent runs. The article tries to provide insight into which computational hardware is most efficient. It differs from this thesis in how the hardware is compared. While this thesis compares the performance of the CPU and the GPU using a specific, highly parallelizable case, the article takes a more general approach with different scenarios. Both the article and this thesis use several independent runs to determine the performance of the hardware. The article concludes that while the GPU is powerful, it is not always the best option for parallelization, as the CPU outperforms the GPU in certain scenarios.

2.9.3 A Parallel ASIC Architecture for Efficient Fractal Image Coding

The relevancy of this article is related to its focus on a parallel ASIC architecture. Instead of processing images to extract an image’s features, this article focuses on encoding the images using quad-tree fractal coding as the method. Fractal encoding, despite its advantages, is not used that much because of the slow and complex nature of the compression algorithm. Because of this, fractal coding is typically used as an archival strategy, where compression is performed once and decompressed multiple times. This is the problem that the article tries to solve by utilizing the highly parallelizable property of the compression algorithm.

The article presents an ASIC architecture for quad-tree fractal coding that compresses images in near real-time. This is achieved by designing a parallel architecture and optimizing the compression algorithm for parallel computation. The results found in the article show a near real-time encoding of grayscaled images as large as 256 x 256 pixels using the parallelized ASIC design (Acken et al., 1998).

2.9.4 Design of a Gabor Filter-Based Image Denoising Hardware Model

This article starts by introducing the problem with noise in images from methods such as ultrasound and CT scans, suggesting denoising these images will make the analysis of the images more efficient. To achieve this denoising of the images, Gabor filters are presented as a solution for feature extraction. The reason that Gabor filters are used is that the frequency and orientation expression of the filters are identical to the human visual system.

The article then proceeds to present a design for an image-denoising hardware accelerator based on Gabor filters. The results presented in the article suggest good performance on edge detection on images with noise. The limitations of the design are related to on-chip memories, as the input images are buffered when edge detection is performed. The article then concludes by suggesting that the proposed design can be integrated into an FPGA for further improvements (Dakshayani et al., 2022).

In contrast to this thesis which looks at the parallelization of image processing methods, this article focuses on the actual problem of image processing. The similarities between the article and this thesis come out in the proposed design of a hardware architecture of the Gabor filter module. Both articles present the potential gains from implementing such methods on an integrated circuit such as FPGA.

3. Method

In this section, the different research methods used during the study will be explored. Research methods often reference the strategies, processes, and techniques utilized to gather the data used for further analysis to uncover new knowledge or to obtain a deeper knowledge of a subject (University of Newcastle Library, 2024). Therefore, the choice of research method is an important matter, as it will deeply affect the validity and reliability of the findings of the study. In the following sections, there will be given a thorough review and explanation of the methodical choices made throughout the project period.

3.1 Theoretical Foundation

Due to the inherently theoretical nature of the thesis, before any programming was initiated, a comprehensive review of the literature and academic reports was conducted to establish a robust theoretical foundation for all members of the group. This involved extensive research on computer fundamentals, GPU programming, FPGA and ASIC, publications addressing the potential efficiency gain of parallel computing, and advancements in the field of GPU technology.

Throughout the search for information, the scope had to be carefully adjusted to ensure both theoretical and technological findings were valid and up to date. For this, the group utilized both academic literature and academic search engines and databases, such as IEEE Xplore. Considering the rapid enhancements in GPU technology in recent years it was also important for the group to make sure any source of information was fairly recent to ensure its validity.

Through this approach, the group developed a deeper understanding of the theory on how the GPU can be utilized for complex computational demanding tasks. Information from detailed studies conducted on parallel computing, memory management, and algorithm optimization provided the group with greater knowledge on the matter. This strategy helped the group grasp the concepts and underlying principles related to potential performance gain on the GPU compared to the CPU. It enlightened the group on how the machine learning algorithms can fully utilize the power of the GPU.

3.2 Research Methodology

When aiming to enhance machine learning methods by utilizing parallelization, one faces a complex challenge where multiple factors come into play simultaneously. Therefore, a robust research methodology is required to ensure the provided solution is not only efficient but also reliable and scalable.

This project utilizes the Design Science Research (DSR) framework, which forms the connection between scientific research and system development. DSR is a research methodology where the development of a technological artifacts is considered a part of the knowledge development process. While expanding on the knowledge about the development process is important, DSR also aims to generate knowledge through the development process itself (Brocke et al., 2020).

3.2.1 Design Science Research

DSR is a problem-solving paradigm that focuses on facing real-world issues, while at the same time developing knowledge through innovative development. It emphasizes the creation of new solutions, ranging from ideas and models to practical implementations.

The primary objective of DSR is to give insight that explains exactly how and why the developed solutions work in their respective environments. This form of research method contributes to the understanding of the practical application of the implemented solution, and how this insight may inspire innovations.

For this project, with the enhancement of machine learning methods in focus, the DSR method provided an appropriate framework. The structure of DSR did not only support the complexity of the project but also enforced the iterative nature of the technological development. By adhering to DSR guidelines, this thesis aims to make practical contributions through the development of enhanced machine learning methods, and theoretical contributions by expanding knowledge on programming and parallelization techniques on the GPU (Brocke et al., 2020).

3.2.2 Adaption to DSR

The DSR research methodology systematically addresses the problem statement through six steps, aiming to develop new solutions, while at the same time contributing to scientific knowledge.

In the initial phase of the project, a more refined problem statement was derived from the task description. This problem statement laid the foundation for the subsequent research. From this detailed problem statement, specific research challenges were identified and addressed using the DSR methodology.

Following the refinement of the problem statement, the DSR methodology aims to extract research problems from this statement. These are essentially smaller, more manageable parts of the problem the project aims to solve. This segmentation ensures a focused approach, ensuring each problem is systematically handled through the DSR process. Initially, the group extracted a set of research problems, which evolved and expanded as the project progressed. This will be explained in detail through the six steps of the DSR methodology:

1. **Identify the research problem:** Through a thorough analysis of the problem statement, the research problems were identified. This helped find the core issues and critical aspects of the project. This approach ensured that the research objectives were aligned with the real-world needs of the problem statement.
2. **Define Objectives of a Solution:** Objectives were derived from the refined problem statement and specific research problems, considering what could realistically be achieved throughout the project period. Each artifact had a set of defined objectives to outline its theoretical capabilities.
3. **Design and Development:** This involves creating an artifact, which is a constructed solution, specifically designed to address the research problem An artifact can be in any shape or form, ranging from conceptual ideas to specific code implementations, or any other format that would be beneficial to the research.
4. **Demonstration:** The artifact is tested using mock data in a simulated scenario to demonstrate its practical applicability and functionality. This is an important step, as the initial design of an artifact is rarely perfect, and it is therefore often important to take a step back and iterate on the design based on the demonstration.
5. **Evaluation:** The artifact is assessed for efficiency, functionality, and its ability to solve the research problem. This also involves testing the artifact in a simulated test scenario, focusing on evaluating its performance and usability.
6. **Communication:** The final step involves documenting and presenting the knowledge obtained and the practical implications of the developed artifact to both academic readers and potential stakeholders.

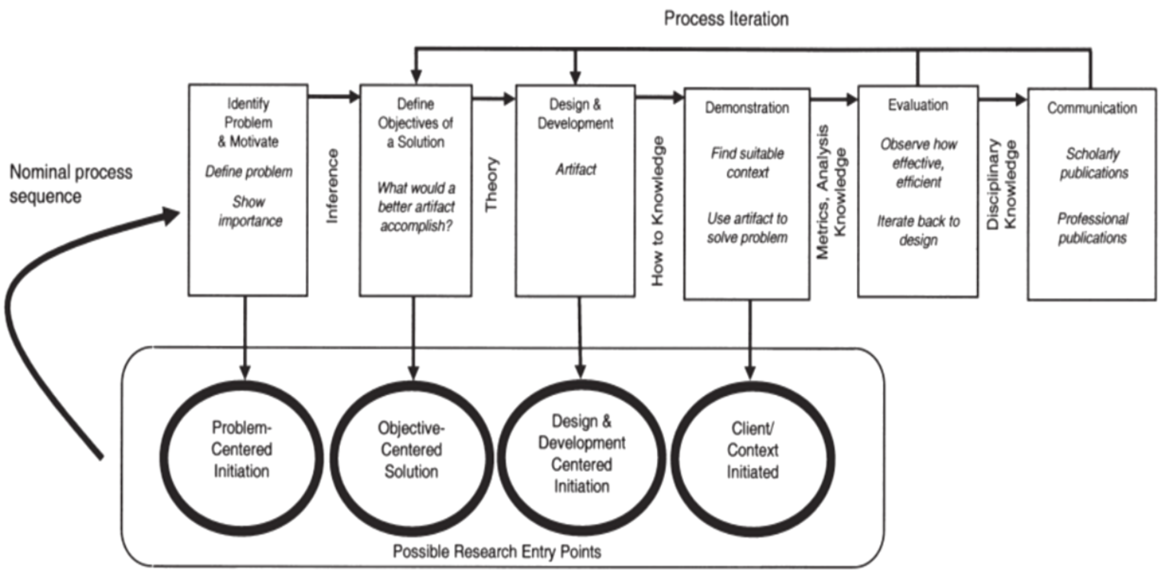

Process Iteration
Nominal process sequence
Identify Problem & Motivate
Define problem
Show importance
Inference
Define Objectives of a Solution
What would a better artifact accomplish?
Theory
Design & Development
Artifact
How to Knowledge
Demonstration
Find suitable context
Use artifact to solve problem
Metrics, Analysis Knowledge
Evaluation
Observe how effective, efficient
Iterate back to design
Disciplinary Knowledge
Communication
Scholarly publications
Professional publications
Problem-Centered Initiation
Objective-Centered Solution
Design & Development Centered Initiation
Client/ Context Initiated
Possible Research Entry Points

Figure 3.1: DSR Process Sequence. Owner (Brocke et al., 2020)

Reiteration is fundamental in the DSR methodology, as it is common to revisit and refine a prior step throughout the stages of the process. Feedback and new insights lead to enhancements in the artifact, adjustments of the objectives, or even redefinition of the research problem, ensuring the solution remains in line with the evolving understanding of the research problem (Brocke et al., 2020).

3.2.3 Evaluation in DSR

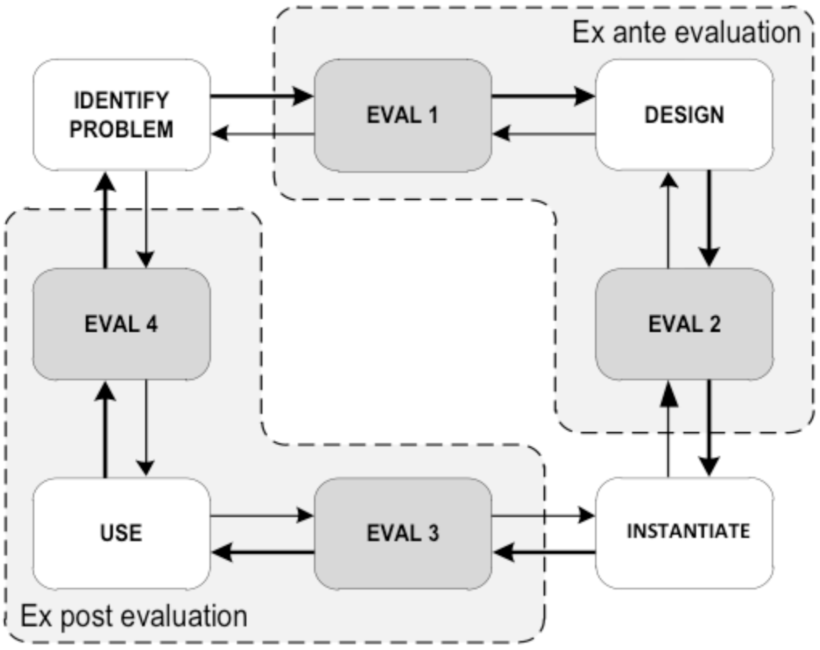


Figure 3.2: Illustration of Evaluation Activities within DSR. Owner (Brocke et al., 2020)

Figure 3.2, found in (Brocke et al., 2020), illustrates four evaluation types within DSR. These are used to ensure that the artifacts developed by the group meet both the theoretical and practical applications to assist in solving the problem.

The first type is the ex-ante evaluation of problem identification. This ensures that the research problem is well-defined and aligns with the research goals before any work is initiated, setting the foundation for the design process.

The second type is the evaluation of the design, which occurs after the design phase of the artifact. It examines whether the proposed design is likely to solve the identified problem, assessing its theoretical foundation and alignment with the objectives set for the artifact.

The third type is the evaluation of instantiation, conducted after the implementation of the artifact. The functionality of the artifact is then tested in a fixed environment to ensure it's functioning as expected.

The final type is the ex-post evaluation of use, performed once the artifact is in use in a real-case scenario. This evaluation is important as it gives insight into how well the artifact solves the initial problem in its intended environment.

These evaluations reflect the iterative nature of the research process during the project period, which has been comprised of a series of development cycles. Each cycle involved adapting this evaluation style to understand the efficiency and relevance of each artifact to the Assignment Details (Brocke et al., 2020).

3.2.4 Integration of DSR

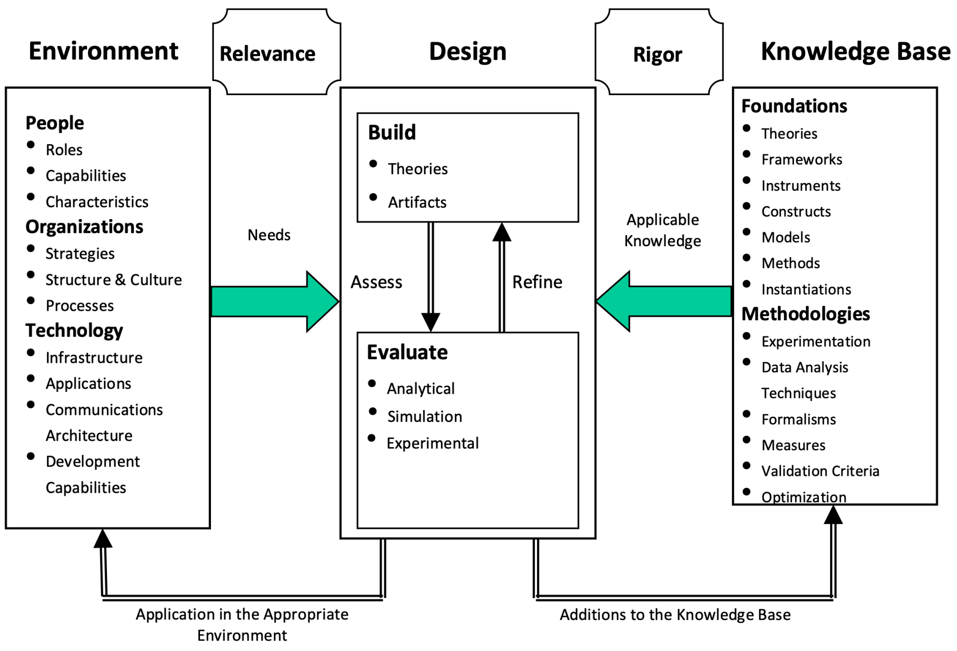


Figure 3.3: Design Science Framework for the project. Owner (Brocke et al., 2020)

Figure 3.3, adapted from (Brocke et al., 2020) to fit the group's development process, illustrates how the group adapted the DSR framework to meet project needs, emphasizing the iterative process and feedback mechanism.

The process began by analyzing the environment where the solution would be implemented. This involved an assessment of the roles, knowledge, and capabilities of the group members, as well as the group's strategy and culture. This thorough understanding of theoretical and practical needs prepared the group for the challenges ahead.

The knowledge base established from the initial research laid the groundwork for the first design phase, initiating the cyclic approach of the DSR methodology. Here. artifactss were developed based on both the theory obtained and the problem statement. Each artifact then underwent simulated runs and experimental evaluations to assess their efficiency and functionality. This played a crucial role in enhancing the overall performance of an artifact ensuring continuous improvement through feedback during the cyclic approach (Brocke et al., 2020).

3.3 Choice of Technology

Choosing appropriate technologies is essential for the success of any project, requiring careful consideration of trends, compatibility, and scalability. This section outlines the technological tools and frameworks used, with a focus on their contribution to the project goals and their impact on performance, reliability, and usability. The reasons for these decisions are detailed in the following subsections, giving an evaluation of their impact on project outcomes in subsequent sections.

3.3.1 Administrative technology

For most of the administrative parts that needed coordination in this project, we decided to use Microsoft 365 due to its comprehensive selection of software and tools. Microsoft Teams was used to store all administrative documents and to schedule meetings with the group's supervisor. Microsoft Word was utilized for writing all administrative documents, except for this thesis. Furthermore, we used Microsoft Excel to manage the group's time sheets to track time spent on the project.

During the period Discord was used as the group's primary communication channel among group members. Discord is an online platform that allows for communication over voice, video, and text. It was chosen over Teams due to the group members being more experienced with Discord, making it the more suitable communication platform.

3.3.2 Technology utilized during development

In the development of the project, various technologies were used to enhance efficiency, collaboration, and project management. This section details the primary tools and systems used throughout the development process.

Integrated Development Environment

For the project, Visual Studio Code (VS Code) was utilized as the primary Integrated Development Environment (IDE). VS Code is a lightweight and powerful IDE and was chosen due to its support of a wide range of programming languages.

Version Control

To manage and collaborate on the codebase, the group used Git as the version control system. Known for its robustness and flexibility, Git makes it possible to track changes and branch out to implement new features seamlessly. This simplifies group development, as it prevents conflicts within the main codebase.

To host the project in an online repository for easy access for all members, the group used GitHub. This allowed for an efficient development process as all members had access to the latest version of the codebase at all times.

3.3.3 Rust

Rust was chosen as the primary programming language for this thesis due to its system-level programming capabilities. Additionally, its recent advancements in GPU programming, despite being in an early phase, made it a suitable choice. This choice aligned with the thesis' objective of parallelization of machine learning methods using system-level programming languages.

Initially, CPU implementations of the algorithms were developed to establish a baseline for performance comparison. This implementation used Rayon, a data-parallelism library that guarantees data-race-free execution by converting sequential computations into parallel (Nikolai Vazquez and Josh Stone, 2024).

In addition to the CPU implementation, a GPU-accelerated implementation was also created, using the wgpu library. The wgpu is a graphics library based on the WebGPU API (wgpu Team, 2024), designed to provide high-performance, cross-platform graphics and computing capabilities. To effectively utilize the resources of the GPU, the WebGPU Shading Language (WGSL) was used.

WebGPU Shading Language

WGSL is the shading language for WebGPU used to write shaders that run on the GPU. WGSL is an imperative programming language that has a sequential execution flow. It is essential for tasks that require parallel processing utilizing WebGPU.

In this project, WGSL is utilized to manage and execute command pipelines efficiently. Through dispatch commands, which serve as triggers, the execution of the specific pipeline stages associated with a specific shader is initiated. This allows for defining shaders that handle complex data processing tasks (World Wide Web Consortium (W3C)), 2023).

3.3.4 Python

Python was used to develop an implementation of the application of the Gabor-like filters, serving as benchmark for comparison with other CPU implementations. Python was chosen because of libraries like NumPy, which were leveraged to simplify and optimize the application of the filters. By comparing the performance of Python with Rust, valuable insights could be gained into the trade-offs between high-level language simplicity and system-level programming efficiency.

3.3.5 C++

The C++ implementation was developed to challenge the Rust implementation. C++ was chosen due to its maturity as a programming language, having been around for decades, and its extensive documentation on CUDA programming. Its long history in the programming community made it a reliable and well-suited option for comparison.

The C++ version was designed to mirror the Rust implementation closely. By being based solely on the Rust version, the goal was to evaluate and compare how much data each implementation could handle at a time and the total processing time required. This comparison aimed to highlight the strengths and weaknesses of both languages in terms of performance and efficiency in handling the computations of large sets of data.

3.3.6 Testing environment

For testing purposes, two environments were selected to ensure a comprehensive evaluation. One testing platform was provided by a group member. Additionally, the group had access to the NTNU's IDUN High-performance computing (HPC) cluster, so it was decided to use a node from this cluster as the second testing platform. This allowed for a robust comparison of performance across different hardware configurations.

Specification \ Platform	Platform 1	Platform 2
CPU	i7-12700k	Xeon(R) Gold 6248R
Cores	8 Performance (P) 4 Efficient (E)	16 out of 24
Max clock (GHz)	4.9 (P) 3.8 (E)	4000
Base clock (GHz)	3.6 (P) 2.7 (E)	3.0
Threads	20	16
Memory (GB)	32	1400
GPU	RTX 3080	A100
Boost clock (MHz)	1710	1410
Base clock (MHz)	1440	765
Memory clock (MHz)	1188	1215
Memory Size (GB)	10	40
Memory Type	GDDR6X	HBM2e
Memory Bus (bit)	320	5120
Bandwidth (GB/s)	760.3	1560
CUDA Cores	8704	6912
Tensor Cores	272	432
Compute Units	68	108
CUDA Cores per Compute Unit	128	64
Tensor Cores per Compute Unit	4	4

Table 3.1: Summary of Hardware Specifications

3.4 Development

The application of Gabor-like filters is highly parallel and has significant potential for image classification. An algorithm for applying the filters in parallel is desirable due to the large amount of data that must be processed during training. An efficient algorithm on the GPU for parallel filter application can significantly shorten training time.

During the development of several algorithms, the team used the principles of the lean model, as outlined in the book ”Lean Thinking” by Womack and Jones from 2003 (Rolstadås, 2022). Although there are various interpretations of the lean methodology, the team followed the following principles:

1. Define customer value – value is always defined based on customer needs and is itself the core of the lean philosophy.
2. Map the value stream – identify all activities products must go through to completion.
3. Create flow - adapt the production processes so that the value flow is as flexible as possible.
4. Use a management model based on suction - Kanban management.
5. Strive for perfection - establish and plan for continuous improvement.

Since this is a research project there is no final product and no customer. However, the value is in the result of the project and representing the results in a structured manner so that others can use the group’s findings for future research and development. The result stems from comparing the methods that exist today with the group’s developed algorithms on the GPU and possible solutions in the future.

The lean methodology emphasizes learning from one’s mistakes through trial and error, which complements the DSR approach effectively. Given the team had no prior experience programming on the GPU there have been a lot of errors and bugs to correct and the team has quickly learned from their mistakes and wrote several shaders to test different solutions. Due to the limited time of the project, the team focused only on tasks directly related to the task description to collect quality data. The group developed a structure that can be used for classification purposes, though not the classification itself. One of the main objectives was to test performance for optimal throughput, leading to an iterative process of coding, testing, and rewriting to improve throughput.

Effective communication was crucial for maintaining good structure. The group worked together physically and used Discord for informal communication, Teams for storing important documents and planning meetings, and Git for version control of the code to ensure consistency. This clear structure facilitated task prioritization and made it easy for team members to assist one another.

A general plan was made in January and visualized in a Gantt chart, see Appendix B. Since the group had no prior experience with GPGPU, the chart was a rough estimate expected to have errors. This plan consistently changed based on the results and workflow of the group, guided by the feedback of the supervisor. As new priorities came along, tasks with little to no value to the end goals were eliminated.

3.4.1 Images and filters

In the performance analysis of the different implementations, realistic data was needed to get an accurate evaluation of the performance. The images used for development and testing throughout this thesis stem from the MNIST dataset (Deng, 2012). The filters were generated utilizing the algorithms presented in the research conducted at NTNU, as described in (Devakumar & Eidheim, 2024). These filters are designed to have an odd height and width dimension, to ensure there is a center in the filter generated.

The size of the images from the MNIST dataset is 28×28 in size. To ensure compatibility between the images and the filters, the MNIST images needed to be zero-padded. Therefore, the size has been 29×29 for both the images and filters throughout the project.

The images and filters generated are reused once they exceed a specified amount. Image reuse occurs when the number of images reaches a certain threshold, which varies by implementation. Filter reuse happens when the number of filters exceeds 1,000. This approach focuses on reducing the data processing time of each run. It is important to distinguish between the data processing time, which involves data loading, and the actual processing time dedicated to applying filters. The primary focus of this thesis is not related to the result of the image processing itself, but the processing time required for the processing of the images.

3.4.2 Parallelization strategies

Since the group had no experience utilizing the GPU for general-purpose computing, the first step was to research the wgpu library and successfully execute simple programs on the GPU. Subsequent steps involved more research on GPGPU for better insight and inspiration on different approaches. This utilization of DSR can be reflected in the pattern of research and development for the rest of the project.

The first implementations were created in the Rust programming language using the wgpu to execute the shader code on the GPU. Later on, the group translated one shader into C++ using CUDA.

3.4.3 GPU

During the development phase, various algorithms utilizing the GPU were created to test different strategies using the computational power of the GPU. These strategies varied based on the volume of data transferred between the CPU and the GPU, and the workload assigned to each thread.

One optimization involved minimizing the steps required to calculate the cosine similarity. The filters were normalized before being transferred to the GPU, which is reasonable as the real part of the filter remains constant for every image and is not affected by the value of the image, unlike the absolute part of the filter.

This simplifies the cosine similarity calculation to:

$$\cos\theta = \frac{\sum_{i=1}^{d} \text{Filtered Image}_i \times \frac{\text{real}_i}{\sqrt{\sum_{i=1}^{d} \text{real}_i^2}}}{\sqrt{\sum_{i=1}^{d} \text{Filtered Image}_i^2}} \tag{3.1}$$

Since the normalization of the real part is done during the filter generation, the normalized real filters are just like any other matrix with real values in it. This reduces the amount of cycles needed to do the calculation on the GPU.

The formula can be as shown below:

$$\cos\theta = \frac{\sum_{i=1}^{d} \text{Filtered Image}_i \times \text{normalized Real}_i}{\sqrt{\sum_{i=1}^{d} \text{Filtered Image}_i^2}} \tag{3.2}$$

In the context of image classification, it's not necessary to use all of the values generated from the cosine similarities between an image and the filters. The results from the cosine similarity are therefore max pooled to the highest value of those filters. Since the classification of images is not the focus of this thesis, a constant number of elements that is max pooled was selected. It was decided on a max pool chunk of 500 as this choice strikes a balance of highlighting the performance implications of the max pooling operation and compatibility between the number of images and filters involved in the analysis.

Cosine Similarity Shader Strategy

To evaluate how memory transfers impacted the total processing time of the calculation of the cosine similarity operations, two strategies were used: ”One image and all filters” and ”All images and all filters”.

For ”One image and all filters”, during the initialization, all required buffers are allocated and the filters are transferred to the memory of the GPU. The filters remain on the GPU for the entirety of the calculation process. Following the initialization, one image is transferred at a time to the GPU, to perform the cosine similiary. Then when it is done, the results are returned to the CPU. Due to the approach requiring a for-loop to transfer the images from the CPU to the GPU, it is not fully parallel. However, once the image is transferred to the GPU, the operations performed are parallel.

As for ”All images and all filters”, all images and filters are transferred to the GPU during initialization. It is therefore using more of the memory of the GPU, but reducing the number of memory transfers compared to the one-image-all-filters strategy. This approach executes in a more parallel fashion on the GPU as all the operations run concurrently for each image. There were implemented three types of shaders for all images and all filters strategy, where one of them was a continuation of the One-image-all-filters shader strategy.

The strategy of one image and one filter seemed redundant based on the numerous memory transfers and the gain in computational time would be insignificant since the calculation for one image and one filter is not computationally demanding. It would only parallelize the calculation of the cosine similarity on the GPU where multiple threads work on the same cosine similarity and not run them simultaneously. Because it would have to call it in a double for loop, first for the images then for the filter.

The second consideration is how much work should one thread do. Is it best to let multiple threads work on the same memory and have synchronization or let the thread work with it is own memory space with no synchronization?

One Image and All Filters

The one-image-all-filters strategy has a dispatch in only x dimension that equals the number of filters. Each execution block has access to one image data and one filter. To add the two together and get the cosine similarity two while-loops are implemented, one reduces the data and stores it in a temporary buffer, and the second while-loop sums the temporary buffer to a single value that is then stored in the output buffer. Both while-loops use barriers to synchronize the instruction between the threads that read from the same memory addresses.

Further development on the one-image-all-filters strategy made it possible to transfer all the images and all of the filters at the same time and then reduce the total number of data transfers. This would work similarly to the previous implementation, where the main difference is the dispatch. It is in both x and y dimensions, where x equals the number of filters and y the number of images. This made it possible to add an offset to select the image and the correct filter to apply. One execution block is responsible for calculating one cosine similarity, meaning synchronization between threads was still necessary.

All Images and All Filters

Three shaders have been developed to optimize cosine similarity calculations with unique thread and block distribution strategies on the GPU. They are named based on how they distribute workload within a block. The for-loop shader, lacking parallelization between threads within a block; the optimized one-image-all-filters or while-loop shader, detailed previously; and the parallel shader, distributing the workload across threads in a block. The for-loop and the parallel shader will be described in more detail below.

The for-loop shader dispatches in x and y dimensions, where the x dimension is for identifying filters and y for the images. The amount of threads spawned inside the execution block decides how many cosine similarity products the execution block can calculate. Each thread runs a for-loop that performs the entire calculation and does not branch out or sleep. There is no need for a temporary buffer or thread synchronization Since each thread only works on its section of memory. Later in the project, this shader was copied from Rust to C++ and ran using CUDA. This enables the testing of not only logic but also different language's performance and its optimization on the GPU.

The third shader, referred to as the parallel shader, dispatches in x and y dimensions, with the x dimension for images and the y dimension for filters. The shader calculates three cosine similarities per execution block, meaning the dispatch in the y dimension is the number of filters divided by three. The shader divides the amount of work per thread using a for loop. Each thread has a start and a stop index on the image matrix, iterating over the values with the corresponding filter value. The thread then stores this value in a temporary buffer. When the buffer is full, one thread sums up the values in the temp buffer to get the complete cosine similarity. Due to the temporary buffer, a barrier is required for synchronization among the threads in the same execution block. With this strategy, each thread can perform more work, and each execution block can do more in total compared to the other while-loop implantation that only calculates one cosine similarity per execution block.

Max pool strategies

The strategy related to the max pooling of the cosine similarities results is also split into two: a parallel and a looped shader. All of the different shader strategies above are combined with the simpler looped max pooling shader except the third shader explained above.

Looped max pool shader

The looped shader works similarly to the looped cosine similarity shader explained in Section 3.4.3. The only difference here is that the shader only dispatches in x dimension as the shader only needs to work in one dimension to max-pool. Here each thread runs the whole computation of reducing the cosine similarity values in a chunk of 500 down to the largest value. Just like the other one, the looped max pool shader doesn't need to store temporary values in a buffer or synchronize the threads because one thread does the whole calculation.

Parallel max pool shader

The third cosine similarity shader explained in Section 3.4.3, is the inspiration for the parallel max pooling shader. It works similarly but only in one dimension instead of two. Each computation block is responsible for max pooling five chunks compared to the looped version where each block computes 256 chunks.

All of the different shaders make it possible to test for different bottlenecks of GPGPU programming. What affects the processing time? Is it the data transfers, the number of blocks, the number of barriers for synchronization or is it the how much one thread does? The table below provides a summary of the different implementations.

Summerizing the sheer strats

Name	**HW**	**Method**	**Data transfers**	**Threads per CS**	**CS per Block**	**Block barriers**
Python	CPU	SEC	no transfer	one thread does all calculations	–	—
Rust	CPU	SEC	no transfer	one thread does all calculations	–	–
Rust-par	CPU	PAR	no transfer	Rayon handles the distribution on the cpu	–	Rayon handles race condition
Rust-one-image	GPU	PAR	one image all filters	256	1	5
Rust-opt-one-img	GPU	PAR	all images all filters	64	1	2
Rust-parallel	GPU	PAR	all images all filters	85	3	1
Rust-loop	GPU	PAR	all images all filters	1	256	0
Rust-loop-opt	GPU	PAR	all images all filters	1	64	0
C++-loop	GPU	PAR	all images all filters	1	256	0

Table 3.2: A table of all the implementations and their shader strategies. **HW** = Hardware, **CS** = Cosine similarity

3.4.4 CPU

The CPU implementations were developed for two primary reasons. Firstly, the process of applying one filter to one image is relatively straightforward, facilitating sequential calculations with accurate cosine similarities. These calculations served as benchmarks for the GPU implementations. Secondly, the CPU implementations allowed for performance benchmarking across different programming languages, providing insights into both sequential and parallel processing times.

The languages utilized are Python with the NumPy library, rust, and C++. Rust included a parallel implementation using the Rayon library, which evenly distributes the workload across all CPU cores. Both Python and C++ are well-established in the field of machine learning, making them suitable for comparison against the current Rust implementations. This comparison aims to evaluate the traditional performance and efficiency of Rust relative to these traditional machine-learning languages.

3.4.5 Testing

Since the group is working with technology that they have not worked with before, it is important to test and verify the different methods and strategies. To achieve this the group utilized testing. As mentioned previously in Section 3.4.4, the CPU implementation was used to calculate the expected result for the other implementations. By doing this, the group could confirm that the different shader designs worked as intended by producing the correct results. The method used for these tests where integration testing as the tested result is from the whole computational process of processing the images, max pooling the processed results and then retrieving the result back to the CPU from GPU memory.

It is difficult to both test and debug the behavior of a shader since shaders run on the GPU. Because of that, there is no way of running unit tests for the shaders. This is one of the many reasons for utilizing integration tests as the testing method. Another reason is that integration tests can test the whole process as one combined unit validating that the different shaders and operations work properly together.

3.5 Data Collection and Analysis

The data collection involved the processing of a predefined number of images through a research run. The number of images remained constant for the entire research run, while the number of filters increased every 60 iterations within those configurations. During these 60 iterations, the cosine similarity operation was performed between each image and all the filters, followed by the max pooling of these results. The choice of 60 iterations was based on statistical guidelines for normal approximation (‘Calculating Confidence Intervals’, n.d.). This was important when establishing a reliable baseline for the average runtime of this configuration on the given hardware. After completing the 60 runs with the specified configuration, the number of filters incrementally increased and would go on for another

60 runs until it reached the used-defined max filter limit. This process has been visualized in Algorithm 1.

Algorithm 1 Image Processing with Dynamic Filter Adjustment

```
Input: $n_{\text{images}}, n_{\text{filters\_initial}}, \Delta n_{\text{filters}}, n_{\text{pool\_size}}, n_{\text{filters\_max}}$
1: $n_{\text{filters}} \leftarrow n_{\text{filters\_initial}}$
2: while $n_{\text{filters}} \leq n_{\text{filters\_max}}$ do
3:     for $i = 1$ to $60$ do
4:         Process all $n_{\text{images}}$ images with $n_{\text{filters}}$ filters
5:         Calculate cosine similarity
6:         Apply max-pooling of size $n_{\text{pool\_size}}$
7:     end for
8:     $n_{\text{filters}} \leftarrow n_{\text{filters}} + \Delta n_{\text{filters}}$
9: end while
```

3.5.1 Empirical and Quantitative Data

Empirical data has been gathered through experimentation with the different implementations developed throughout the development process. During research runs, data is obtained in real-time by writing to CSV, providing insights into the machine learning algorithm under various loads. This empirical data is also quantitative, represented in a numerical format, and is therefore appropriate for statistical analysis. This data is crucial as it is the objective analysis of these that represents the findings of the research conducted.

A sequence of identical experiments would then be performed on the different hardware shown in Table 3.1. This ensured data collected was comparable for all experiments across the different platforms.

3.5.2 Experiments

To gather data that can be used for a thorough comparison of the different parallelization strategies and hardware performance, several experiments had to be conducted. The different experiments and a description of what those experiments were supposed to find out or test are listed below:

ID	Images	Filters Start	Filters End	Increment Amount	Pool Size	Hardware
1	1000	100	2500	10	500	CPU
2	500	1	500	1	500	CPU & GPU
3	100	10	5000	10	500	CPU & GPU
4	10 000	500	100 000	500	500	GPU
5	10 000	500	100 000	500	500	GPU

Table 3.3: Summary of Experiments
Where img is the number of images, start is the start amount of filters, and end is the ending amount of filters, increment amount is how many filters are added each time, pool size is the number of elements being max pooled and hardware is the targeted hardware for the experiment.

All experiments work with a specified constant number of images and an increasing number of filters. Because of this, the result is expected to be linear as this approach has a time complexity of $O(n)$ despite the algorithm having a time complexity of $O(n \times m)$ given that all the images have the same size.

Experiment 1: Comparison of different CPU implementations

This experiment looks at a smaller number of computations between filters and images to target the CPU and compare the performance of different implementations. This experiment is meant to showcase today's methods, Python with NumPy, compared to a similar version in Rust and a parallelized version in Rust.

Experiment 2: Comparing CPU with GPU

The number of images processed in this experiment is less than in the experiment above. The reason for this and the reason for the low numbers of filters, both start, end, and increment, is because of the targeted hardware, the CPU and the GPU. This experiment is trying to showcase the computational power difference between the CPU and the GPU, i.e. when the GPU is the better hardware option for the computation amount.

Experiment 3: Comparing different shader implementations on the GPU

As this thesis deals with different shader designs and implementations for running on the GPU, this experiment is run to showcase the performance of the different strategies compared to each other. This will tell what strategy is best suited for computing on the GPU. The number of images and filters is low as the point of this experiment becomes clear within the given range.

Experiment 4: Exploring the limits of the GPU

This experiment targets the GPU and tries to find the limit of what the GPU is capable of. Therefore all three numbers, images, filters, and the increment amount, are high enough to capture the whole range of performance from the GPU without having the computation take several days.

Experiment 5: Exploring the parallel shaders

The last experiment explores the performance of a parallel shader design in contrast to a looped design. This will indicate which variant is the most suitable for the given task. In the looped version, the calculations are processed separately. i.e. that one thread is responsible for one entire calculation. while The parallel variant divides an entire calculation into several threads. This results in data having to be temporarily stored in a temporary buffer on shared memory and threads having to be synchronized to calculate the last part as explained in the Section 3.4.3.

3.5.3 Timing of Computation

For accurate timing of the experiments, the project utilized the built-in hardware clock of the computers. Further, it was necessary to determine the relevant points within the experiment to begin and end the timing. It was decided that the timing should start once the images and filters were loaded into the main memory of the platform, and stop once the final results were returned. This approach was decided because, while the CPU has direct access to the main memory, the data needs to be transferred to the GPU before it can be processed, as mentioned in Section 2.5.1. Since the calculated results reside in the memory of the GPU, the CPU can not access them before they are returned from the GPU. Therefore, the timing of the data transfer between the main memory and the memory of the GPU was also included.

The only exception to this approach is found in Experiment 5, described in Section 3.5.2, which focused solely on the computation time itself, excluding any data transfer times.

3.5.4 Data analysis

The theoretical background for the different data analysis techniques used throughout this thesis is presented in Section 2.8. The collected data is stored in CSV files with the format displayed in the table below.

Filter Amount	**Run ID**	**Time (Microseconds)**	**Average Time**
500	1	259	0
..	..	..	..
0	60	280	0
0	0	0	270
1000	1	359	0
..	..	..	..

Table 3.4: The format for the data saved on each run in a CSV file with example data.

The example data, as presented in the table above 3.4, consists of the 60 runs discussed in Section 3.5 and illustrated in Algorithm 1. These runs involved a specific number of filters before the average time is stored and the new 60 runs start with a larger amount of filters. All runs here have the same amount of images processed.

In this thesis, the techniques for data analysis are a combination of interpolation and standard deviation calculations. The analysis interpolates the average run times derived from the 60 iterations for each experimental configuration, as referenced in 1. To quantify the variability in these runs, the standard deviation of the run times is calculated. This is visually represented in the graphs as a shaded area surrounding the interpolated graph, illustrating the variability of the data. The reason for choosing interpolation instead of regression, despite both being able to showcase the characteristics of the collected data, is that interpolation fills in the gaps between the data points instead of modeling the relationship between filters processed and computation time.

3.6 Team process

The group consists of three students, each pursuing a degree in Computer Science, with some experience working together on previous projects. Due to this prior experience, the group did not require any team-building exercises to efficiently work together. Regardless, the group did have some social gatherings after work hours throughout the project period. All members of the group are independent students used to initiating work and analyzing various obstacles they encounter. Therefore, it was decided to have a flat structure and not appoint any group leader for the project. However, some roles were defined to ensure that team meetings with the supervisor were more structured and concise.

Each member of the group had similar relevant experience for this thesis, though with some variation. For instance, in terms of system-level programming, one member had experience in both C++ and Rust, while the two others each had experience in each of their respective languages.

3.6.1 Division of Labor and Roles

At the beginning of the project period, roles and responsibilities were defined to make the overall workflow more efficient. The responsibilities were based on past knowledge and experience and ensured that everyone contributed equally to the end goal. The member with experience in both Rust and C++ programming took charge of the GPU implementation in Rust. As one of the members had prior experience with working with filters for image recognition, it was natural that that member took responsibility for this area.

The roles meant to structure team meetings with the supervisor, mentioned in the section above, are the referent, the meeting coordinator, and the meeting organizer. The meeting organizer sets up the upcoming meetings with time and date and is responsible for that everyone who intends to join the meeting gets the meeting notice. The meeting coordinator takes the meeting through the pre-planned meeting schedule while the referent takes notes for later use so that anyone missing the meeting can read up on what was discussed.

3.7 Procedures

Early in the project, the group established procedures to make everyday interactions and assignments more clear and effective. These procedures were related to meetings, attendance, behavior, and document management, detailing both the methods and the reasoning for their implementation. An example of a procedure regarding meeting protocols says that the notice of a meeting should be sent via email at least 48 hours in advance of the meeting.

3.7.1 Working Hours

As the group had no office to work in, the group resorted to booking group rooms at NTNU as an alternative. Due to the high demand for these group rooms, NTNU only allows for bookings up to two weeks in advance. This meant the group had to strategically distribute booking responsibilities among the group members to ensure a workspace each day. As a result of this, the group established a solid work routine, with working hours from 09:00 to 17:00. Towards the end of the project period these hours were occasionally extended to ensure the high quality of the thesis.

4. Results

The results present the findings from the research conducted and are crucial for supporting the discussions later in the thesis. This chapter is structured to give an overview of the different aspects of the overall result.

The results section of the thesis contains administrative, engineering, and scientific results to give a comprehensive understanding of the research conducted. The experiments and the data collected from the research will be presented with the aid of visual tools such as tables and graphs to facilitate easier understanding and analysis of the data.

4.1 Scientific Results

This section will provide the collected processing results from the experiments introduced in Section 3.3. The data gathered from the two platforms, as displayed in Table 3.1, will be visualized using the analytical techniques presented in Section 3.5, interpolation combined with standard deviation. All figures show the different implementations and versions processing time, except Figure 4.1.5 which shows computational times for the two different shader strategies.

4.1.1 CPU

This section presents the result gathered from Experiment 1 (Section 3.5.2), conducted on both platforms. This experiment has a constant number of images, one thousand, with increasing numbers of filters from 10 to 2,500 in steps of 10. Figure 4.1 shows data gathered from Platform 1 while Figure 4.2 shows data from Platform 2. The red graph is the Python implementation utilizing NumPy to perform the calculations while the other graphs are implemented in Rust. The green is the sequential Rust implementation, and the blue is the parallel Rust implementation using Rayon for parallelization.

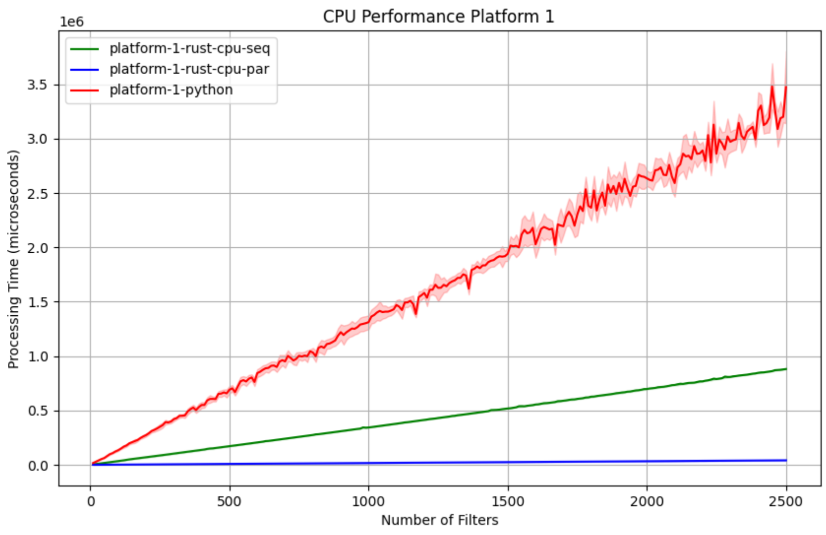


Figure 4.1: The performance of the CPU on Platform 1

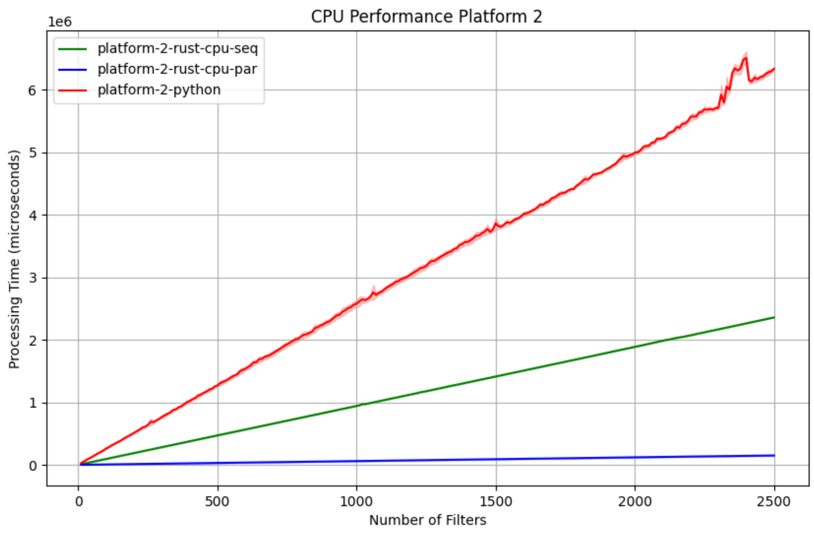


Figure 4.2: The performance of the CPU on Platform 2

All implementations show a linear trend of increasing processing time as the number of filters increases on both platforms, with the parallel Rust version demonstrating the best results. Both sequential implementations (green and red) perform worse than the parallel implementation, with Python being the least efficient. There is some variation in the performance of the Python implementation (red) while both Rust implementations remain stable. This variation is more present on Platform 1 shown in Figure 4.1 while still maintaining better performance than Platform 2 as Figure 4.2 shows.

4.1.2 CPU versus GPU

The graphs shown in Figure 4.3 and Figure 4.4 compare the performance CPU and GPU on Platform 1 and Platform 2 for Experiment 2 presented in Section 3.5.2. This experiment has a constant number of images, 500, with an increasing number of filters from one to 500 in steps of one. The experiment was done using three different implementations: a Rust implementation in parallel utilizing the CPU (red), and GPU implementation in Rust utilizing the for-loop shader (blue), and a C++ implementation utilizing CUDA (green).

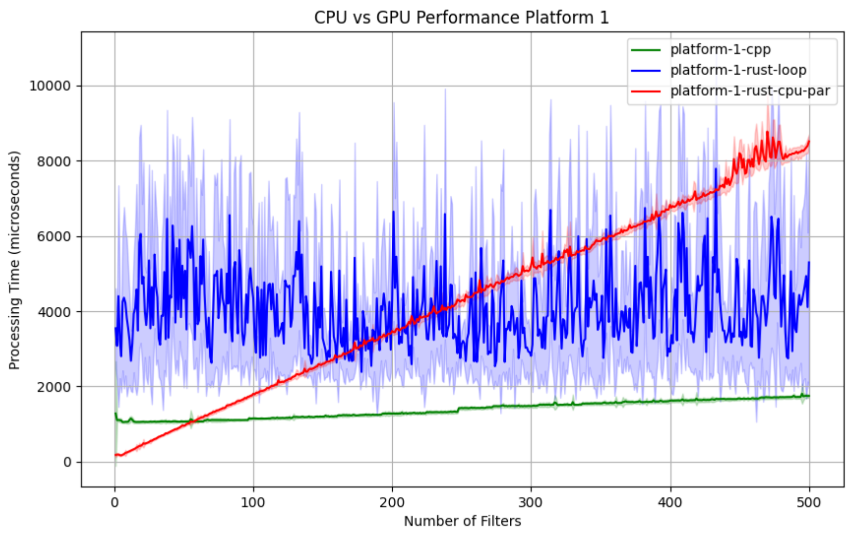


Figure 4.3: The performance of the implementations CPU and GPU on Platform 1

On Platform 1, the C++ implementation exhibits the lowest processing time. The Rust for-loop implementation displays significantly variable performance across the run with a generally higher processing time. The CPU parallel implementation exhibits a consistently increasing linear processing time, surpassing both the Rust for-loop and the C++, with a slightly increasing variability in processing time and standard deviation throughout the experiment.

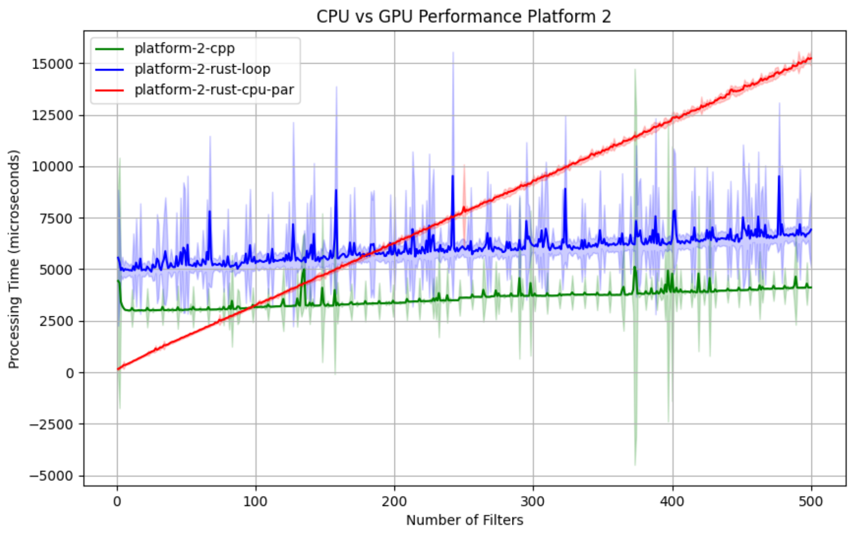


Figure 4.4: The performance of the implementations CPU and GPU on Platform 2

For Platform 2, the C++ implementation still shows the lowest processing time, although with more variability in both performance and standard deviation. In contrast, the Rust for-loop implementation displays more consistent performance, although with some variability. The CPU implementation shows more consistent performance than on Platform 1, with an increase in standard deviation in processing time towards the end of the experiment.

In Experiment 2, the CPU implementation shows a linear increase in processing time along with an increase in standard deviation. Conversely, the Rust GPU for-loop implementation displays a high degree of variability on Platform 1, with a significant standard deviation, indicating fluctuating performance variability. On Platform 2, the Rust GPU for-loop implementations show a slightly higher linear increase with a smaller standard deviation.

In contrast, the C++ GPU implementation, like the CPU, also has a linear trend, but with a significantly slower increase rate, maintaining its speed even with increasing dataset size. One can also observe an evenly fluctuating standard deviation for the C++, which is much more visible on Platform 2 than Platform 1, but less frequent than the Rust for-loop.

The C++ implementation remains faster than the CPU implementation throughout the entirety of the run. On Platform 1, it takes about 60 filters before the CPU implementations get slower than the C++ implementation, while on Platform 2, it happens at around 100 filters. For the Rust GPU for-loop implementation, the CPU becomes slower after approximately 200 filters on both platforms. However, it is the C++ implementation that performs best throughout the entire experiment.

4.1.3 Shader Strategy

The graphs shown in Figure 4.5 and Figure 4.6 compare the performance results of Experiment 3, where 100 images are processed, starting with 10 filters and increasing by 10 filters until reaching 5000 filters, as described in Section 3.5.2. Different GPU shader strategies and the implementation in parallel on the CPU are tested on both Platform 1 and Platform 2. The experiment was performed using three shader implementations: the Rust for-loop shader (blue), the Rust one-image-all-filters shader (purple), the optimized Rust one-image-all-filters shader (red), as well as the Rust parallel CPU implementation (green).

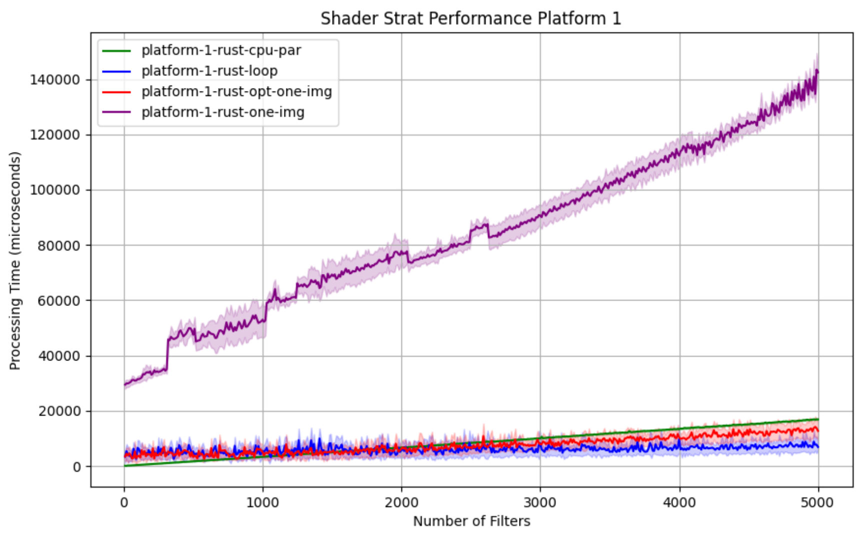


Figure 4.5: The performance of different shader strategies on Platform 1

Figure 4.5 shows the performance of the different shader strategies on Platform 1. The graph displays that the Rust one-image-all-filters shader without optimization consistently has the highest processing time, increasing faster than the others with the number of filters. The optimized version shows a significant improvement over the original but still displays a slower processing time than the Rust for-loop shader strategy. The Rust for-loop shader remains stable throughout the experiment. CPU parallel implementation increased linearly starting as the fastest implementation, but soon got surpassed by both the Rust optimized one-image-all-filters shader and the for-loop shader.

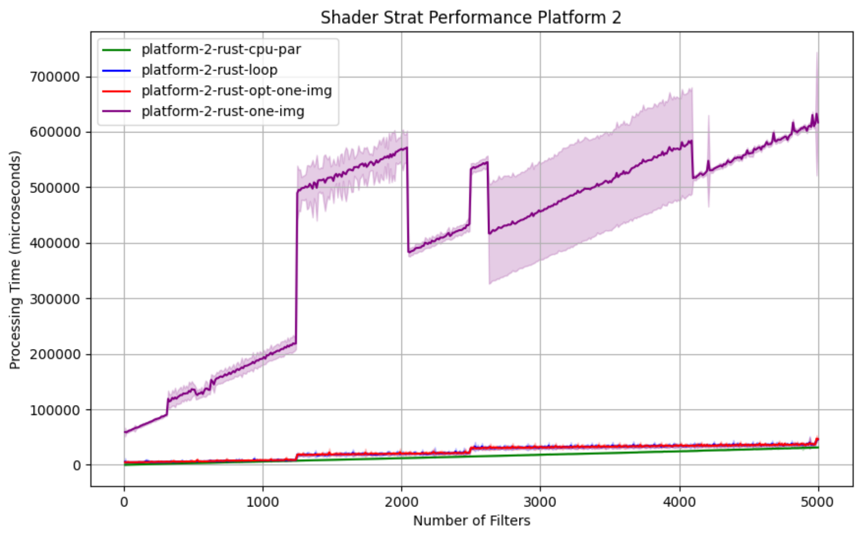


Figure 4.6: The performance of different shader strategies on Platform 2

Figure 4.6 presents the performance of the various shader strategies on Platform 2. The Rust one-image-all-filters shader exhibits the overall highest processing time, with substantial fluctuations and spikes, especially around 2000 and 3000 filters. In the optimized Rust one-image-all-filters demonstrated better performance, equal to the performance of the Rust for-loop shader. The Rust for-loop shader shows stable performance with minor variations in standard deviation. The Rust parallel CPU implementation increases linearly and consistently, achieving the lowest processing time across all the implementations.

4.1.4 Limits of GPGPU

Figure 4.7 and Figure 4.8 shows the performance of the different GPU implementations from Experiment 4 presented in Section 3.5.2. This experiment has a constant number of images, 10,000 with an increasing number of filters from 500 to 100,000 in steps of 500. On both platforms, the red, blue, and purple are different shaders run with Rust and the green is one of the same shaders run with C++. The three different shaders are the for-loop shader (blue for Rust and green for C++), parallel shader (red), and optimized one-image-all-filters (purple). All Rust runs stop at 53,500 filters because of a memory limitation with the current implementation.

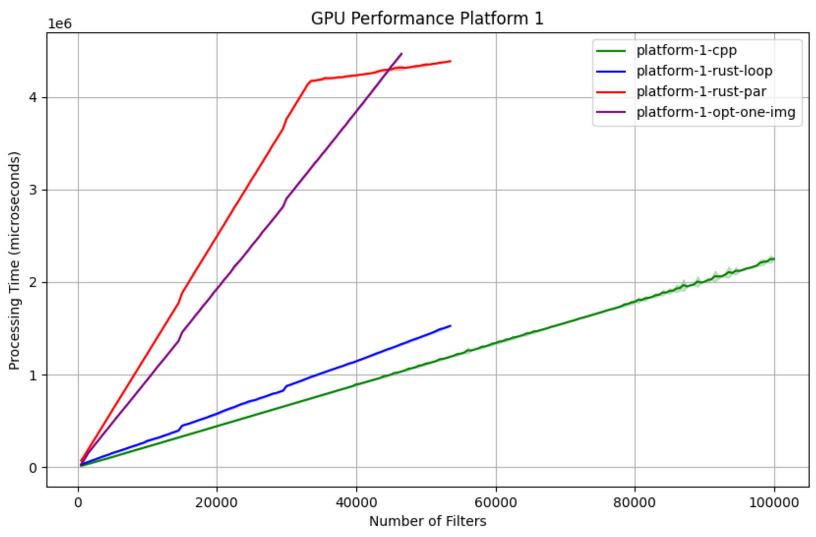


Figure 4.7: Interpolated processing times for various GPU programming implementations on Platform 1.

The graph shown in Figure 4.7 displays the processing times of Platform 1. The C++ implementation had a linear increase throughout the experiment. The same goes for the Rust for-loop implementation, except for two anomalies observed around 15,000 and 30,000 filters where the processing time suddenly increased. The same anomalies are seen in the optimized Rust one-image-all-filters implementations, which consistently have higher processing times. The Rust parallel shader has the slowest processing time out of the shaders, with similar spikes at 15,000 and 30,000 filters. Unlike the others, at around 30 to 35 thousand filters, this implementation slows down, stabilizes, and then shows a slower increase.

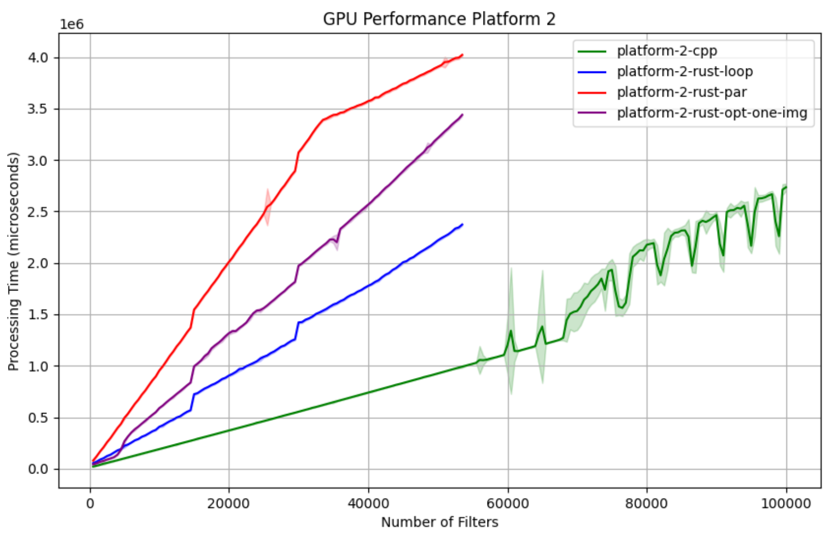


Figure 4.8: The performance of the GPU on Platform 2

The processing time of the shaders on Platform 2 is displayed in Figure 4.8. Similar trends as on Platform 1 can be observed, although with some deviations. As for the aforementioned anomaly, it can still be observed for the Rust shaders at around 15,000 and 30,000 filtes. The C++ implementation displayed a linear increase up until around 60,000 filters, at this point, it started to show a spike in processing time, followed by an inconsistent performance for the remaining part of the experiment. The Rust for-loop and optimized one-image-all-filters experienced similar trends, with the latter being the slower of the two shaders. As for the Rust parallel shader, it displayed a linear increase in processing time up until around 30 to 35 thousand filters, where the increase rate slows down.

4.1.5 Parallel versus Sequential Shader

This section displays the results from conducting Experiment 5 presented in Section 3.5.2. This experiment has a constant number of images, 10,000, with an increasing number of filters from 500 to 100,000 in steps of 500 trying to showcase the performance difference between a sequential and a parallel shader design. Figures 4.9 and 4.10 show the computation time of both shaders for calculating the cosine similarity while Figures 4.11 and 4.12 shows the computation time of both max-pool shaders. Every shader in this section is run with Rust. The parallel shader design has the color blue while the sequential is green.

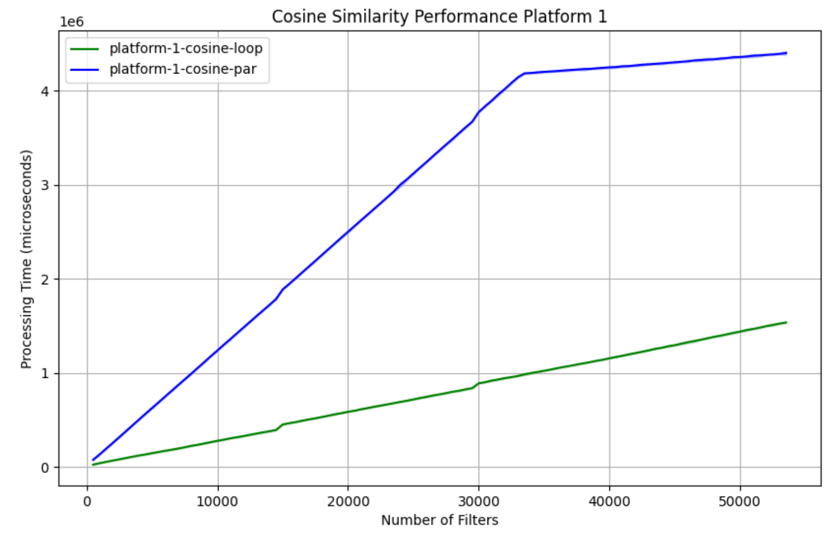


Figure 4.9: Interpolated computation times for the Cosine Similarity shaders for Platform 1.

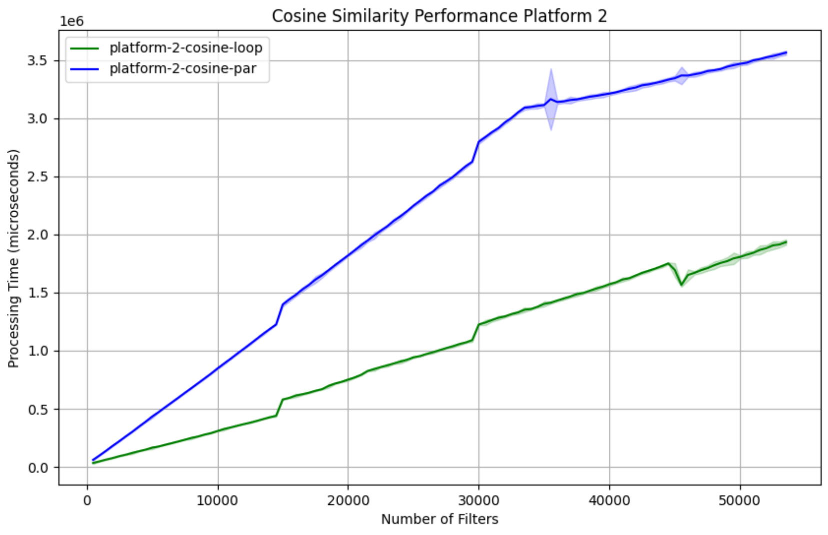


Figure 4.10: Interpolated computation times for the Cosine Similarity shaders for Platform 2.

The cosine similarity shaders on both platforms shows a similar performance whereas the sequential shader utilizing a for-loop for the whole calculation performs the best. Both graphs have a linear trend while Platform

2 has some spikes and drops in performance on both designs. The parallel shaders begins to plateau around 30 to 35 thousand filters on both platforms with Platform 1 flattening out more.

Max Pooling Perfomance

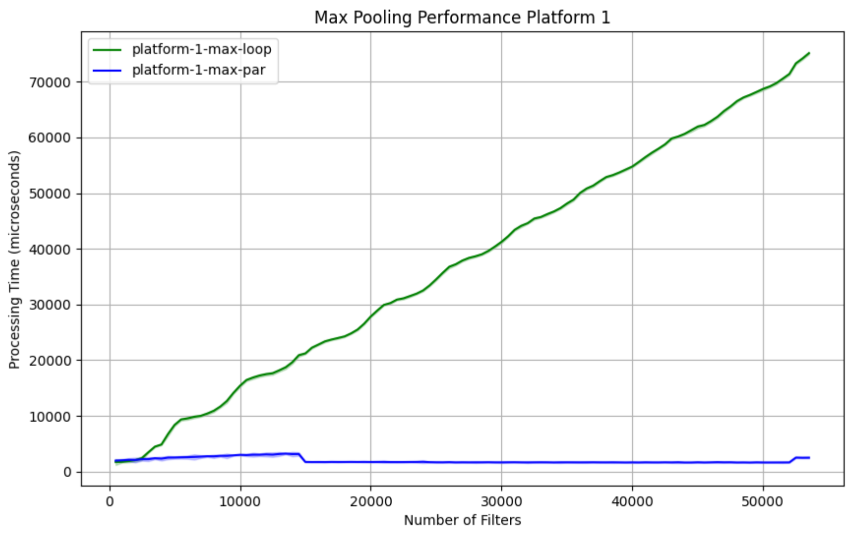


Figure 4.11: Interpolated computation times for the max pooling shaders for Platform 1.

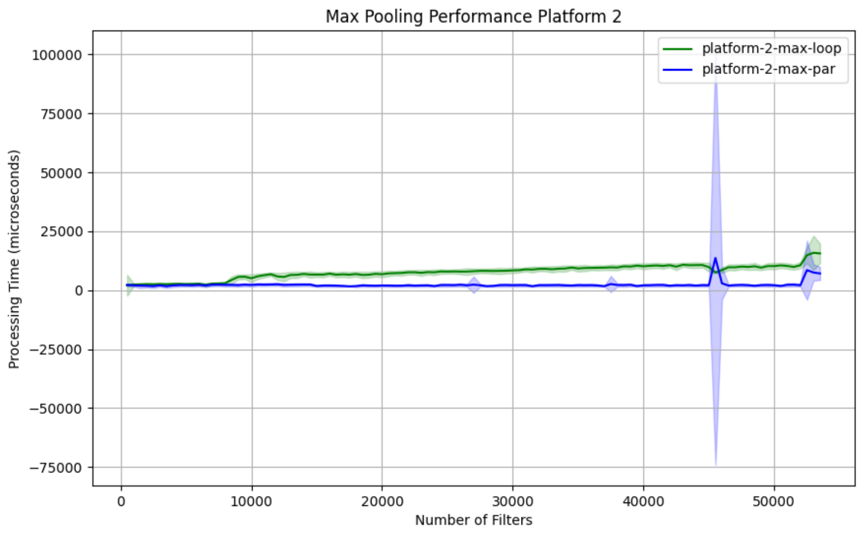


Figure 4.12: Interpolated computation times for the max pooling shaders for Platform 2.

The max-pool shader on both platforms shows the opposite result of the cosine similarity with the parallel design performing the best on both platforms. Here in figures 4.11 and 4.12 the parallel shader design has almost a constant computational time regardless of the increase in processing data. There are some spikes and drops in performance with Platform 2 having the most fluctuating computational time and one high deviation around 46 thousand filters. Both sequential max-pool shaders perform worse than the parallel ones with Platform 1 being the overall worst having a linear increase in computational time while Platform 2's run is almost a flat graph.

4.1.6 Other findings

This section presents some other findings throughout the project. These findings were found with a constant number of images, ten thousand, and an increasing number of filters from 500 to 100,000 in steps of 500. Here Figures 4.13 and 4.14 show the difference in the performance of the for-loop shader in Rust when adjusting the dispatches on the GPU. Additionally, Figures 4.15 and 4.16 show the fluctuating performance of the C++ runs with the same shader on both platforms.

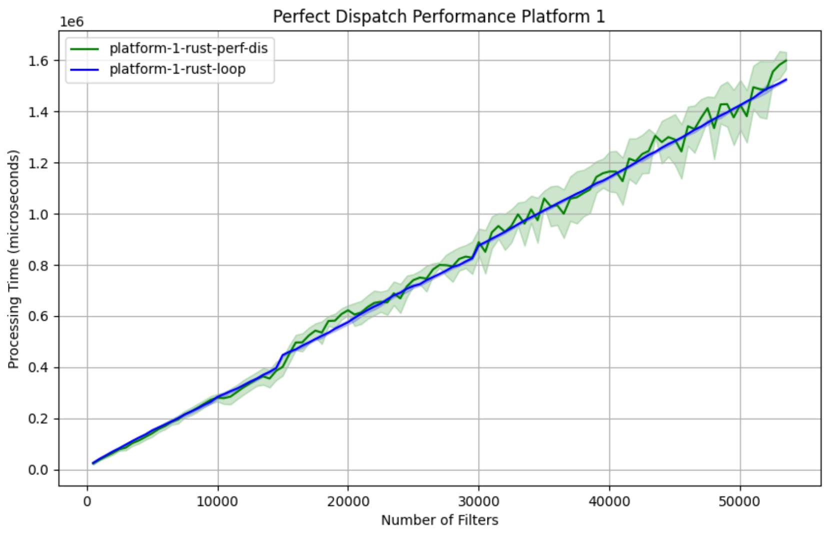


Figure 4.13: Interpolated processing times for adjusted dispatch GPU programming implementations for Platform 1.

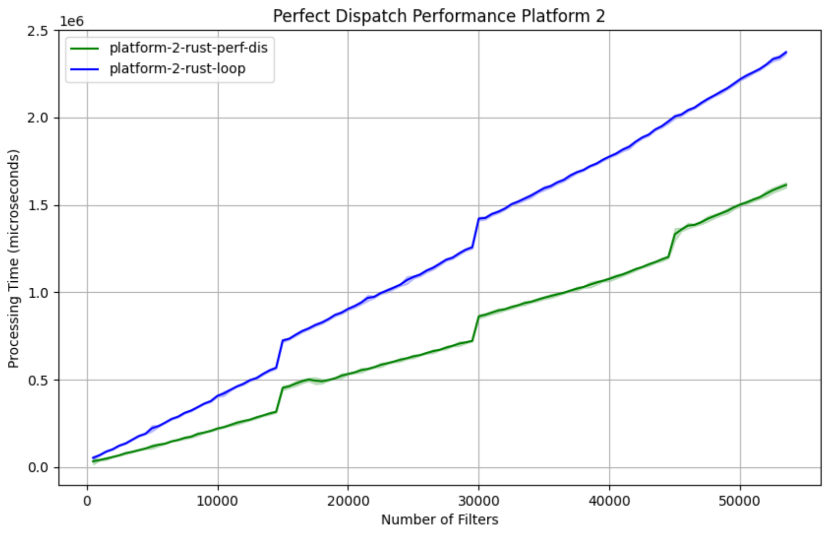


Figure 4.14: Interpolated processing times for adjusted dispatch GPU programming implementations for Platform 2.

Figure 4.14 shows an increase in performance for the adjusted-dispatch-run (green) compared to the same shader with an overestimate in the number of dispatches. On the other hand, figure 4.13 only shows a larger variation in performance for the adjusted-dispatch-run instead.

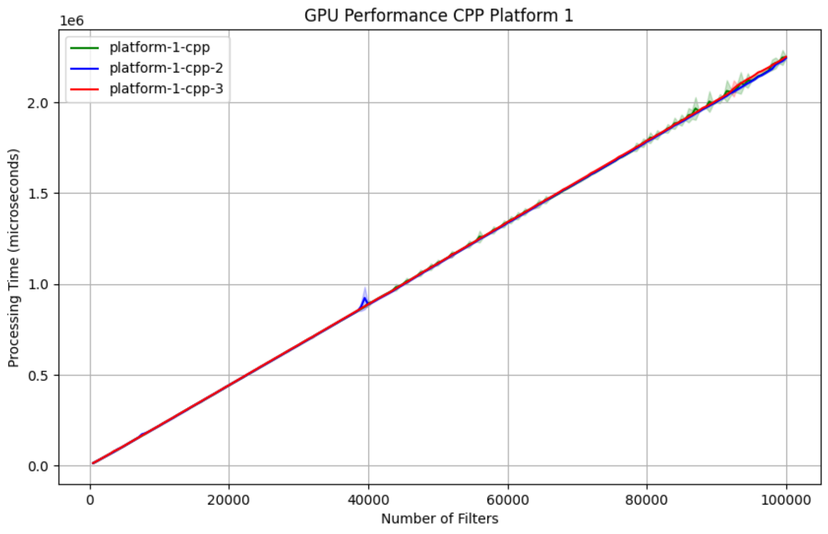


Figure 4.15: Interpolated processing times for C++ GPU programming utilizing CUDA for Platform 1.

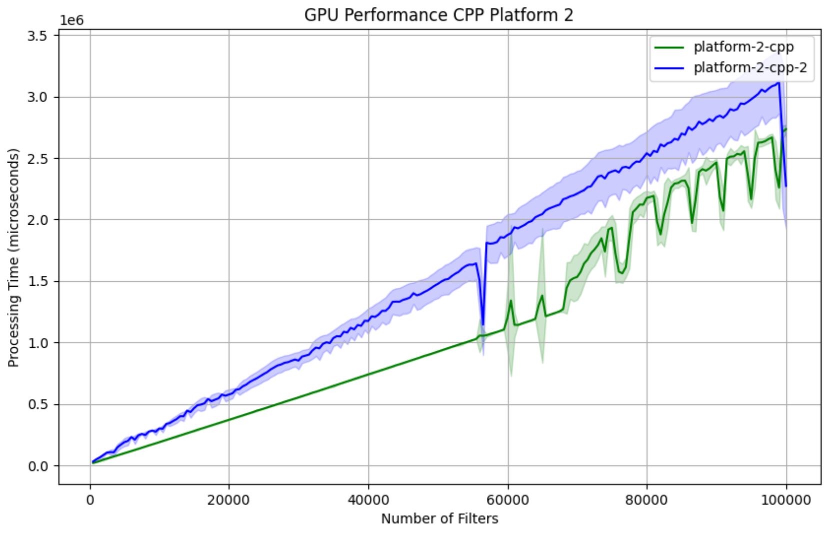


Figure 4.16: Interpolated processing times for C++ GPU programming utilizing CUDA for Platform 2.

Figure 4.15 shows the stable and almost zero variation between three separate runs with C++ implementation of the for-loop shader. Figure 4.16 on the other hand, shows the variation in two separate C++ runs. The green graph displays the run processed in day time while the blue more stable run was processed at night time.

4.2 Engineering Results

Subsequently, this section will focus on the engineering aspects of the project, highlighting the development and testing of the different solutions. This also includes a review of the project's original objectives, as defined at the beginning of the project period.

4.2.1 Project Objectives

The primary objective for this project was to parallelize machine learning methods using different programming languages, such as C, C++, Rust, and/ or other specialized hardware.

The group also established a set of impact, performance, and process objectives to evaluate both the group and the implementations at the end of the project period. The impact objectives focused on enhancing machine learning methods using different programming languages. A long-term objective was to reduce power consumption from complex calculations which would give societal benefits given the increasing use of Artificial Intelligence (AI). The group also aimed to increase their knowledge on the topics of GPU hardware, system programming, and parallel programming.

The performance goals included the implementation of a solution that could perform the cosine similarity operation followed by the max pooling operation on the GPU in parallel, and ensure concise results across multiple experiments. This also included the objective of obtaining a deeper understanding of the field of GPU.

Lastly, the process objectives aimed to enhance the group's expertise in system programming and parallel computing.

4.2.2 Status at Delivery

At the end of the project period, the group had successfully developed several implementations across different programming languages. These included CPU implementations made for sequential experiments in Python using NumPy, C++, and Rust, as well as an implementation running parallel on the CPU in Rust. For computations on the GPU, the group had developed one implementation in C++ utilizing CUDA, alongside several implementations in Rust to explore different optimization techniques. The results of these implementations are presented in section 4.1.

During the project period, the group made the strategic decision to move away from the idea of implementing a specialized hardware solution, using FPGA and potentially ASIC. This will be further discussed in section 5.2

4.2.3 Comparison of Objectives and Results

When evaluating the results against the objectives set by the group, it’s clear the group has made significant advancements towards them. The overall impact goal was to enhance machine learning methods by utilizing the GPU, which was achieved through the implementations in C++ and Rust, which surpassed traditional CPU implementations.

The performance objectives of developing an implementation that could perform cosine similarity and max pooling operations in parallel on the GPU were also achieved, as proved with the C++ and Rust implementations. These solutions did not only show the efficiency benefits of the GPU but also showed how the GPU performs better on larger data sets as shown in section 4.1.

4.3 Administrative Results

For the administrative aspect of results, the focus will be on management and resource utilization within the project. An overview of the distribution of time across the group members and different phases of the project will be presented.

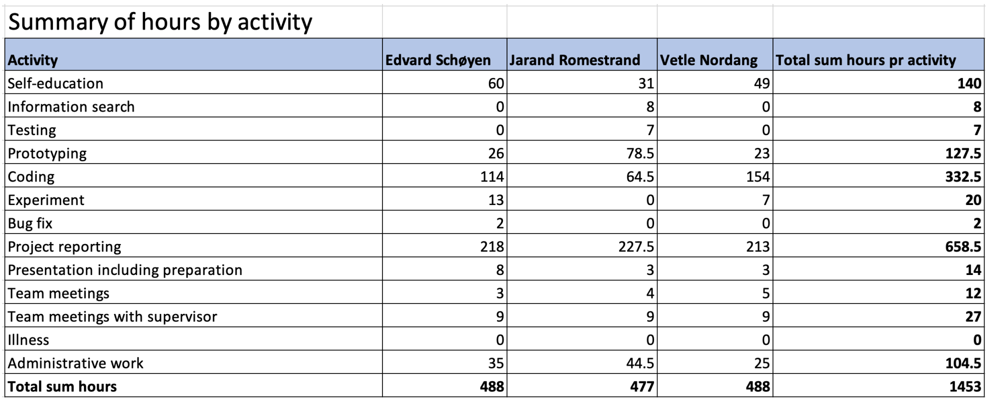
4.3.1 Project Plan

During the initial phase of the project, the group created a GANTT chart to illustrate the expected time usage per activity for the coming semester found in Appendix B. This chart was a rough estimate due neither of the group members had conducted scientific research of this kind before the project.

Due to this, there was a deviation between the estimation and reality seen in table 4.3.2.

4.3.2 Time Management

During the project period, the group logged hours spent each day. This gives a clear understanding of how time was distributed throughout the semester.



Summary of hours by activity

Activity	Edvard Schøyen	Jarand Romestrand	Vetle Nordang	Total sum hours pr activity
Self-education	60	31	49	**140**
Information search	0	8	0	**8**
Testing	0	7	0	**7**
Prototyping	26	78.5	23	**127.5**
Coding	114	64.5	154	**332.5**
Experiment	13	0	7	**20**
Bug fix	2	0	0	**2**
Project reporting	218	227.5	213	**658.5**
Presentation including preparation	8	3	3	**14**
Team meetings	3	4	5	**12**
Team meetings with supervisor	9	9	9	**27**
Illness	0	0	0	**0**
Administrative work	35	44.5	25	**104.5**
Total sum hours	**488**	**477**	**488**	**1453**

Figure 4.17: Logged hours amongst the group members

Activity	Estimated Time Spent	Actual Time Spent
Self education	360	148
Programming	720	469
Data analysis	36	20
Project reporting	396	777
Team meetings	0	39

Table 4.1: Comparison of estimated versus actual time spent

4.3.3 Development Methodology

The integration of the DSR and the Lean methodology have been essential for the development process of the project. Each methodology contributed to an effective, structured, and flexible development cycle, which was vital given the complex assignment.

DSR introduced a systematic approach to attack the task description. This was crucial for structuring the project in clear, manageable phases. It also encouraged a continuous improvement cycle for each phase of the development. This ensured the development of artifacts aligned with the original task description. This also helped maintain focus on the aspects crucial for the task, which improved development in the development process.

The Lean methodology complemented DSR by introducing principles that optimized resource usage within the group and prioritized development in line with the project's needs and constraints. Frequent work sessions and close collaboration among the group members were some of the key contributions of the Lean principles. Meeting regularly to work together allowed for quickly addressing challenges and making the necessary changes, enhancing the productivity of the group, while ensuring the project stayed on track. This allowed for fast iterations and adjustments based on the feedback and results of the development and the testing. This turned out to be beneficial for the group given the steep learning curve of the GPU programming.

By emphasizing values defined by the project's potential contribution to the field of machine learning, Lean ensured that each step in the development process was targeted and contributed to the end goal of the project. This helped eliminate unnecessary tasks and maximized the collected contributions of the group.

The use of Lean's iterative approach to development ensured continuous improvement of the implementations. These cycles of programming, testing, and improving were essential for the optimization of the performance of the solution.

The project was driven by a cooperative group environment and made possible through the methodologies used. Tools such as Discord, for informal communication, Teams for document management and meeting planning, and Git for version control, illustrated the application of Lean methodology in daily practice.

5. Discussion

The discussion section of the thesis will delve into the various results of the project and evaluate them. It is through this analysis that one gains a deeper understanding of the impact of the project, the effectiveness of the methods implemented, and the technical advancements made.

5.1 Administrative

This section will explore the administrative results of the project. Topics such as project management, time allocation, and deviation (from time management).

5.1.1 Project Time Estimations and Deviations

To highlight the differences between the estimated project times and the actual time spent on activities, this subsection compares the Gantt chart projections with the real project timeline.

During the planning of the project, the group estimated significant time for self-education, with the steep learning curve of GPU programming. However, the actual time spent turned out to be considerably less than expected. This is due to the application of the DSR research methodology, where the practical creation of artifacts contributed significantly to the group's understanding of the topic. This hands-on learning through testing and experimentation, proved to be more efficient than theoretical study alone.

Furthermore, the group underestimated the amount of time that would end up going to project reporting, given the theoretical complexity of the research-focused thesis. This resulted in a misalignment between the estimated and actual workload, which forced the group to work longer hours in the final stages of the project period.

The group overlooked including group meetings with the supervisor in the Gantt chart, as it did not occur to the members at the time. This oversight resulted in a discrepancy, as a significant amount of hours went into planning and attending these meetings, as well as writing meeting minutes.

5.2 Engineering

The engineering subsection will cover the technical execution of the project, highlighting the development and implementation phases.

5.2.1 Achievement of Project Objectives

Specific impact, performance, and process-driven objectives were defined, throughout the project. The primary performance objective was the effective utilization of the GPU to enhance the specified machine learning methods. This objective was achieved through the development of successful implementations in C++ and Rust. These implementations demonstrated superior performance when compared to traditional CPU-based solutions, especially when executing the cosine similarity and max pooling operations. A notably positive and surprising outcome was how flexible the GPU implementation was. It managed to handle larger sets of images and filters with relative ease.

In terms of impact goals, the project aimed to enhance the performance of the specified machine learning methods by utilizing the GPU. The GPU implementations, as previously mentioned, achieved a significant improvement when compared to the CPU implementation. This aligns with the objectives of developing more efficient machine learning methods.

In parallel with the work on the implementations, the group also gained knowledge that helped work towards the educational objectives which was a part of the impact objectives.

Therefore, comparing the objectives set by the group before the project period against the results achieved, one can say it's been a successful endeavor. The group has managed to achieve all objectives which proves the effectiveness of the research and the development methodology.

5.2.2 The shift away from FPGA and ASIC

Initially, the group wished to, in addition to the Rust and C++ solutions, implement a solution for FPGA and potentially ASIC due to their promising capabilities in parallel programming as presented in section 2.9.

Access to FPGA was available through NTNU's IDUN Cluster, making it a potential option for the project. However, after a dialog between the group and the primary FPGA specialist at IDUN, the decision was made to move away from this option. The primary reasoning behind this shift was the resource constraints, particularly time and lack of knowledge in the field. Attempting to implement a solution utilizing FPGA would therefore have made too much of a risk to the rest of the project. This strategic choice allowed the group to reallocate time and resources towards developing solutions for the GPU.

5.3 Scientific

This subsection will present a detailed discussion covering the results presented in section 4.1. The analysis and interpretation of the data will be discussed, and it is through this the potential for further research and development will be highlighted.

5.3.1 Overall findings

Some aspects of the results repeat themselves on all the graphs on both platforms from the different experiments presented in section 3.3. These general findings across the two platforms will be discussed here before delving into findings in the specific experiments and comparisons of the platforms.

The standard deviation of the runs conducted on the GPU using Rust seems to be constantly high compared to the other runs. There can be various causes for this. One of them is that Rust is in an early lifecycle phase compared to other system programming languages like C and C++, especially wgpu on version 0.19.3 during this project. Some drawbacks can be expected because wgpu is an early version abstraction layer for GPU programming.

One of these drawbacks discovered in this project is related to retrieving memory from the GPU to the CPU. When creating a layout of buffers for execution on the GPU using wgpu usage of these buffers needs to be defined. Storage is used when the buffer contains some data from the CPU, copy-src is used to be able to copy the data from buffer to another as the source, copy-dst is used as the destination of this data transfer, and map-read is used to map the contents from the GPU over to the CPU making that buffer unusable for the GPU as long as the buffer is mapped. To be able to retrieve data from the GPU to the CPU using wgpu the data needs to be mapped to the CPU and when the whole buffer is mapped retrieved to CPU memory. Because this is the only way, a buffer needs to have the storage usage for it to store data, and map-read to be retrieved. As of now, the combination of these two is not beneficial unless the system has shared memory between CPU and GPU. Therefore to be able to retrieve the data a staging buffer needs to be created with the usages copy-dst and map-read. Then the data gets copied over to this staging buffer before it gets mapped to the CPU. This could result in the larger standard deviation shown in the graphs as well as why Rust is that much slower compared to the C++ implementation that utilizes a specialized library for a specific architecture, CUDA.

The other drawback related to wgpu is that it is an abstraction layer for several different hardware architectures. It is not certain that being an abstraction layer over several architectures comes with some loss in performance, but this is usually expected. This could be another reason that the C++ implementation outperforms the Rust when both versions use the same shader, as the C++ uses CUDA which is assumed to be specialized for NVIDIA architectures because they are the developers as well.

5.3.2 CPU

Delving into the performance of the various CPU implementations shows a large difference in processing time depending on the implementation. This is shown in Figure 4.1 and Figure 4.2. All implementations show a linear trend, but the Python implementation on both platforms has some variation. There is a large difference in processing speed despite all implementations showing a linear trend.

The implementation performing the worst is the one in Python, despite it utilizing the NumPy library to perform the calculations. As most of the NumPy are written in C and C++ it is interesting to see that the sequential Rust implementation performs that much better. There can be many reasons for this, but it is most likely related to Python being an interpreted language. Interpreted languages are slower than compiled languages like Rust. But also linked to the large amount of data needed for the algorithm to be executed. This data must then be transferred from Python to the corresponding C and C++ code in NumPy resulting in some overhead. This data transfer between two languages could also be the reason for the large variation compared to the other implementation's performance.

As expected, the parallel implementation performed the best with a low increase in processing time for Experiment 1, shown in Section 3.5.2. This result suggests that a parallel approach is best suited for a machine learning method like the one used in this thesis.

5.3.3 CPU versus GPU

The graphs seen in Figure 4.3 and Figure 4.4, show that little processing is needed for the GPU to outperform the CPU and showcase its computational power. The fastest, shader C++ for-loop, has a distinct intersection at around 50 filters because of its consistency and low standard deviation. The rust for-loop has such a high standard deviation with no clean intersection but roughly outperforms the CPU from 250 - 300 filters. The CPU implementation has a low standard deviation because there are no noticeable memory transfers and two simple for-loops that do the calculations. The reason for the high standard deviation in rust-for-loop is uncertain but can be related to the early stages of wgpu and its memory transfer from the CPU memory and the layers of abstraction for communicating to the GPU. The consistency of C++ can be because of optimizations in terms of memory transfer in CUDA and communication as CUDA is owned by NVIDIA.

5.3.4 Shader Strategy

The one-image-all-filters performs more memory operations between the CPU and the GPU than the other strategies. These memory operations are what make the utilization of the GPU poor as most of the time used under processing is related to writing memory to the GPU and retrieving it. The one-image-all-filters strategy performs poorly compared to the "all images all filters" strategy and even the parallel CPU implementation. This is related to the other strategy's memory efficiency by only sending and retrieving data from the GPU once and giving the GPU more data to parallelize at once utilizing the hardware more.

When it comes to the all-images-all-filters strategy, it becomes clear that this strategy utilizes the characteristics of the GPU better than the one-image-all-filters strategy. This strategy uses two of the GPU's three dimensions to calculate the cosine similarities from all images with all filters. Using two dimensions this way makes the GPU process the calculations like it is a double for-loop in a parallel way all at once. Both figures, Figure 4.5 and Figure 4.6 show that modifying the one-image-all-filters shader to process all images with all filters is a better strategy as it outperforms the other shader by around seven times at the end of the run on platform 1, and even more on platform 2.

Utilizing the GPU, by calculating all the features from all images and all filters, is the best strategy when it comes to computation time but comes at a price of complexity. As this strategy utilizes two dimensions to process the data, it becomes more complicated to both make the shader race-free and not introduce other undefined behavior. This becomes even more difficult when considering that debugging on the GPU is hard, as mentioned in Section 3.4.5.

5.3.5 Limits of GPGPU

3.5.2 explores the limitations of the GPU when it comes to parallel algorithms like the one implemented in this thesis. As expected, all implementations have a linear trend. When the number of filters applied to the images increases, so does the processing time. The most obvious finding from looking at the graphs in Figure 4.7 and Figure 4.8 is that the parallel implementations perform worse than the looped-shader design implemented in both Rust and C++. This aspect is discussed in the section below and will not be the focus point here. The focus points are instead deviation from the linear trend, the memory limitations for GPU-programming, and the large variation in performance when the dispatches are adjusted to exactly what is needed.

Deviation from a Linear Trend

There are a couple of instances where the different implementations deviate from the linear trend. The most notable is the parallel implementation that almost begins to plateau at around 35,000 filters. This is discussed in Section 5.3.6.

By once more taking a close look at the graphs on the GPU performance from Figure 4.7 and Figure 4.8 shows a deviation from the linear trend appears at the same spot for all the Rust implementations. This happens at around 15,000 filters and again at 30,000 across both platforms. This deviation comes in the form of a spike in processing times as all implementations increase before stabilizing back to the linear trend. While the exact cause is uncertain, one plausible explanation is that around 15,000 filters, a significant number of blocks are dispatched to the GPU, leading to context switching between these blocks. This sudden increase in context switching could contribute to the observed spikes in processing times at around 15 and 30 thousand.

The spikes could also be related to the use of Rust and its libraries, as these spikes only occur for the Rust implementations. The C++ implementation utilizing the same shader as one of the Rust implementations has no visible spike before the Rust runs stop. The C++ run on platform 2 starts spiking and dropping after the Rust runs stop, this will be discussed in Section 5.3.7. It is difficult to say why the Rust implementations have sudden spikes at a set pattern while the C++ implementation is rather stable. There could be many reasons for this, but the most obvious reason is that the wgpu library used for GPU-programming is on version 0.19.3 during this thesis, so version 1 has not yet been published. This early version suggests that the code is not as optimized as maybe the C++ version is.

Memory Limitations

When looking at the graphs seen in Figure 4.7 and Figure 4.8, comparing the GPU implementations performance of both platforms the most obvious finding is that the Rust implementations stop at 53,500 filters. The reason for Rust stopping is that the output buffer created for storing the calculated results exceeds the limitation of the size of a shared memory buffer. This limitation gathered from the system using the WGSL library is weird as the C++ implementation using the CUDA library on the same system is capable of processing more than the doubled amount of filters. Why this is the case is uncertain as CUDA is not open source, but is assumed to also be related to wgpu's early version or related to how CUDA manages memory.

Performance Variation

Some minor testing where done to see the difference in performance by adjusting the dispatches to match the given number of calculations for the looped shader implementation in Rust. The results from this test are shown in the graphs in Figure 4.13 and Figure 4.14. The adjusted dispatch run has a larger variation in its linear trend compared to the one that dispatches the number of images and filters in the x and y directions on platform 1.

Why does the adjusted dispatch run vary so much? This variation is most likely related to a thread divergence phenomenon shown in Figure 2.3. Too many threads spawned vary depending on the number of calculations planned with adjusted dispatch. These superficial threads will result in thread divergence, and as the number of these threads vary so will the number of divergences and the performance. The same concept is also why the other one is so stable, as it always dispatches a constant number of superficial threads resulting in the same thread divergence and the stable linear trend.

Platform 2 shows the more expected result, which is that the adjusted dispatches perform overall better than the run that dispatches more than needed. This is explained as more dispatched will result in higher overhead, especially when extra dispatches are unused and only tribute to thread divergence.

5.3.6 Parallel Shader versus Sequential

The all-images-all-filters shader strategy can be implemented in several ways. The three implementations this thesis covers utilize a for-loop to process the whole image with the filter, and two parallel shaders that split the image processing into several parts introducing temporarily shared memory storage and synchronization. Both designs are implemented for the cosine similarity and the max pooling operations. The graphs presented in Section 4.1.5 illustrate the performance difference between these two strategies by comparing the computation time of the for-loop shader and the similar implementation in parallel.

The main difference between the two shaders and their strategy is related to work distribution between threads. The looped shader has fewer threads that do more work. This implementation does not need synchronization and storing unfinished calculations on shared block memory. While the parallel shader does the opposite, more threads do less work.

Cosine Simularity

The downside to this parallel design is the need for synchronization to finish the whole calculation as the method that is used throughout this thesis is a reduction operation. This makes the calculations of one image and one filter split between several threads dependent on each other to be reduced to the final value. This synchronization is one of the reasons that the parallel shader performance is worse than the looped design.

The gain of parallelizing the processing of an image and filter inside the thread block gets overshadowed by the cost of synchronizing the threads. An interesting find here is in the graphs for the cosine similarity graphs shown in Figure 4.9 and Figure 4.10 which shows that the parallel design begins to plateau and almost stops increasing the computational time of image processing between 30,000 and 35,000. The gain of parallelizing the computation becomes almost larger than the cost of synchronization combined with the increase of data being processed.

The parallel design, as presented in Section 3.4.3, uses a block-level shared memory to store the unfinished cosine similarities temporarily. This sets a higher demand on the hardware as each block allocates some bytes for this staging buffer. This has shown throughout the development phase to potentially be too demanding as several of the group’s computers have not been able to run this design as the demand becomes too large at a certain amount of data that the computation on the GPU crashes.

Max Pool

An interesting finding, shown in the max pooling graphs in Figure 4.11 and Figure 4.12, is that the design which performs worse when it comes to calculating the cosine similarity performs better at max pooling. There could be several reasons for this, but it is most likely because max pooling requires the use of if-statements. As presented in Section 2.5.2 and shown in Figure 2.3 if-statements can result in thread divergence, taking a toll on the performance of the GPU.

Several if-statements and subsequently thread divergences will occur depending on the data being processed by the threads. As threads naturally diverge with an algorithm such as max pooling, the looped design with fewer threads will result in a higher portion of the calculation being idle. In contrast to the looped design, the parallel design will spawn more threads resulting in a smaller portion of the calculation being idle.

Synchronization Barriers

Both the max pooling and the cosine similarity calculations performed in parallel utilize a barrier for the synchronization of threads at the block level. This is one of the main reasons for the parallel cosine similarity design being slower than the looped design.

Why is the use of a barrier in the parallel max pooling design not visible? There can be multiple explanations for this, but it is most likely related to the dimensions that each design computes in. The parallel cosine similarity shader dispatches and spawns threads in two dimensions, x and y. This makes a block-level barrier synchronizing blocks x$\times$y times separately instead of just x times as in the parallel max pool design. As fewer blocks need to synchronize, less overall toll on processing time.

5.3.7 The Different Platforms

This subsection elaborates on the configuration of both platforms as shown in Table 3.1, and discusses their impact on the results from experiments presented in 3.5.2. Platform 1 is equipped with an i7-12700k CPU, featuring a hybrid architecture of 8 performance cores and 4 efficiency cores, totaling 12 cores. The 8 performance cores support hyperthreading, allowing for a total of 20 threads. The maximum clock speed is 4.9 GHz for its performance cores and 3.8 GHz for its efficiency cores. This CPU is made for high-performance consumer-grade usage that requires rapid processing. Platform 2 with a Xeon 6248R CPU, prioritizes stability and consistent performance over peak processing speed. It features 16 cores and 16 threads with a maximum clock speed of 4 GHz, making it optimized for long-duration tasks under continuous load. Due to being a rented node on NTNU's IDUN HPC cluster, it only has access to 16 of the actual 24 cores of the CPU.

The superior performance of Platform 1 in the sequential CPU experiments, displayed in 3.5.2, can simply be explained by the higher clock speed of its CPU compared to the CPU of Platform 2.

As for the CPU parallel part of the experiment, Platform 1 remained the better-performing platform. Even though the Xeon processor of Platform 2 boasts more total cores, the CPU of Platform 1 supports hyperthreading, resulting in a total of 20 threads. This combined with the higher clock speed of the performance cores makes Platform 1 perform better than Platform 2 also on CPU demanding tasks in parallel.

Platform 1 is equipped with an NVIDIA RTX 3080 GPU, which is a consumer-grade high-performance GPU. The GPU features high clock speeds and a vast amount of CUDA cores, making it suitable for fast processing tasks requiring fast processing.

In contrast, Platform 2, equipped with an NVIDIA A100 GPU, is designed for HPC. The A100 GPU, while having a lower clock speed, has a greater amount of Tensor cores than the RTX 3080 GPU. These cores, as mentioned in the subsection 2.5.4, are highly efficient for performing algorithms involving matrix operations.

CPU

Apart from the processing speed itself, the results from the CPU experiment graphs shown in Figure 4.1 and Figure 4.2 show that the implementations experienced similar trends relatively equally across the platforms. The Rust implementations showed a consistent standard deviation in run speed on both platforms. On the other hand, the Python implementation utilizing NumPy, showed an increasing standard deviation throughout the experiment. The pattern was not observed in the Rust implementation.

This variation in the Python implementation can be related to the underlying differences in how the Python interpreter interacts with the hardware. NumPy, primarily making system calls to underlying C-libraries, introduces an extra layer of abstraction to the interpretation compared to the Rust implementations. While the Rust implementations have been compiled with the release flag, making them directly optimized to maximize hardware utilization, NumPy does not necessarily get the same level of optimization. This could be what is giving the Rust implementations more stable and predictable performance measurements.

Another contributing factor may be the properties of the respective CPUs. While Platform 1 is equipped with a high-performance consumer-grade CPU, capable of executing at higher clock speed, it will also come with increased power consumption and heat generation. This can lead to thermal throttling, which makes the CPU automatically reduce its clock speed to prevent overheating. This will result in varying performance during intensive tasks.

Platform 2, utilizing a Xeon CPU, is designed for more consistent performance for longer durations. This architecture makes the CPU handle heat regulation and power consumption more efficiently, reducing thermal throttling and ensuring consistent performance levels.

This analysis suggests the CPU architectures and their handling of heat and power consumption are considerable factors that affect the variability in performance in the Python implementation, especially in scenarios with a higher amount of filters.

CPU versus GPU

Once again, one can observe the trends in the CPU graphs shown in Figure 4.3 and Figure 4.4. On Platform 1, the experiment starts with a consistent standard deviation which increases over time. On Platform 2, the standard deviation remains steady for the duration of the experiment, as previously explained.

As for the difference in GPU graphs across the platforms, it can be explained by the architectural differences between the GPUs. Generally, Platform 1 outperforms Platform 2, due to its higher clock speed on both the CPU and GPU, along with the higher amount of CUDA cores. This indicates an underutilization of the Tensor cores of the A100 GPU on Platform 2 during the experiments, which will be further discussed as a potential source of error in Section 5.3.8.

In the case of the Rust for-loop implementation, Platform 1 displays more variability in both performance and greater standard deviation throughout the experiment, compared to Platform 2. This variability can be due to the memory bandwidth differences described in Table 3.1. Despite having a lower clock speed, Platform 2, with its double bandwidth capability had a more stable performance throughout the experiment. This can likely be explained by the combination of high clock speed and lower bandwidth on Platform 1 leading to throttling on the GPU.

Regarding the C++ implementation using CUDA, it displays better performance on Platform 1 than on Platform 2. This is likely due to the higher amount of CUDA cores on the GPU. CUDA is developed by NVIDIA, potentially offering better abstraction and allowing for more efficient utilization of the GPU compared to wgpu. This results in greater performance and reduced standard deviation for the experiment on Platform 1 compared to Platform 2.

Although CUDA likely offers a more efficient abstraction than wgpu, the exact level of abstraction is unclear, making any comparison speculative.

Shader Strategy

The results from the experiments related to the performance of specific shaders are displayed in Figure 4.5 and Figure 4.6. The one-image-all-filters shader performed significantly worse than all the other shaders. Therefore, the group decided not to conduct further research on this shader and instead focused on the others. Due to this decision, no in-depth analysis was conducted to identify the cause of the poor performance.

For the other shaders, consistent processing times were observed between the CPU and the GPU. This consistency was due to the low number of images and filters used in this experiment. On Platform 2, both the Rust for-loop and One-image-all-filters maintained a low standard deviation throughout the experiment. In contrast, on Platform 1, these shaders experienced an increasing standard deviation over time, as discussed in Section 5.3.7.

Parallel versus Sequential Shader

The Optimized-one-image-all filters and Parallel implementations are designed to leverage concurrent computing on the GPU to speed up the processing but to achieve this they have dedicated temporary buffers specified in each block. These temporary buffers live and die with the block. Therefore, these implementations benefit from larger and faster memory to initiate more temporary buffers at the same time. This is because each block runs on a compute unit, and more compute units make it possible to run more blocks at the same time. Therefore a limited number of compute units and memory space needed for the temporary buffers will affect the computation time of this strategy.

Platform 2 has more memory, higher memory bandwidth, and more compute units than Platform 1, as shown in Table 3.1. This means more processing can be done in parallel with more demanding memory per block resulting in Platform 2 performing better at both the parallel-designed shaders shown in Figure 4.7 and Figure 4.8.

Continuing to inspect the same graph as the paragraph above, a huge difference between the two systems is spotted. The C++ implementation on Platform 2 deviates from the linear trend after the Rust runs stop. This was first thought to be a one-off result as the implementation has performed stable on Platform 1 throughout the whole project. However, further testing of the C++ implementation, shown in Figure 4.15 and Figure 4.16, still shows a similar variation on Platform 2 while Platform 1 remains stable. It is difficult to determine why Platform 2 varies so much after the Rust runs stop. A reason for this can be related to Platform 2 being a shared resource server. This may be why the second C++ run on Platform 2 is more stable as this run was processed at night time in contrast to day time. Another possible reason for this variation could be related to the memory aspect. There is probably a reason that the Rust runs stop, as it is because of a limitation set by the system by using the wgpu library. If this limitation is related to the system it could explain why the behavior becomes so unpredictable, but this also indicates that the C++ run on Platform 1 should have a larger variation which it does not. It is uncertain why the C++ run behaves as it does on Platform 2.

Inspecting all the parallel versus sequential graphs in Section 4.1.5 shows that Platform 1 performs worse than Platform 2 for the looped max-pooling shader. There are several potential reasons for this large difference in performance between the systems.

Firstly, Platform 1 has more cores per compute unit, resulting in more threads running in parallel at the same time. When performing many if-statements, having more threads in parallel within the same unit likely leads to more thread divergence, which could contribute to the performance gap.

Secondly, Platform 1 has fewer compute units than Platform 2. This means Platform 1 has fewer execution blocks running concurrently, as one block requires resources from a compute unit. The combination of increased thread divergence and fewer execution blocks running simultaneously could explain the large difference in performance between the platforms when processing the looped max-pooling shader.

5.3.8 Sources of Error

When testing algorithms one has to be aware of how results can be affected by real-world inconsistencies and human flaws. In this section, we discuss what types of inconsistencies and errors can affect the result of this project.

Hardware

Platform 1, which was one of the primary test environments, was a personal computer and not a part of a dedicated test bench. Although the platform was exclusively used for running experiments during the test phase, occasional use for other tasks was avoidable. This might have introduced minor fluctuations in processing time due to background processes consuming system resources. However, no resource-intensive applications that would affect the results were active during the test phase. Therefore, any impact on the overall results is expected to be minimal.

For Platform 2, as it was a part of the NTNU's IDUN HPC cluster it was not possible to monitor the platform in a way you can with self-owned hardware. Additionally, the group only had access to 16 of the 24 total cores of the CPU, limiting the possibility of fully controlling and observing the performance of the platform during the experiments.

Experience

Before the project, none of the group members had any experience with GPU programming, which resulted in a significant learning curve in the initial phase. Throughout the project, the members have studied the topic of GPU programming relentlessly to acquire as much knowledge as possible on the subject. A variety of theoretical sources were used, as described in the section 3.1.

Despite these efforts, a solution implemented by someone more proficient in GPU programming might result in different or perhaps more effective results.

Use of Cores

The experimental results highlight the significant performance advantages of the GPU implementations due to their superior parallel computing capabilities. Figures 4.3 and 4.4 in Section 4.1.2 illustrate that all the GPU implementations outperform their CPU counterparts.

What is more interesting is how Platform 1, equipped with an RTX 3080 GPU with 8704 CUDA cores, generally outperformed Platform 2. Despite Platform 2 having a greater number of Tensor cores, it did not result in better performance in the experiments.

This discrepancy indicates that CUDA cores play a more crucial role in accelerating the machine learning methods used in the experiments. Given that the A100, with its 432 Tensor cores specialized in tensor operations, generally underperformed compared to Platform 1, it suggests that these cores were underutilized in the test environment.

To give an exact number on the utilization of the specific type of GPU cores turned out to be challenging. Therefore, the insights provided are based on assumptions from the results, rather than from direct measurements.

This analysis underscores the importance of the architectural differences between CUDA and Tensor cores into account when attempting to improve machine learning algorithms through GPU parallelization.

6. Conclusion

The benefits of parallel programming are demonstrated in the results of this thesis. Even the simplest sequential implementation in Rust has a faster processing time than the Python implementation utilizing NumPy, which is considered the standard for implementing the machine learning method used in this thesis.

The results also indicate that GPGPU is well-suited for such highly parallelizable algorithms that involve only simple calculations. The parallel nature of the GPU is what makes it outperform the CPU even with a relatively low amount of data, as shown in Figure 4.1 and Figure 4.2. This potential that the GPU shows, compared to the CPU, can be utilized in many different ways. This thesis presents, implements, and tests several shader-design suggestions for utilizing the GPU for such algorithms in Rust. These different designs consist of simple implementations and more complex ones using synchronization barriers and temporary buffers. An in-depth comparison of these different designs is presented and discussed showing the simple shader design being the overall fastest implementation. The same shader is also implemented in C++ using CUDA, demonstrating even better performance here.

The potential of implementing machine learning methods, such as the one in this thesis on parallel hardware has shown good performance and makes it interesting to explore other parallel architectures. Therefore, expanding the exploration to FPGA and ASIC architectures, as this thesis intended to do, holds promise for unlocking similar performance gains and efficiencies as what the GPU shows. Harnessing the inherent parallelism of these specialized hardware platforms could potentially lead to even more optimized and efficient implementations of algorithms, as demonstrated in this thesis.

6.1 Future Work

There are plenty of unexplored directions and aspects related to the topics of this thesis that would benefit from being further explored. Firstly, if the implementations used in this thesis were to be continued, then an exploration into implementing and testing the algorithm with larger images and filters to gauge the potential there is an interesting improvement.

This thesis focuses on the parallelization of individual calculations based on cosine similarity which has potential for future machine learning methods. Future work includes the practical application of the parallelized cosine similarity in machine learning methods themselves and exploring their potential. This involves developing and testing a model using the new Gabor-like filters for image classification like Appendix A tried to achieve with Gabor filters. An interesting extension is to introduce noise into the images being classified, as Gabor filters have shown promising results in this area (Dakshayani et al., 2022).

A more time-consuming but potentially the most interesting direction is to explore the potential of implementing the algorithm on an integrated circuit such as an FPGA and or ASIC. These implementations will have less overhead compared to those on either the CPU or GPU. This is because, instead of programming code for the hardware to run, the hardware itself is ”programmed” and designed to take input, process it in parallel, and then produce the output. There is a lot of potential and exploration needed in such a direction since the performance of a FPGA or ASIC seems promising but has not yet been tested with the algorithm addressed in this thesis.

While working on this thesis, it was discovered early on that the current GPU implementations likely do not utilize Tensor cores effectively or at all for the matrix operations. This became evident because Platform 2, which has more Tensor cores, generally performed worse than Platform 1, which has more CUDA cores. Given that Tensor cores are generally faster for the types of operations used in this thesis, the performance gap should not be that significant. Therefore, there is potential in optimizing the algorithms used in this thesis to better leverage Tensor cores, either instead of or in combination with CUDA cores, as the algorithm prioritizes speed over accuracy.

This thesis has delved into mostly Rust but some C++ and Python. Because of the lack of exploration in languages other than Rust, there is some potential exploration to be done here by looking at other languages and frameworks and then gauging their performance. In addition, coming back and testing the algorithm with a newer version of the libraries used in this thesis when they are developed could also be interesting as the main library in Rust wgpu has yet to publish version one.

Lastly, as the algorithm used in this thesis relies heavily on memory exploring the potential of optimizing the overall memory usage is an interesting direction for further work. This is especially interesting with the current Rust implementations as they all stop at around 53,500 filters because of memory exceeding. This is not entirely correct as there is more memory on the system, but either wgpu is not optimized enough, or other strategies need to be implemented.

7. Societal Impact

Machine learning and deep learning have risen in popularity the recent years, leading to an exponential growth in training data, model parameters, and system resources. This requires more computational power to train the models for better accuracy. However, this increased demand for power negatively impacts the carbon footprint associated with the infrastructure, development, and inference of AI models.

For example, in 18 months, Meta increased its infrastructure capacity by 2.9 times for training models and 2.5 times for inference (Wu et al., 2022). The AI model Meena had a carbon footprint equivalent to an average passenger vehicle driving 242,231 miles. However, to understand the carbon footprint one must have a holistic approach, and not just look at one aspect such as training the model but include the infrastructure created for both training and deploying these models (Wu et al., 2022).

7.1 Environmental impact of AI

This thesis focuses on optimizing the calculation of a potential new machine-learning model using Gabor-like filters, which can reduce both training time and power consumption. This is based on the findings that Gabor filters in the first layer and a mix of Gabor filters and CNN kernels deeper in the model has little impact on the accuracy of the model and the same time reduces training time and power consumption (Sarwar et al., 2017). This is because the biggest power consumption stems from back-propagation, which involves gradient computation and weight updates of the convolutional and fully connected layers.

The study concludes that specialized hardware can have beneficial results in this mixture of Gabor filters and CNN kernels (Sarwar et al., 2017). The new machine learning model that can be developed based on this thesis can help with reducing the training time further. This is because the new model is flat and does not require back-propagation, as the Gabor-like filters are fixed and have no training time.

While this thesis uses only the GPU to test computational power without measuring power consumption, dedicated hardware such as an ASIC would likely offer higher computational power and lower power consumption due to the fixed nature of Gabor-like filters, similar to the conclusion of previous studies (Sarwar et al., 2017). Efficient use of the hardware and energy in the development and use of AI aligns with the United Nations (UN) Sustainable Development Goal 12 (Responsible Consumption and Production), promoting sustainable infrastructure and development of the AI industry. It also supports Norway's goal of reducing total greenhouse gas emissions by 50% to 55% compared to 1990 levels (Norsk regjering, 2023) (FN-sambandet, 2023a).

7.2 Economic impact

Reducing energy consumption in data centers can result in significant cost savings on electricity. This also has the potential to reduce the workload on non-optimal hardware in AI, leading to a more sustainable infrastructure. Such infrastructure lasts longer and saves resources and money by avoiding the need to build unnecessary large centers. By using optimized hardware that utilizes the energy better and with higher computational power for that specific task, data centers can operate more efficiently.

Google's development of a specialized ASIC, known as a Tensor Processing Unit (TPU) has successfully achieved these goals. This development aligns with Amdahl's Law which states: ”low utilization of a huge, cheap resource can still deliver high, cost-effective performance.” (Jouppi et al., 2017)

7.3 AI in medecine

Recent advancements in deep learning have shown promising results in medicine, especially in tasks where image analysis is important for diagnostics, such as radiology. Models are often trained on a labeled dataset, but they can also be trained on unlabeled data consisting of numbers and text. This has improved other fields in medicine such as biochemistry, and genomics, and predicted outcomes from medical signal data, such as EEG, electrocardiogram, and audio data. These models have proven the ability to draw conclusions based on data patterns that humans normally would not see (Rajpurkar et al., 2022). In radiology, image noise is an inevitable challenge. The Gabor filter has proven effective at feature extraction on noisy images images (Dakshayani et al., 2022). With the use of the new Gabor-like filters a new image recognition model can potentially improve the use of machine learning in medicine by analyzing images in radiology and similar fields of image analysis. This correlates to UN sustainable development goal 3 (Ensure healthy lives and promote well-being for all at all ages), which focuses on improving medicine research and the overall health of every age. (FN-sambandet, 2023b)

7.4 Data integrity and research benefits

The results are based on statistical models that ensure data integrity, as explained in Section 3.5. The group has used and followed the structure of DSR to present trustworthy results that can be used for further research. This approach ensures the group remains non-biased and does not cherry-pick results to better fit their desired outcomes. Cherry-picking can lead to incorrect results, wrong conclusions, and wasted resources.

The implementations have also been tested to ensure the calculations are performed correctly and that the data is representative of an actual model. This ensures a non-biased interpretation of the result and reliable testing.

Bibliography

Acken, K., Irwin, M., & Owens, R. (1998). A parallel asic architecture for efficient fractal image coding. *The Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, *19*, 97–113. https://doi.org/10.1023/A:1008005616596

Anderson, T., & Dahlin, M. (2015). *Operating systems: Principles and practice, volume ii: Concurrency* (2nd ed.). Recursive Books, Ltd. http://recursivebooks.com/

Ansorge, R. (2022). *Programming in parallel with cuda: A practical guide*. Cambridge University Press.

ARM. (2024a). *Asic (application-specific integrated circuit)*. Retrieved April 12, 2024, from https://www.arm.com/glossary/asic

ARM. (2024b). *CPU*. Retrieved April 19, 2024, from https://www.arm.com/glossary/cpu

ARM. (2024c). *Fpga (field-programmable gate array)*. Retrieved April 15, 2024, from https://www.arm.com/glossary/fpga

Brocke, J. v., Hevner, A., & Maedche, A. (2020, September). Introduction to design science research. https://doi.org/10.1007/978-3-030-46781-4_1

Calculating confidence intervals [Accessed: May 11, 2024]. (n.d.). Retrieved May 11, 2024, from https://amsi.org.au/ESA_Senior_Years/SeniorTopic4/4h/4h_2content_11.html

Dakshayani, V., Locharla, G. R., Pławiak, P., Datti, V., & Karri, C. (2022). Design of a gabor filter-based image denoising hardware model. *Electronics*, *11*(7), 1063. https://doi.org/10.3390/electronics11071063

Daugman, J. G. (1985). Uncertainty relation for resolution in space, spatial frequency, and orientation optimized by two-dimensional visual cortical filters. *J. Opt. Soc. Am. A*, *2*(7), 1160–1169. https://doi.org/10.1364/JOSAA.2.001160

Deng, L. (2012). The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, *29*(6), 141–142. https://doi.org/10.1109/MSP.2012.2211477

Devakumar, D., & Eidheim, O. C. (2024). Multidimensional gabor-like filters derived from gaussian functions on logarithmic frequency axes.

Eidheim, O. C. (2022). Revisiting gaussian neurons for online clustering with unknown number of clusters.

Eidheim, O. C. (2023, September). Maskinlæring: Convolutional neural networks [Published 8. september 2023, Lecture Notes from Blackboard].

ENCCS. (2024). *GPU Programming: When, Why and How?* [Accessed: April 8, 2024]. https://enccs.github.io/gpu-programming/4-gpu-concepts/

Fatahalian, K. (2011). *How a gpu works* [Lecture slides for CMU class 15-462, accessed April 19, 2024]. Carnegie Mellon University. https://www.cs.cmu.edu/afs/cs/academic/class/15462-f11/www/lec_slides/lec19.pdf

FN-sambandet. (2023a). Ansvarlig forbruk og produksjon [Last updated: 02.05.2023. Accessed: May 11, 2024]. https://fn.no/om-fn/fns-baerekraftsmaal/ansvarlig-forbruk-og-produksjon

FN-sambandet. (2023b). God helse og livskvalitet [Last updated: 15.09.2023. Accessed: May 11, 2024]. https://fn.no/om-fn/fns-baerekraftsmaal/god-helse-og-livskvalitet

Gupta, S., Agrawal, A., Gopalakrishnan, K., & Narayanan, P. (2015, July). Deep learning with limited numerical precision. In F. Bach & D. Blei (Eds.), *Proceedings of the 32nd international conference on machine learning* (pp. 1737–1746, Vol. 37). PMLR. https://proceedings.mlr.press/v37/gupta15.html

Høvding, G., & Sandvig, K. (2020, December). *Synsfelt* [Last updated December 1, 2020. Accessed: April 23, 2024]. Store medisinske leksikon. https://sml.snl.no/synsfelt

Jeon, W., Ko, G., Lee, J., Lee, H., Ha, D., & Ro, W. W. (2021). Chapter six - deep learning with gpus. In S. Kim & G. C. Deka (Eds.), *Hardware accelerator systems for artificial intelligence and machine learning* (pp. 167–215, Vol. 122). Elsevier. https://doi.org/https://doi.org/10.1016/bs.adcom.2020.11.003

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., … Yoon, D. H. (2017). In-datacenter performance analysis of a tensor processing unit. *45*(2), 1–12. https://doi.org/10.1145/3140659.3080246

Khandelwal, V. (2020). The architecture & implementation of alexnet [Originally published in Analytics Vidhya]. *Analytics Vidhya*. Retrieved April 19, 2024, from https://medium.com/analytics-vidhya/the-architecture-implementation-of-alexnet-d90b137d93b5

Kılıç, İ. (2023). What is max pooling and why do we need max pooling? [Accessed: 2024-05-19]. https://medium.com/@ilyurek/what-is-max-pooling-and-why-do-we-need-max-pooling-57247a3fbca9

Levy, D. (2012). *Introduction to numerical analysis* [Chapter 3, pages 19-54]. University of Maryland.

Markovic, D., Chang, C., Richards, B., So, H., Nikolic, B., & Brodersen, R. W. (2007). Asic design and verification in an fpga environment. *2007 IEEE Custom Integrated Circuits Conference*, 737–740. https://doi.org/10.1109/CICC.2007.4405836

Momeny, M., Latif, A. M., Agha Sarram, M., Sheikhpour, R., & Zhang, Y. D. (2021). A noise robust convolutional neural network for image classification. *Results in Engineering*, *10*, 100225. https://doi.org/https://doi.org/10.1016/j.rineng.2021.100225

Navarro, C. A., Carrasco, R., Barrientos, R. J., Riquelme, J. A., & Vega, R. (2021). Gpu tensor cores for fast arithmetic reductions. *IEEE Transactions on Parallel and Distributed Systems*, *32*(1), 72–84. https://doi.org/10.1109/TPDS.2020.3011893

Navidi, W. C. (2011). *Statistics for engineers and scientists* (3rd). McGraw-Hill.

Nikolai Vazquez and Josh Stone. (2024). *Rayon: A data parallelism library for Rust*. Retrieved May 1, 2024, from https://docs.rs/rayon/latest/rayon/

Norsk regjering. (2023). Klimaendringer og norsk klimapolitikk [Last updated: 28.08.2023. Accessed: May 11, 2024]. https://www.regjeringen.no/no/tema/klima-og-miljo/innsiktsartikler-klima-miljo/klimaendringer-og-norsk-klimapolitikk/id2636812/

Patterson, D. A., & Hennessy, J. L. (2014). *Computer organization and design: The hardware/software interface* (5th ed.). Elsevier.

Plancher, B., Neuman, S. M., Bourgeat, T., Kuindersma, S., Devadas, S., & Reddi, V. J. (2021). Accelerating robot dynamics gradients on a cpu, gpu, and fpga. *IEEE Robotics and Automation Letters*, *6*(2), 2335–2342. https://doi.org/10.1109/LRA.2021.3057845

plancherb1. (2024). Fast rbd gradients.

Rajpurkar, P., Chen, E., Banerjee, O., & Topol, E. J. (2022). Ai in health and medicine [Published 20 January 2022, Received 23 July 2021, Accepted 5 November 2021]. *Nature Medicine*, *28*(1), 31–38. https://doi.org/10.1038/s41591-021-01614-0

Robey, R., & Zamora, Y. (2021). *Parallel and high performance computing*. Manning Publications Co.

Rolstadås, A. (2022). Lean [Accessed: May 10, 2024 from Store norske leksikon. Last updated: October 5, 2022]. https://snl.no/lean

Sarwar, S. S., Panda, P., & Roy, K. (2017). Gabor filter assisted energy efficient fast learning convolutional neural networks. *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 1–6. https://doi.org/10.1109/ISLPED.2017.8009202

Semiconductor, L. (2024). *What is an fpga?* Retrieved April 16, 2024, from https://www.latticesemi.com/en/What-is-an-FPGA

Stewart, M. (2019). Simple introduction to convolutional neural networks [Accessed: 2024-05-19]. *Towards Data Science*. https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac

Syberfeldt, A., & Ekblom, T. (2017). A comparative evaluation of the GPU vs. the CPU for parallelization of evolutionary algorithms through multiple independent runs [Available at SSRN: https://ssrn.com/abstract=3937048]. *International Journal of Computer Science & Information Technology (IJCSIT)*, *9*(3).

Tanenbaum, A. S. (2013). *Structured computer organization* (6th ed.). Pearson.

Truong, L., & Hanrahan, P. (2019). A golden age of hardware description languages: Applying programming language techniques to improve design productivity. *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, *136*, 7:1–7:21. https://doi.org/10.4230/LIPIcs.SNAPL.2019.7

Understanding cpu clock speed [Accessed: 2024-05-07]. (2024). *Intel*. https://www.intel.com/content/www/us/en/gaming/resources/cpu-clock-speed.html

University of Newcastle Library. (2024). Research methods: What are research methods? [Accessed: 2024-04-24]. *University of Newcastle, Australia*. https://libguides.newcastle.edu.au/researchmethods

University of Oslo, Institute of Biosciences. (2019, January). *Sanseresptorer* [Published: January 14, 2019, 10:50 - Last modified: May 6, 2020, 10:47, Accessed: April 19, 2024]. https://www.mn.uio.no/ibv/tjenester/kunnskap/plantefys/leksikon/s/sansereseptorer.html

Wang, J., Turko, R., Shaikh, O., Park, H., Das, N., Hohman, F., Kahng, M., & Chau, P. (2024, April). What is a convolutional neural network? [Research collaboration between Georgia Tech and Oregon State University. Supported in part by NSF grants IIS-1563816, CNS-1704701, NASA NSTRF, DARPA GARD, gifts from Intel, NVIDIA, Google, Amazon.]. *Georgia TechOregon State University*.

wgpu Team. (2024). Wgpu – rust graphics api [Accessed: 2024-04-22]. https://wgpu.rs/

World Wide Web Consortium (W3C)). (2023). WebGPU Shading Language Specification [Accessed: April 23, 2024]. https://www.w3.org/TR/WGSL/#intro

Wu, C.-J., Raghavendra, R., Gupta, U., Acun, B., Ardalani, N., Maeng, K., Chang, G., Aga, F., Huang, J., Bai, C., Gschwind, M., Gupta, A., Ott, M., Melnikov, A., Candido, S., Brooks, D., Chauhan, G., Lee, B., Lee, H.-H., … Hazelwood, K. (2022). Sustainable ai: Environmental implications, challenges and opportunities. In D. Marculescu, Y. Chi & C. Wu (Eds.), *Proceedings of machine learning and systems* (pp. 795–813, Vol. 4). https://proceedings.mlsys.org/paper_files/paper/2022/file/462211f67c7d858f663355eff93b745e-Paper.pdf

Xilinx Inc. (2024). Programming an fpga: An introduction to how it works [Accessed: 2024-05-07]. https://www.xilinx.com/products/silicon-devices/resources/programming-an-fpga-an-introduction-to-how-it-works.html

Zafar, A., Aamir, M., Nawi, N. M., Arshad, A., Riaz, S., Alruban, A., Dutta, A. K., & Almotairi, S. (2022). A comparison of pooling methods for convolutional neural networks. *Applied Sciences*, *12*(17), 8643. https://doi.org/10.3390/app12178643

A. Machine Learning Project

CNN Alternative - The Gabor filter

Aleksander Olsvik *alekol@stud.ntnu.no*
Department of Computer Science
Norwegian University of Science and Technology

Jonatan Andre Vevang *jonatanv@stud.ntnu.no*
Department of Computer Science
Norwegian University of Science and Technology

Stian Wilhelmsen *stianwil@stud.ntnu.no*
Department of Computer Science
Norwegian University of Science and Technology

Vetle Ålesve Nordang *vetlean@stud.ntnu.no*
Department of Computer Science
Norwegian University of Science and Technology

Ole Christian Eidheim *ole.c.eidheim@ntnu.no*
Department of Computer Science
Norwegian University of Science and Technology

Abstract

In CNN, all the filters used are learned through back-propagation. The goal of this project was to generate fixed Gabor-filters and use the results of these to classify the MNIST-digits through max-pooling and K-means clustering. Studying the Gabor-filters and improving it's efficiency is interesting as it's widely used in image analysis, and it's given a considerable amount of attention because they closely resemble the human visual system. And because the filter are aimed at mimicking the visual cortex, it also tries to mould itself to suit different scenarios where it does not perfectly fit.(Joshi, 2014) The results of this study shows that Gabor-filters can be implemented with quite a simple model that works well for image recognition that is not very susceptible to noise. If you want to try and reproduce the results, you can do so by looking at the repository[1] for the project.

[1]The source code to reproduce the results can be found at https://gitlab.stud.idi.ntnu.no/idatt25021/cnn

Contents

1 Introduction

Authors have claimed that simple cells in the visual cortex of mammalian brains can be modeled by Gabor functions.(Daugman, 1985)(Marčelja, 1980) Thus, image analysis with Gabor-filters is thought by some to be similar to the perception in the human visual system and they can therefore be named *biologically based Gabor-filters*. The Gabor-filters are orientation-sensitive filters, used for edge and texture analysis. The way a Gabor-filter works is that it detects different shapes, sizes and smoothness levels in the image, and the filter can be viewed as a sinusoidal plane of a particular frequency and orientation.(Kafuo A. & Gonifeda A., 2017)

Normally with CNN, you would use few filters along with back-propagation to classify pictures. In this assignment the group has worked on creating various Gabor-filters to attempt to make it less susceptible to noise, as this is a known flaw of CNN. It's these filters that have been applied to the MNIST dataset, with max pooling and K-means clustering. The MNIST dataset is a collection of handwritten digits (0 through 9) and is commonly used for training various image processing systems. The Gabor-models from this study have been trained and tested on this dataset.

2 Related Work

Gabor-filters are generally used for feature extraction and texture analysis in image processing. More specifically, these filters can be used for noise removal and edge detection in images. The way these filters work have been compared to that of the human visual system and how our eyes process what they see. This is due to the properties of such filters, with their spatial locality and orientation. (Olshausen & Field, 1996)

There has been prior research as to evaluate the effects of Gabor-filter parameters on texture classification. *In texture classification, in particular, Gabor filters show a strong dependence on a certain number of parameters, the values of which may significantly affect the outcome of the classification procedures.*(Bianconi F., 2007) From this study it indicates that the frequency and gamma values of the filters affect the results the most.

Due the Gabor-filter's rise in popularity there's been a lot of biologically inspired research, a field inspired by how biology works, such as object recognition. William Hamilton wrote about the importance of biologically inspired object recognition. He states that the biological vision systems are still far superior to the current state-of-the-art computer vision systems.(Hamilton, 2013) Nevertheless, Gabor-filters are still important due to their ability to closely mimic the human visual processing and it's ability to adapt. Therefore, the Gabor-filters show a lot of potential for different applications. Hamilton used two different algorithms combined together with Gabor-filters, one that is supposed to mimic biological systems, and one that didn't. His results made it clear that the biological system got a lot better results, and strengthens the theory that these filters work in similar ways to out eyes. His object recognition algorithm was tested on the Caltech-101 data set which contains pictures belonging to 101 categories.

The neurons in the visual cortex has a similar way of working to max pooling, though slightly different. Both the neurons and max-pooling share conceptual similarities of feature selection, in-variance and efficiency in handling information. They aren't identical, but can be compared to draw parallels between biological neural processing and artificial operations. Cells in the primary visual cortex of higher mammals can be separated in two groups, simple and complex cells, where simple cells sends signals to complex cells and the complex cells select the strongest signals from the simple cells.(Boutin et al., 2022) K-means clustering has been used in the study to summarize a large amount of data under the assumption that the clustered data set centroids gives a summarizing of the clustered data set. Although not traditional, this use of clustering is a way of reducing the size of the data while minimizing data loss. (Ahmed & Mahmood, 2014)

3 Methods

Throughout the project there were several iterations of the method used for feature extraction. In this section the description is for each main iteration.

The key concept of all of them is how the Gabor-filters are applied on the image to extract its features. The filters are placed at a given point in the pictured, and a scalar value is computed from the area the filter "sees". The computation of this value is done by the formula below.

$$\cos\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \times \|\mathbf{b}\|}$$

Through the project there were used 216 different filters in all of the iterations, with different parameters in theta, lambda, sigma, gamma and psi to create different filters. For each of the iterations, as all of them uses K-means clustering, the group has attempted to find the optimal amount of clusters through testing. This will be further discussed in the result section.

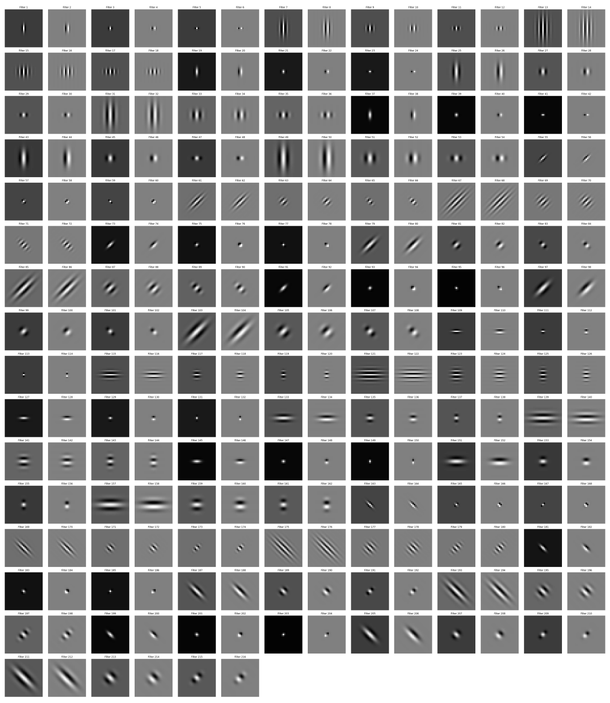


Figure 1: All 216 Gabor-filters generated. Each of the parameters have been given a set of values, and the filters are the result of the Cartesian product of these parameters

3.1 First Iteration - Simple Gabor

The first iteration used the 216 filters in the center of the padded image from 28x28 to 29x29. That is done because the library that produce the Gabor-filters only generate in odd numbers and made 29x29 sized filters. Then the images are normalized by dividing by 255 to convert pixel values to a range of 0.0 to 1.0. This is a common practice in image processing to work with standardized floating-point numbers. The filters were then applied and made a list of length 216 containing a scalar value from each filter. This list contains the features of the image. Then, for every digit, the model uses K-means clustering with k-amount of clusters to reduce the 216 dimensional space from 5421 to k data points. These cluster-centers describe a single digit of the trained model, and is applied to each of the digits.

The model predicts what number the image is by applying the filters and plotting the features in the 216 dimensional space, then find the closest cluster-center among all the digits' cluster-centers for that image's features. This is the main reason for using K-means clustering to reduce the size of the trained data, and thus the amount of checks.

3.2 Second Iteration - Gabor Mesh

The third iteration added two new concepts. The model was expanded to apply the filters in a honeycomb structure around the center of the image. Six new points were added with approximately equal distance to the center. For future reference, these points will be referenced to as centers. To apply the filters on these new centers, the image is further padded to 57x57. When a filter is applied to a shifted center, the image to be filtered is cut down to 29x29 again, with a new center as shown below.

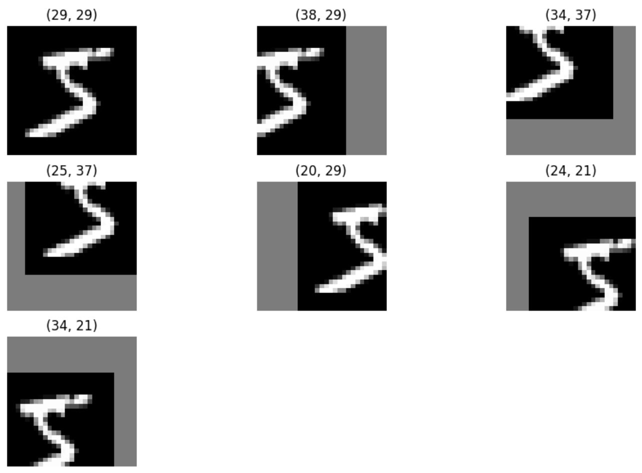


Figure 2: Image containing the digit 5, applied with honeycomb structuring. The gray parts are only for visualization, and are actually black when the filters are applied

The filters are then applied to this shifted image for each center. This produces 216 features for each image, per center. To reduce the dimensionality and increase how well the images are described, the model applied max pooling on the filtered values to reduce each image down to four features per center after filtering. For our model, this is equal to one descriptive feature per orientation. These orientations correspond to horizontal lines, vertical lines and diagonal lines in both directions. The result of each max pooling for each center per image is concatenated back into an array of 28, which is equal to four features times seven clusters. K-means clustering is used in the same way as before for testing and reduction of the data.

3.3 Third Iteration - Gabor Mesh 5x5

For the third iteration, the Gabor-model is extended in regard to how the filters are applied. For each of the centers, the image is applied in a 5x5 grid, with center in each of these pixels. The area of this grid for each center is shown in Figure 3.

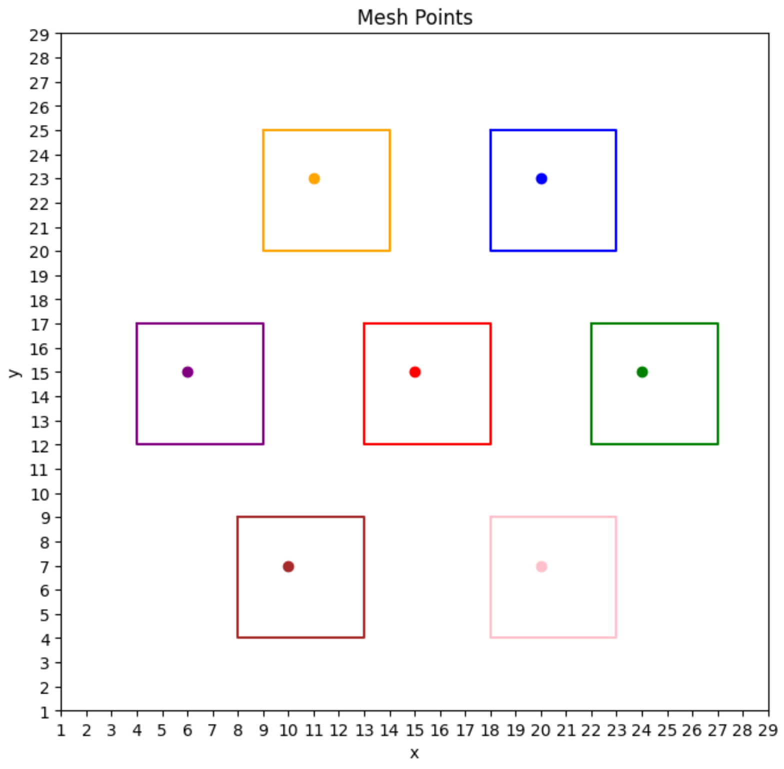
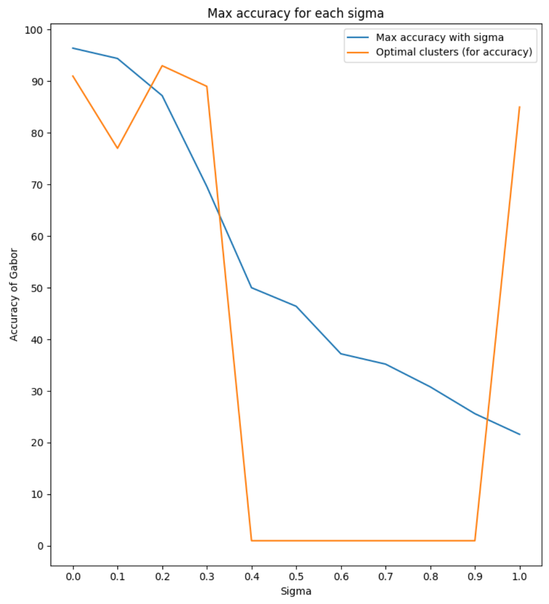


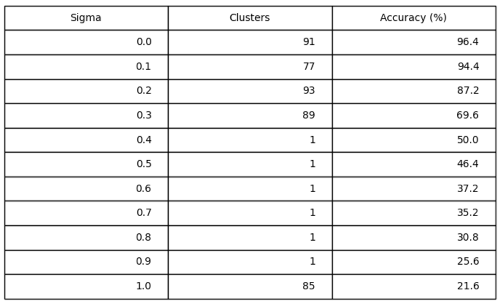
Figure 3: Honeycomb grid structure

For each of these centers, the filters are applied 25 (5x5) times. The maximum value, meaning the optimal position of that filter, is then selected through max pooling. This is then added to the list as normal, and the result of the filtration is still 216 features. Each of these features are thought to be optimized for the given filter and center. This is then processed with max pooling and K-means clustering as in the previous iteration.

4 Results and Discussion

As stated in the methods section, the optimal amount of clusters were found through testing. This was done for both the Gabor-filter with and without noise. The results conclude that using a total of 91 clusters for K-means clustering is most optimal when using Gabor-filter Mesh 5x5. However using 66 clusters is most optimal when using Gabor-filter with Mesh. This result was found by training the model and testing the images with different values for k in K-means clustering. These tests were run again on the Gabor-model with Mesh 5x5 with increasing levels of sigma. This was doen to further improve our model when dealing with higher levels of noise. The results of these tests are shown in the plot below, with the values in the table. The levels of noise for each sigma is also displayed.





Sigma	Clusters	Accuracy (%)
0.0	91	96.4
0.1	77	94.4
0.2	93	87.2
0.3	89	69.6
0.4	1	50.0
0.5	1	46.4
0.6	1	37.2
0.7	1	35.2
0.8	1	30.8
0.9	1	25.6
1.0	85	21.6

Figure 4: Most optimal amount of clusters for different levels of noise

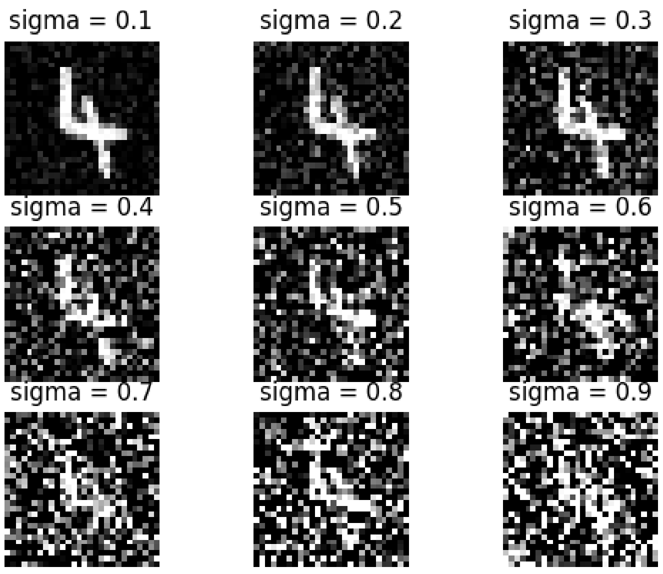


Figure 5: Different levels of sigma/noise applied to an image containing the letter four.

After receiving the optimal amount of clusters, three different Gabor-models were tested and compared to a normal CNN model. The three different Gabor-models differ in the way they apply filters to the images. The three models are the results of the three iterations of the study. How filters are applied in each of these models are described in the Methods section.

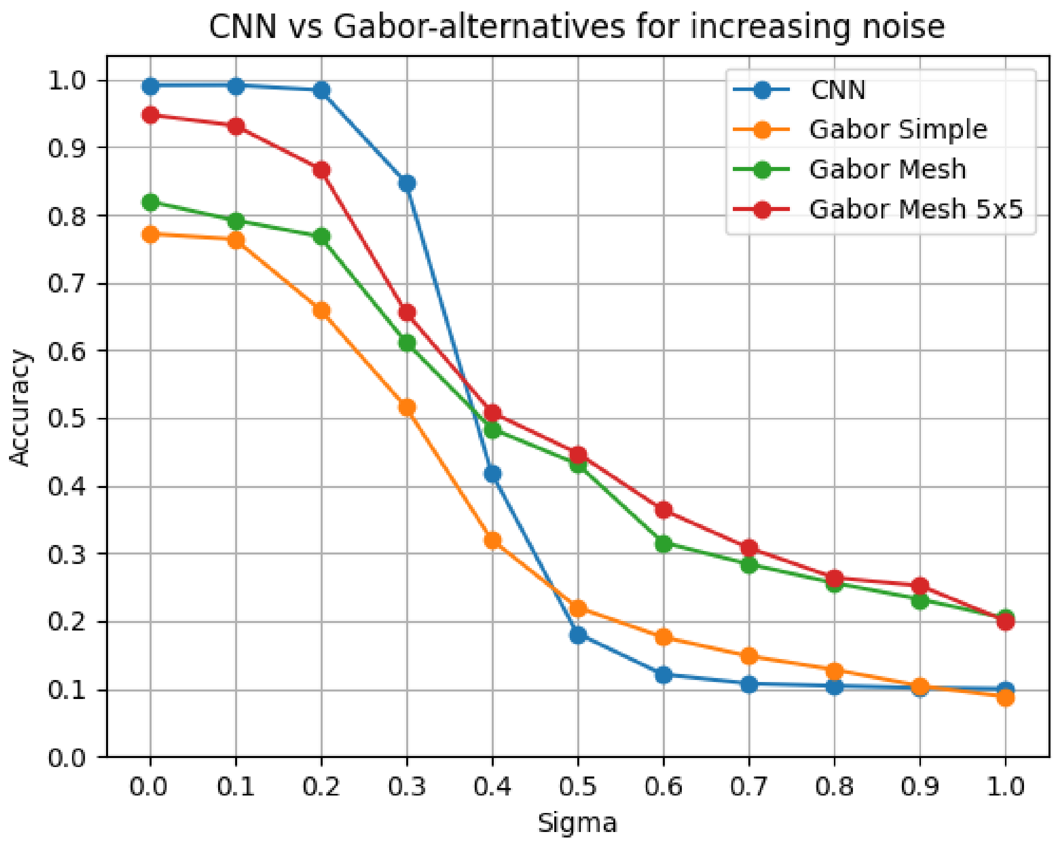


Figure 6: Here are the accuracy's of the different models with increasing noise. CNN is trained on 60000 images and tested on 10000, while the different Gabor models are trained on 54210 images and tested on 250 images. The accuracy's are in decimal, and the sigma values are the standard deviation of the Gaussian normal distribution.

CNN outperforms the Gabor-models when applying noise < 0.4 to the test pictures according to the group's study. However, as seen on the result plot above, the Gabor-model with mesh and Gabor-model with mesh 5x5 outperforms CNN as soon as the applied noise is over sigma=0.4.

To visualize the data from the model, the group used t-SNE for the cluster-centers and a summarization in the Fourer room for the filters. These plots describe how active our filters are in different regions, and how closely the images are clustered together.

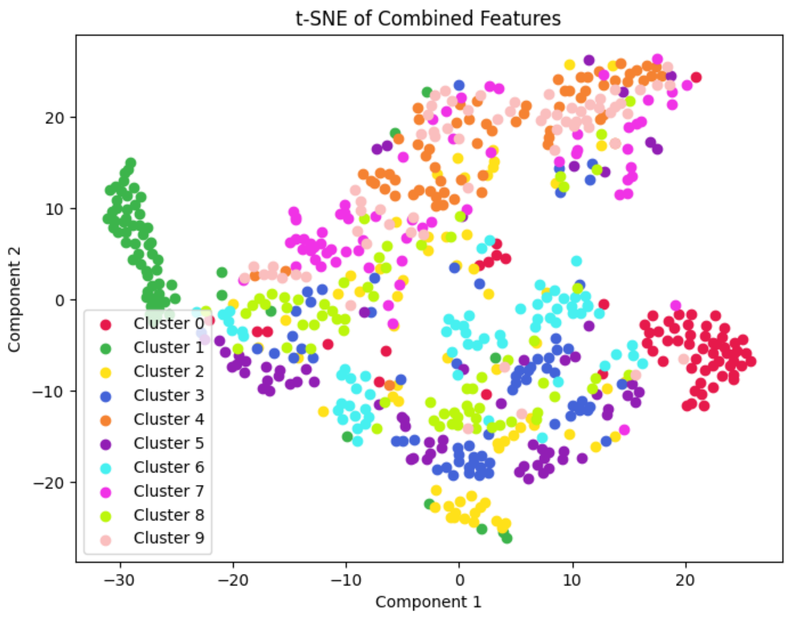


Figure 7: t-SNE plot. This visualizes a multidimensional space, which are the features of the images from the K-means clustering. Here we can see some clear clusters for the different images

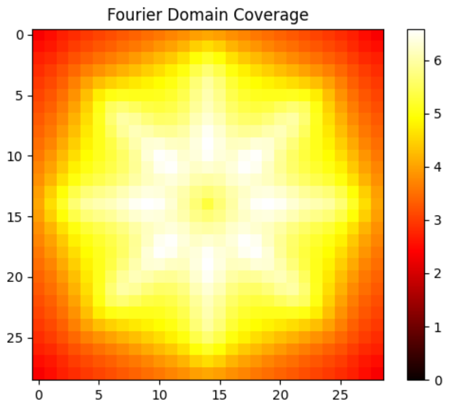


Figure 8: This is a Fourier domain coverage. It visualizes where the filters are most active, and one can see that the angles used in the filters show the most activation. The colors in the plot describe how active the sum of filters are, with lighter colors showing more activation.

5 Conclusion and Future Work

The results in the previous section show a working Gabor-model that is less affected by noise than CNN. The final model also achieves quite high accuracies for the MNIST digits without noise, with 96.4% being the best result. The final results were quite good for such a simple model with "only" 216 filters used. Although it's not the most complex dataset, this shows the potential of what filters like this can do.

There are many ways of improving the performance of this specific model, both in terms of speed and accuracy. In terms of speed, there are probably many clever ways of optimizing the performance. For the groups's Python model, better use of numpy is the most obvious one. Numpy was used for many of the matrix operations, but probably not optimally. To make up for a slow training- and testing process, multi-threading was softly implemented. This is also another point of improvement for this Python implementation. Ideally, one would consider using another programming language for this task, such as C. Python was chosen due to experience with it and an expectation of using more libraries for the Gabor-filters and such. This turned out to be less used than expected, and therefore not necessary.

For the performance of the model in terms of accuracy, the group discussed increasing the amount of filters, with the amount of orientations being the main focus. This can improve how the filters detects edges and identifies different numbers even more. Looking more at the honeycomb structure is also a part that could be subject to improvement. Increasing the amount of honeycomb centers and changing the value of the radius between them could potentially improve how and where the filters find their features. For future work, one could also look more into optimal clusters. This was found to drastically improve the accuracy with a simple optimization technique, and can probably become even better. Looking more into max pooling, and taking other factors into considerations here could also be interesting to look at. The possibilities are endless.

References

Mohiuddin Ahmed and Abdun Naser Mahmood. Clustering based semantic data summarization technique: A new approach. oct 2014. URL https://ieeexplore.ieee.org/abstract/document/6931456/references#references.

Fernandez A. Bianconi F. Evaluation of the effects of gabor filter parameters on texture classification. *sciencedirect*, 2007. URL https://www.sciencedirect.com/science/article/abs/pii/S003132030700218X.

Victor Boutin, Angelo Franciosini, Frédéric Chavane, and Laurent U. Perrinet. Pooling strategies in v1 can account for the functional and structural diversity across species. *PLOS Computational Biology*, jul 2022. URL https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1010270.

John G. Daugman. Uncertainty relation for resolution in space, spatial frequency, and orientation optimized by two-dimensional visual cortical filters. *ournal of the Optical Society of America A*, 1985. URL https://opg.optica.org/josaa/abstract.cfm?uri=josaa-2-7-1160.

William Hamilton. Biologically inspired object recognition using gabor filters. *cim*, 2013. URL https://www.cim.mcgill.ca/~siddiqi/COMP-558-2012/willhamilton.pdf.

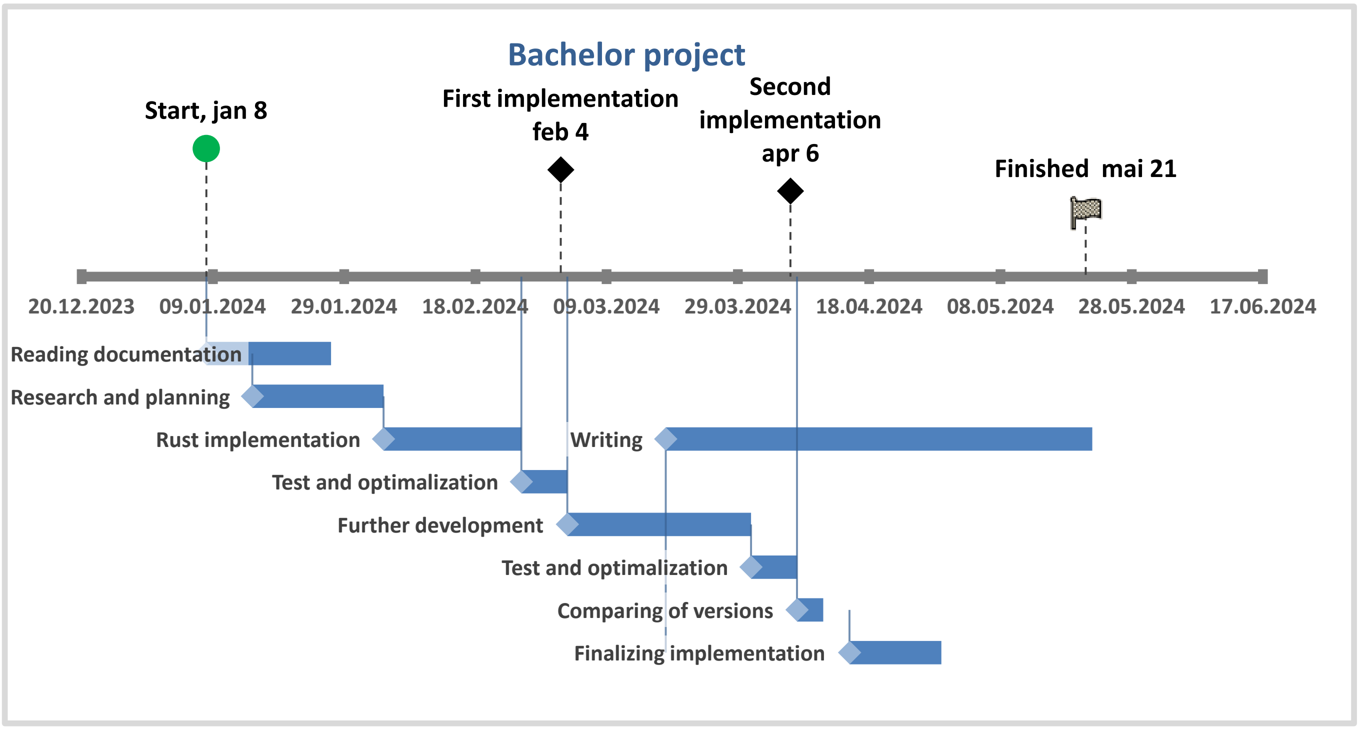
Prateek Joshi. Understanding gabor filters. *Perpetual Enigma*, 2014. URL https://prateekvjoshi.com/2014/04/26/understanding-gabor-filters/.

Diaf S. Kafuo A. and Alshatouri Z. Gonifeda A. A literature survey of gabor filter and its application. 2017. URL https://www.researchgate.net/publication/317257288_A_Literature_survey_of_Gabor_filter_and_its_application.

S. Marčelja. Mathematical description of the responses of simple cortical cells. *Journal of the Optical Society of America*, 1980. URL https://opg.optica.org/josa/abstract.cfm?uri=josa-70-11-1297.

Bruno Olshausen and David Field. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, 07 1996. URL https://www.nature.com/articles/381607a0.

B. Gantt chart



Oppgaver

Start	Slutt	Varighet	Etikett	Vert. Posisjon	Vert. Linje
08.01.2024	26.01.2024	19	Reading documentation	-18	-18
15.01.2024	03.02.2024	20	Research and planning	-28	-10
04.02.2024	24.02.2024	21	Rust implementation	-38	-10
25.02.2024	02.03.2024	7	Test and optimalization	-48	-48
03.03.2024	30.03.2024	28	Further development	-58	-58
31.03.2024	06.04.2024	7	Test and optimalization	-68	-10
07.04.2024	10.04.2024	4	Comparing of versions	-78	-78
15.04.2024	28.04.2024	14	Finalizing implementation	-88	-10
18.03.2024	21.05.2024	65	Writing	-38	50
			Sett inn nye rader over denne		

Milepæler

Dato	Etikett	Posisjon
08.01.2024	Start, jan 8	30
02.03.2024	First implementation feb 4	25
06.04.2024	Second implementation apr 6	20
21.05.2024	Finished mai 21	15
	Sett inn nye rader over denne	

C. Project Handbook

This appendix has been excluded from the report on the grounds of sensitive information. It can be found in the .zip folder.

D. System Documentation

This appendix has been excluded from the report on the grounds of sensitive information. It can be found in the .zip folder.

NTNU
Norwegian University of
Science and Technology