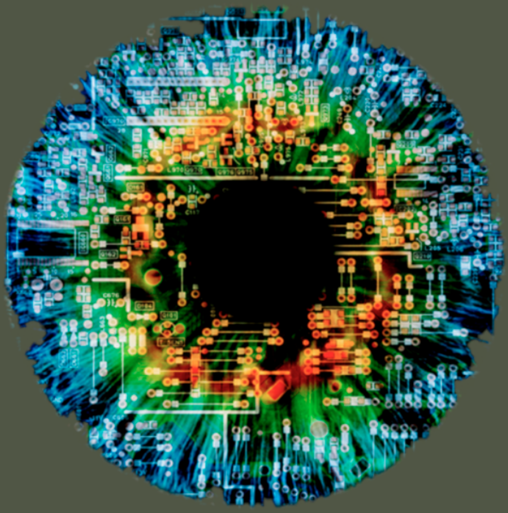


Acceleration of machine learning through an FPGA

A way to obtain minimal latency

Master Thesis

by Amira Moussa

May 31, 2021

Advisors

Asst. Professor Kenneth Skovhede

University of Copenhagen

Niels Bohr Institute

Acknowledgements

I would like to express my gratitude to the *eScience* group, that I was allowed to be a part of. Also I am very grateful for Asst. Prof *Kenneth Skovhede* providing an open environment where I felt heard and supported. Further I want to thank the members of the SME group who were always open to answer all of my questions, give me their feedback and inspire me during my time in the group.

Especially, I want to thank *Carl-Johannes Johnsen* for the strong support during the work of my thesis. Without his insight, help and encouragement, the goal of this project would not have been fulfilled.

Finally, I want to *thank Nam Tran* for being a big support throughout my studies and all of my friends who have read this thesis and made suggestions for improvements. Additionally, thanks to *Niklas Heim* and *Fabian Dornhecker* for letting me use this visually appealing template.

Abstract

This thesis presents the design of an application specific hardware for machine learning which can be used in various physics applications, such as high energy physics and quantum optics. Even though we are better at optimizing and have more computational power was previously possible, there is also a continuous need to make simulations even faster, more reliable, and cheaper to run. We are specifically investigating a FeedForward Neural Network that is used to interpret market data feeds and hence enable minimal round-trip times for executing electronic stock trades. This is because there are similar trades such as hard time restriction, which is also the case for computational physics.
In this thesis the network is optimized to achieve the lowest possible latency. For this purpose, we use Synchronous Message Exchange Synchronous Message Exchange (SME) which is suitable for describing hardware and enables the flexibility to support a wide range of applied trading protocols. This demonstrates how to construct the components of a FeedForward machine learning script down to a processor as SME processes, and how to connect them by using SME busses. The complete system has been implemented in C# and evaluated on an Field Programmable Gate Array (FPGA). The results are promising compared to the Python implementation of the model. We present a proof of concept of an initial solution and its performance provides results that make us believe that a full Neural Network implementation would be feasible and competitive. The final result is a successful implementation of a FeedForward Neural Network model on a FPGA, which runs 21 times faster then the same algorithm on a CPU.

Notation

In this project we highlight processes from the SME program with `this font`.

ASIC	Application-specific integrated circuit. 4
BRAM	Block ram. 47, 48
CLB	configurable logic blocks. 4
CPU	Central processing unit. 3
CSV	Comma-Separated Values. 12
Fintech	Financial technology. 2
FNN	Feedforward Neural Network. 1, 17, 18, 21, 36
FPGA	Field Programmable Gate Array. iii, 2, 4, 8, 12
GD	Gradient Descent. 24
GPU	Graphic Processing Unit. 3
HFT	High frequency trading. 1–3
PReLU	Parametic Rectified Linear Units. 19
ReLU	Rectified Linear Units. 19, 20
SME	Synchronous Message Exchange. iii, 8
VHDL	VHSIC Hardware Description Language. 12, 46

Contents

Chapter 1
Introduction

This thesis will investigate machine learning methods and examine or propose techniques for implementing them effectively in hardware. To validate and drive the design decisions, we will use a sample of applications from High Frequency Trading High frequency trading (HFT), which has strict latency requirements. To avoid complexities in an already complicated implementation problem, we use a simple real-world feed-forward network for stock-price prediction obtained from an industry collaboration with a company wishing to remain anonymous and thus will not be mentioned.

FPGAs are used in various physics applications, such as high energy physics and quantum optics, because of their versatility, programmability, high bandwidth communication interfaces and signal processing capabilities.

Due to the complexity of programming these, their use has been limited to somewhat simple applications. However, many experiments could benefit from having a machine learning model on the FPGA. This is currently done in the top-tier facilities, such as CERN. In this thesis we explore the implementation of machine learning models on FPGAs using SME to accelerate inference.

In this Chapter we introduce the motivation behind optimising HFT alongside descriptions of commonly used processor architectures. Because the FPGA chips are not instruction based hardware and have to be built up on hardware level, there is a need for a tool to translate the algorithms down to the hardware. In this case Synchronous Message Exchange (SME) which will be introduced in Chapter 2. We want to implement a Feedforward Neural Network (FNN) used in high frequency trading to see if we can optimize the speed and efficiency of this model. The FNN model will be introduced in Chapter **??**. The implementation of the FNN from Python to SME and then down to the FPGA will be described in 4. Lastly, in Chapter 5 the testing of the implementation and results will be discussed.

1.1 High Frequency Trading

High Frequency Trading HFT is an automated trading platform that large investment banks, hedge funds, and institutional investors use to automate trading. It consists of algorithms that runs through powerful computers to transact a large number of orders at extremely high speeds. Over the last couple of millenniums, new technologies have constantly been developed and with the evolution of computers, financial markets also evolved. Manual labour is increasingly being automated through the use of algorithmic trading strategies [Gianluca 2017]. One example of algorithm strategic trading is liquidity-providing strategies, where high frequency traders try to earn the bid-ask spread which represents the difference of what buyers are willing to pay and sellers are willing to accept for trading stock. "High volatility and large bid-ask spreads can be turned into profits for the high frequency trader while in return he provides liquidity to the market and lowers the bid-ask spread for other participants, adopting the role of a market maker" [Leber, Geib, and Litz 2011].
These types of problems are studied in deep learning programming. Moreover, these algorithms can work at multiple time scales, but the most interesting one is High-Frequency, which are trades that execute in milliseconds or less. At this time-scale, machines are needed because of their speed and can process more information with greater time efficiency.

1.2 Motivation

The stock market is moving towards HFT and micro optimization which has received a lot of attention during the past couple of years [Agarwal 2012], turning it into an increasingly important component of financial markets.
The demand for storing big data and running algorithms faster has reached its limit with the commonly used hardware, mainly the CPU and GPU. Companies interested in optimizing their trading have grown an interest for the FPGA. These have very specific technical characteristics that enable them to execute certain types of trading algorithms much faster than traditional software solutions.
HFT algorithms compete with each other on two dimensions: Firstly, they receive large amounts of stock market pricing data every microsecond. Secondly, they must therefore be able to act extremely fast on the received data, as the profitability of the signals they are observing results in latency.
The first point is crucial. High-frequency trading is all about speed and minimizing latency. This is due to the fact that the faster you can run trading strategies and algorithms for analyzing minute price changes and executing trade orders, the higher the probability to gain most profit. Secondly, keeping a large amount of data in memory will slow down the hardware. Therefore, it is important that algorithms use only a minimal amount of data and parameters, which can be stored in fast accessible memory.
Artificial neural networks have become a necessary tool in almost any area handling data and help predict future ranges from financial technology Financial technology

(Fintech) to physics because of its abilities to predict different outcomes from complex relationships between data. Nevertheless due to large incoming data and the need for faster inference, the interest in looking into other alternatives on implementing algorithms has grown. The FPGA has shown great improvement in both power consumption and performance on some specific tasks when compared to the GPU and could therefore be a potential candidate as an alternative to the other hardware [Véstias and Neto 2014]. However, the FPGA is not ideal for replacing all processors because of their customization cost. Nonetheless, due to its parallelism it could help optimize the latency of memory sharing, which would mean a lot in the HFT field. The latency of memory sharing is vital as the algorithms developed by trading firms are competing on the scale of nanoseconds. To investigate these trade-offs, this thesis will investigate how the implementation of a Feed forward network on an FPGA could potentially enhance the performance and whether it is worth replacing parts of the traditional hardware solutions with an FPGA.

1.3 The different hardware

Due to the increasing popularity of using ML in fin-tech, there has been a race to use the fastest hardware to achieve rapid predictions and trading [Cong et al. 2018]. Thus, there is a growing demand for hardware platforms able to compute such intensive machine learning algorithms. As deep learning has driven most of the advanced machine learning applications, it is regarded as the main comparison point. Nowadays, there are several types of processors, although when discussing heavy calculations, the main ones are the Central Processing Units (CPU), Graphics Processing Unit (GPU), Application Specific Integrated Circuit(ASIC) and Field Programmable Gate Array (FPGA). To provide an understanding of the main applications and differences underlying each processor, the following section will present a brief overview of the different hardware and how they compare.

1.3.1 Central Processing Unit

Central processing unit (CPU), is chip that executes a program based on a specified set of instructions in a sequential manner. These are optimal for single process systems, where the code needs to be executed in a sequential or linear manner. The internal hardware structure is defined by the CPU vendor and cannot be modified [Asano, Maruyama, and Yamaguchi 2009]. As a result, CPUs are general purpose and can thus perform any function based on the software program that is uploaded onto them.

1.3.2 Graphic Processing Unit

Graphic Processing Unit (GPU), is a chip that performs fast mathematical calculations, primarily for the purpose of graphics. In the 1980s, they were only used to offload graphics from the CPU [Jones et al. 2010]. As we progressed, graphics became more advanced, which triggered a concurrent evolution of advanced GPUs. An image is composed of thousands of pixels which are processed by hundreds to thousands of identical cores that are specifically designed to execute the same program in a parallel manner. Because of their extremely efficient parallel functioning, GPUs are now used in a variety of different fields and applications such as vector and matrix mathemathics for which they render jobs faster than the CPU [Jones et al. 2010]. The benefit is that they are instruction based hardware that can be bought as a finished product. This means that you can write instructions for them in different high level languages, which makes is much easier for software engineers to work with.

1.3.3 Application-Specific Integrated Circuit

Application-Specific Integrated Circuit Application-specific integrated circuit (ASIC), are single purpose chips, meaning that they can only execute commands for which they have been specifically designed. Hence, they cannot perform another function or execute different applications once programming is complete [Brogioli 2012]. The logic function of ASIC uses hardware description languages such as Verilog or VHDL. However CPU, GPU and FPGAs are all types of ASIC, their applications being general purpose central unit, graphics oriented or field programmable, respectively.

Since these chips are built for a singular purpose they do not have to run through unnecessary circuits to run a function, allowing the power consumption of ASICs to be very minutely controlled and optimized. As a result, they are often more power efficient compared to other alternatives.

1.3.4 Field Programmable Gate Array

FPGA, is a chip optimized for being re-configurable hardware, as in the user 'programs' the hardware circuit. A basic FPGA, consists of small chunks of configurable logic blocks configurable logic blocks (CLB) which can be configured to perform different functions. It consist of various components such as transistor pairs, look-up tables (LUTs), flip flops (registers), and multiplexers. The blocks are connected to each other with electronic wiring that can be turned on and off [Brogioli 2012]. The logic blocks can be thought as separate modules which can operate in parallel. These can be controlled and it is possible to hook these together by programming the interconnects in order to build something meaningful. An FPGA can be reconfigured by programming the logic blocks and manipulating the internal state. This can also be done even after deployment, which makes them ideal for systems and devices that need frequent updates such as prototypes, networking products and other electronic systems [Véstias and Neto 2014]. Writing instructions to the FPGA can mainly be done in low level languages such as VHDL or Verilog.

1.3.5 Comparison

The FPGA has a different architecture from the GPU and CPU, although in some cases it is possible to apply either one for similar tasks. FPGAs tend to have more flexible architectures as compared to GPUs and CPUs. The main difference is that the CPU/GPU are instruction based hardware, which means that one can access the hardware using higher level language, where the FPGA needs to be accessed from lower level languages such as VHDL or Verilog. This difference can make FPGA more responsive and allows more dedicated special-purpose implementation desired by the developer, described in [Lockwood et al. 2012]. CPUs/GPUs, comparatively, are easier to use for software engineers than FPGAs as the development process for the latter tends to be much more knowledge extensive and complicated than for the former, which explains why modern GPUs are being applied across a multitude of fields [Jones et al. 2010].
One of the prime features and advantages of FPGAs is that the entire internal hardware can be reprogrammed and reconfigured as the user is permitted to determine the logic of each block in the system. Hence, they are much more flexible in their programming and can be customized according to the needs of the programmer.
FPGA are in comparison to ASIC reconfigurable whereas the latter is a permanent circuit. The FPGA has a simpler design flow due to the chips flexibility, whereas the ASICs entail more complex design flows, given their reliance on a permanent architecture. In fact, the ASIC design requires dedicated and more expensive tools. The FPGA is a low power consuming chip compared to the CPU/GPU. It is more power intensive in comparison to ASIC, which is a known established solution for battery operated products. In the following Chapter, the software to deploy code on the FPGA will be introduced.

Chapter 2
Synchronous Message Exchange

Synchronous Message Exchange (SME) [Johnsen, Thegler, Skovhede, et al. 2021b] is a programming model used to develop highly concurrent systems. In order to translate the FeedForward neural network to the FPGA we will use this tool to make it possible. The ideal goal is to program on a high abstraction level using e.g Python or C# without knowing and working too much with low level programming language such as Verilog or VHDL.
The purpose of this Chapter is to provide an introduction to SME and give the reader a better understanding of the logic and the design ideas applied in this thesis. Notably, the purpose is not to go into deep theoretical and technical detail, but rather provide the necessary background knowledge to understand the proceeding discussions. The structure of SME is also introduced, for a more detailed description of the SME structure see Chapter 4. A brief guide is attached in appendix A.1 with an emphasis on a Sigmoid function that is used in the project.

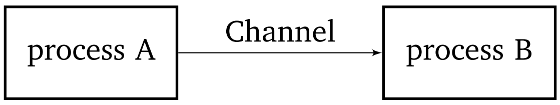


Figure 2.1: *Communicating sequential processes from A to B*

2.1 Concurrent systems

As introduced in Chapter 1 the special characteristic of the FPGA is the fact that it can handle concurrent computing. Concurrent computing is a form of computing providing a way to make effective use of parallel and distributed systems that perform many simultaneous tasks using a multiprocessor [Lönnberg and Berglund 2007]. The benefit of a multiprocessor is the ability to conduct several tasks simultaneously which enhances the speed and efficiency. The efficiency is determined by the speed compared to the resources used in designing and implementing the multiprocessor, which for this thesis will be the FPGA. However, working with concurrent systems also means concurrent problems, such as sharing memory sharing is one of them. It can create a non-deterministic problem of reading and writing the same memory, which often results in unexpected behavior. An example could be the act of printing out an aligned set of numbers to your console using more than one thread, which is an independent set of the values in a process. If the code is executed multiple times, the order of numbers would vary between the runs. This is due to the time dependence in the threads. Consequently, time is an essential factor when working with concurrent systems.

2.1.1 Communicating Sequential Processes

Various attempts have been made to solve this problem. Communicating Sequential Processes (CSP) [Lamport and Schneider 1984] was one, first proposed In 1978, *Communicating Sequential Processes* [Hoare 1978] to solve exactly these issues and with it, CSP was created. CSP was introduced as a model to describe patterns in concurrent systems and communication between sequential processes running in parallel. In current time, CSP is a formal method for modelling concurrent systems described by a process of algebra. CSP is built on two simple ideas, 'processes' and 'channels'. A process is an ordered sequence of operations. These processes do not share any memory, therefore one process cannot access a specific value in another process, which would solve the memory problem. The other is channels, where the processes communicate with each other by passing information. A simple example thereof is depicted in Figure A.1, which illustrates process **A** passing a value onto a channel, which process **B** takes as an input. Once the value is passed through the channels, process **A** will lose access to it. This creates a messaging framework that enforces strict synchrony between communicating processes. However, the asynchronous nature of CSP became a problem for hardware models. This laid the foundation for the establishment of Synchronous Message Exchange SME.

SME was first introduced in 2014 and after several iterations has evolved to a programming model, a simulation library, and VHDL code generators [Vinter and Skovhede

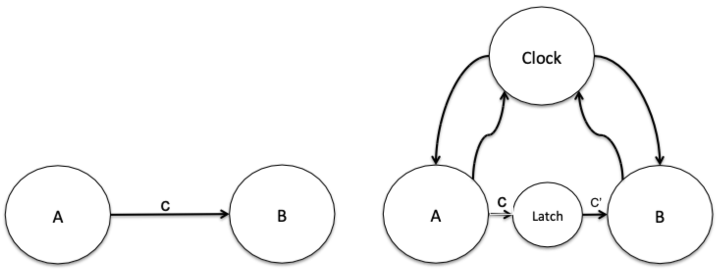


Figure 2.2: *Enforcing global synchrony on a simple CSP model, where the channel **C** transfer information from the process **A** to process **B**, resulting in an increasing complexity. Figure from [Vinter and Skovhede 2015]*

2015]. The original idea was conceived following an attempt to create a hardware implementation of a vector processor, modeled in PyCSP [Lamport and Schneider 1984], a CSP library for Python. The work was initially presented in the paper *BPU Simulator*[Rehr, Skovhede, and Vinter 2013], which introduced a high abstraction level simulation. The subject was explored more in detail in the master's thesis project [Skaarup and Frisch 2014] where two students implemented a vector processor using PyCSP. The results of this master's thesis made it clear that PyCSP could be used to model hardware: Nevertheless, this lead to the discovery of different challenges and CSP was therefore not sufficient. Each process would have to read the clock signal in, to comply with the clock. In order to avoid race conditions, the system had to be implemented with a two-way clock. This meant that the need to enforce global synchrony to the circuit resulted in an outburst. Even simple circuits became overwhelmingly large as shown in Figure 2.2. It is clear how trying to use PyCSP for modelling synchronous hardware would result in extremely large and complex networks, which is not an ideal way to write hardware models. This was because of the number of channels, for controlling the progress and for simulating the clock, became enormous. The conclusion was that PyCSP alone was not a viable tool for describing timed hardware, since it is forcing a globally synchronous environment onto the CSP model. A potential solution could be to isolate the processes and connect via channels, which has shown to be the right approach for building larger hardware models [Vinter and Skovhede 2015].

2.2 Synchronous Message Exchange

Most of this theory is based on [Vinter and Skovhede 2015], [Johnsen, Thegler, Vinter, et al. 2020]. The motivation behind SME rose from the need to provide a simple framework for programming an FPGA. It is an environment for developing and testing hardware designs for FPGAs in C#. With SME it is possible to create hardware structures that can be translated to VHDL. However, FPGAs can be a better choice when it comes to energy sensitive applications, since FPGAs can, in some cases, achieve the same performance as a GPU but with lower energy consumption [Johnsen, Thegler, Vinter, et al. 2020]. Previously, the developer needed to design an integrated circuit on the gate level for the FPGA. This could be difficult due to design framework in the low level languages, which is not a common knowledge amongst software developers. Some high-level methods for programming an FPGA have been developed, however, these are often tedious to work with. Thus, designing and implementing hardware models is beneficial with SME.

The goal with SME is to give software developers a tool which provides the opportunity to program hardware, but with an added abstraction layer which separates the developer from the hardware details, such that, the development resembles the structures and semantics that is known from software development.
The idea is to develop individual processes, test them through simulations and then connect them together to form larger hardware models. By leveraging the features of a modern *C#* Integrated Development Environment, such as Visual Studio, it becomes much faster to develop, experiment and test FPGA designs, especially for a software developer. This is due to the fact that SME adds a software abstraction layer that conceals the complexity that would normally require high level FPGA expertise. SME was built on the requirements of a **Hidden clock** , **Global synchronization**, **Broadcasting channels (Busses)**, **Shared nothing** and **Implicit latches** .

2.2.1 *The hidden clock and global synchronization*

When writing in hardware, many factors have to be considered, such as timing. Since processes could read and write signals at any time there is no way of knowing if the following processes would use old or new data. Therefore, some kind of predictability for the hardware is required, for which reason global synchronization is needed. The hidden clock is made to establish coordination between processes by synchronizing them all. A SME model consists of clock cycles and one hidden clock that is then propagated out to all the processes having a clock cycle and activates them. A clock cycle is the period between each signal, which is divided by all processes. An SME clock cycle consists of three phases: reading, computing and writing. The visual explanation could be shown as a step function going from high to low, see Figure 2.3, The process is activated on the rising clock when the process is executed where it reads from the bus and then it computes and writes to the bus, all in one clock cycle. Just before the rising edge of the clock, all signals are propagated on all busses which means that all communication happens simultaneously. Un-clocked processes will first be activated when the input has been written to by other processes.
The SME model supports both synchronous and asynchronous processes, whereby a synchronous process is run during every clock cycle, while an asynchronous process is only run when receiving all of the signals on its input buses.

2.2.2 *Broadcasting and busses*

In CSP, processes communicate with each other using channels that make use of the rendezvous protocol, meaning that data gets transmitted through the channel once the destination says it is ready to receive and vice versa. On the contrary, SME uses

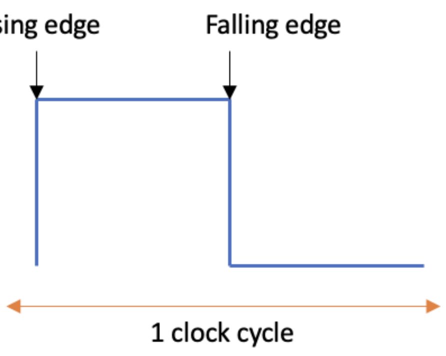


Figure 2.3: *Illustration of a clock cycle*

broadcasting channels called busses, that any process can read from. This means that a process can broadcast its output to multiple processes through a single bus. Using CSP, multiple channels would have to be used to emulate a single bus in SME. The busses define and manage the data that is exchanged between processes. These can be visualised as a pipe or a channel. When data is written to a bus it will be available in the following clock cycle if it is a clocked process and available right away for un-clocked processes. Furthermore, a bus is able to contain multiple data types and values that can be accessed by a process.

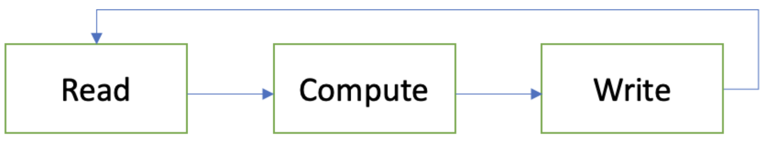


Figure 2.4: *SME process for one clock cycle.*

2.2.3 Shared nothing

The fastest way to handle data through hardware is by handling memory separately. A shared-nothing architecture (SN) is a distributed-computing architecture in which you have a number of separate nodes that do not share memory or storage. Operating under numerous self-sufficient nodes rather than having a single source of particular resources offers several advantages: easier scaling, non-disruptive upgrades, elimination of a single point of failure, and self-healing capabilities. SN eliminates single points of failure, allowing the overall system to continue operating despite failures in individual nodes and allowing individual nodes to upgrade without a system-wide shutdown [Stonebraker 1986].

2.3 Hardware Description Language

Generally, implementing code down to FPGA is programmed using Very High Speed Integrated Circuit HDL VHSIC Hardware Description Language (VHDL). This is harder to use since it is intended primarily to be a parallel programming language. This can be a very difficult task [Johnsen, Thegler, Vinter, et al. 2020], especially for a software engineer wanting to implement on a FPGA. The vendors are not implementing recent updates to VHDL and so it is not evolving with time or the programmer.

The SME framework is implemented in the .NET framework, which is the version of SME that will be used throughout this thesis. With the C# SME library, it is possible to write all control logic in C#. All data that is written to a bus, can be logged for each clock cycle and saved as a test bench, which is a piece of software used for testing hardware models. It provides input data to the hardware model and verifies its output which is saved as a Comma-Separated Values (CSV). Hence, it can then be used for compiling the program into VHDL, which can be put onto FPGA circuits. This eliminates the need to write a separate VHDL test bench which improves developer productivity immensely. This is needed since some of the implementation still is done manually software for implementing on hardware, which in this case will be Vivado. A further explanation on the implementation can be found in Chapter 5.

2.4 SME setup and structure

To show the basic structure of how SME works, we will go over the fundamental structure. For further explanation on the how the structure was set up for the whole project, see Chapter 4. When analyzing the general structure of SME programs, there are three structures, which consists of the following:

1. **Connection**: How the circuit is connected, i.e which busses connect to which processes.
2. **Processes**: The process structure for each function and how they behave.
3. **Verification data**: All data from the Python FNN that could be useful for the verification of the hardware model.

2.4.1 Connection

The fundamental structure behind the SME network is the communication between the processes through busses. Understanding different communication structures in SME will provide the insight necessary for designing the translated structures of the Python model. A network in an SME program is a crucial part that connects all processes together with communication. Defining the network from process instances also has the advantage that one process can be instantiated with different parameters several times within the same network, providing the possibility of reusing the processes for different purposes.

```
        public interface ValueTransfer : IBus {
                double value { get; set; }
        }

```

Listing 1: *Simple bus that transfers one value*

Busses

As previously explained, an SME bus defines a collection of channels which are used for all communication between the processes. Each channel has a type describing the communicated data and can be initialized with an initial value. The IBus interface marks an interface as a bus of a read or write in SME, which is shown as an example in Listing 1 where we define a *bus* to transfer a value. An SME bus does have an identifier which is used for referencing the bus. All channels within a bus are connected to the process at the same time, and it is up to the developer to call the correct channel within the bus for either a read or a write.

2.4.2 Process structure

The processes in an SME program describes the basic behaviour of a model. An SME process is defined by the *SimpleProcess* class and consists of an identifier, process parameters, bus, and variable declarations. The body of the process, the process statements, consists of sequential statements such as communications and calculations that are to be evaluated once for each clock cycle.
A process is initiated in the network of an SME program. A process can be instantiated with a set of parameters. These parameters can be a mix of input and output busses and constants.
When defining a process, we can give either an **Input Bus** or a **Output Bus**. The name of the input bus can be given as a process parameter, which the process can use to read from the actual bus channel, as can be seen in App. A.1 Listing 15. In this example, the processes reads the data from *m_input* in the bus and writes the data to the *m_output*.

2.4.3 Verification data

It will always be necessary to generate input for pure SME networks, which can be done in multiple ways. One way of initialising the data in the SME network is by give the process with a constant given as a parameter, or by hard-coding internal values into the process. Another way is to have a separate process that generates data for the network. The first attempt of writing a process to generate data shown in example App. A.1 Listing 14. Here the process clock is a data generator process. It does not read data from any input bus, thus, it can only generate data to write to the network. The example shows the `Sigsimulator` which generates values from $[1, 10]$ and writes it out onto the output bus. A further explanation of the structure of a process will be

```
    input = File.ReadAllLines(CSVfile)
        .Where(x => !string.IsNullOrWhiteSpace(x))
        .Select(x => double.Parse(x.Trim(), CultureInfo.InvariantCulture))
        .ToArray();
}

public override async System.Threading.Tasks.Task Run()
{
    while (true)
    {
        await ClockAsync();
        output.Enabled = index.Ready;
        if (index.Ready)
        {
            output.Address = index.Addr;
            output.Data = input[index.Addr];
        }
    }
}
```

Listing 2: *C# code to import an input CSV file, using SME processes to save the data output as a flat array*

introduced further in Chapter 4. Another way to generate data is to make a process that imports the data and reads them in through a bus. This was made as a simple SME process structure, where the function, reads the data from a CSV file and saves it as a flat array. This is shown in Listing 8.
An SME process that does not read any input is just a data generation process, but in this case we use the input bus *index* to make sure that each value that gets read from the CSV file gets and index. The output-bus transfers the array of data and makes sure it gets written out. In line 24 the *async process* only runs when the index is ready. The address will get saved of each value and write them out in an array.

2.5 Related work

As mentioned in Chapter 1, FPGAs contain many interesting properties which makes them more flexible to use, faster and more energy efficient than GPU/CPUs for certain tasks. Algorithms are getting more complex and energy consuming. The use of FPGAs is not widespread, due to the low level language used to program the chip [Fang et al. 2020]. This has started a smaller movement in building alternative implementation frames such as SME. An interesting framework constructed from the same idea is **hls4ml**, which was made as a joint project between Xilinx and CERN for accelerating the inference of processing data [hls4ml Contributors 2021]. A growing team of physicists and engineers from CERN wanted to have a flexible way to optimize custom event filters in the Compact Muon Solenoid (CMS) detector they are working with at CERN. The very high data rates in the CMS detector required event processing in real-time, but trigger filter algorithm development hindered the team's ability to make progress [Fahim et al. 2021]. HLS4ML is built on the idea of being a user-friendly software, based on High-Level Synthesis (HLS), designed to deploy network architectures on FPGAs. This is done in *C* as a sequential programming model designed for CPUs, rather than FPGAs. As such, it is not uncomplicated to gain performance, as it relies on automatic derivation of parallelism. Among others to describe hardware using higher level language are **Chisel-lang** [University of California 2012], which is similar to SME, but described as a library in Scala. This means that the programmer needs to be able to keep two states of the program in mind, one that is running in Scala, evaluating the some specific condition statements, and another running the generated code, not being able to see the condition statements [Johnsen, Thegler, Vinter, et al. 2020]. **OpenCL** is a popular framework that intent to get more GPU programmers on board.[Burns et al. 2019] However the focus here is not to accelerate with FPGAs and as introduced in cahpter 1, the different hardware has different benefits. and **QuokkaEvaluation** [Muryshkin Evgeny 2018] is still an ongoing project focusing on specific FPGA implementations and has similar elements from SME, but is still in the initial phase.

Chapter 3
FeedForward Neural Networks

Neural Networks are a set of models, inspired by the human brain, that can be used to recognize patterns by finding regularities and similarities in data using machine learning data [Yun, Huyen, and Lu 2018]. The word Neural originates in the McCulloch-Pitts neuron [Mcculloch and Pitts 1943], a simplified model of the human neuron as a kind of computing element that could be described in terms of propositional logic. Just like the brain, Neural Networks are constructed from small neurons or perceptrons that are connected to each other with weights. These are made to process and pass on incoming data. The patterns they recognize are numerical, contained in vectors, into which all real-world data, be it images, sound, text or time series, must be translated first [Goodfellow, Bengio, and Courville 2016].

In this Chapter we introduce the basic principles behind a Neural Network. This will be followed up by the architecture of a FeedForward Neural Network FNN where the computation proceeds iteratively from one layer of units to the next. This will come to handy for the implementation.
This section's general introduction to the architecture of FeedForward Neural Networks is mainly based on the books [Jurafsky and Martin 2009] and [Goodfellow, Bengio, and Courville 2016].

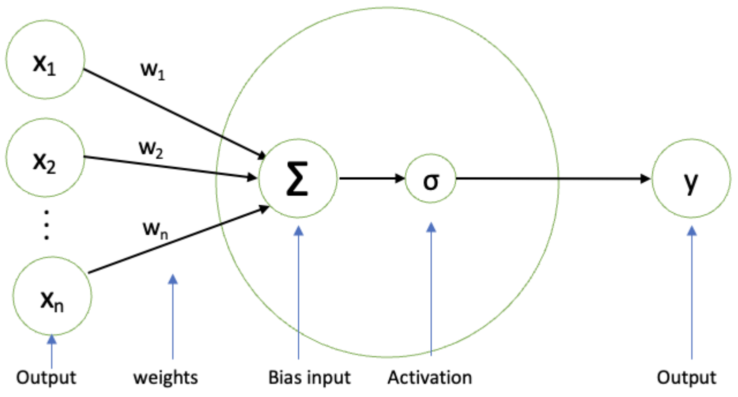


Figure 3.1: *Single neuron model where the neurons x_i takes the n weighted inputs plus a bias. This is an example for a linear case given to the activation function which then outputs an y*

3.1 Artificial Neural Networks

Machine learning is a form of applied statistics with the focus on using computers to estimate statistically complicated functions. By having a data set, the goal is to find a function that fits a data-set best. One class of modelling is Neural Networks which have been shown to work well. One way to use Neural Network is applying FeedForward Neural Network FNN which will be explained later.
But first consider the mathematical model shown in Figure 3.1. This is the simplest example and the building block of a Neural Network is a single computational unit or a perceptron. A unit takes a set of real valued numbers as input, performs some computation on them, and produces an output.
More specifically, a Neural unit is taking its inputs x_i and their weighted sum w_i which determines the contribution of a given input to the output, with one additional term in the sum called a bias term b. The weighted sum z can be represented as:

$$z = \sum_{i}^{n} w_i x_i + b \tag{3.1}$$

The sum over all the inputs multiplied by their weights can conveniently be represented by a dot product.

$$z = w \cdot x + b \tag{3.2}$$

3.1.1 *Activation functions*

Instead of using a linear function z as the output, Neural units apply a non-linear function f to z. This is called an activation function. The *feed forward Neural Networks* (FNN) lets information flow through the function being evaluated from x, through the activation function σ, and finally to the output, y. Figure 3.1 shows a schematic neuron where the unit goes into the activation function and gives the output y. The value y is given by:

$$y = \sigma(z) = \sigma(w \cdot x + b) \tag{3.3}$$

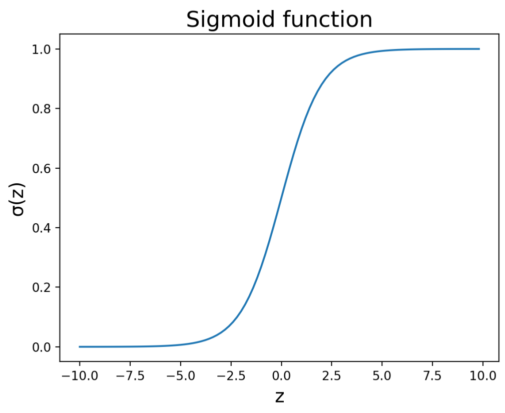


Figure 3.2: *Sigmoid Function. Frequently used as an activation function in FNN The sigmoid function takes a real value and maps it to the range [0,1]*

the bias b is a measure of how easy it is to activate the neuron. Generally, there are a great variety of functions. For the purpose of this paper, we will only discuss the three non-linear functions applied here - the Sigmoid, Rectified Linear Units (ReLU), Parametic Rectified Linear Units (PReLU) and Softplus.

Sigmoid: The sigmoid function, which is shown in Figure 3.2, is defined as

$$\sigma_{sig}(z) = \frac{1}{1+e^{-z}}, \tag{3.4}$$

takes the real valued number and maps it to a value into the range $[0, 1]$ which is useful because the outliers get squashed toward 0 or 1.
substituting Eq.3.3 into Eq.3.4 gives the output

$$y = \sigma_{sig}(w \cdot x + b) = \frac{1}{1 + exp(-(w \cdot x + b))}, \tag{3.5}$$

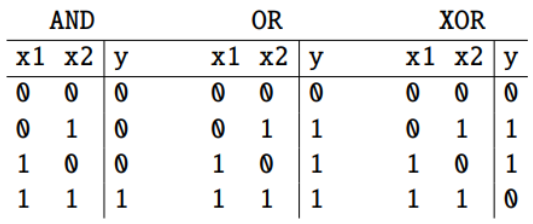
ReLU: Rectified Linear Unit is the most commonly used activation function in deep learning models. The function returns 0 if it receives any negative input, but for any positive value x it returns that value back. So it can be written as

$$y = \sigma_{PReLU} = \begin{cases} x & \text{for x} > 0, \\ 0.01x & otherwise \end{cases} \tag{3.6}$$

PReLU: Parametric PReLU (PReLU) makes it a parameter for the Neural Network to figure out itself: $y = \alpha x$ when $x < 0$, where α is a parameter and x is the input. If $x \geq 0$ then $y = x$.

$$y = \sigma_{PReLU} = \begin{cases} x & \text{for x} > 0, \\ \alpha x & otherwise \end{cases} \tag{3.7}$$

PReLU is another popular activation function because it resolves an issue called "the dead neuron problem" which is common with the ReLU function. This problem occurs when inputs approach zero or are negative, which will make the gradient of the



AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

Figure 3.3: *Truth tables of elementary logical functions of two inputs for AND, OR and XOR. The XOR gives the output 1 if the inputs are different from each other and 0 if they are the same*

ReLU function zero [QingJie and WenBin 2017]. This implies that with poor initialization, where the majority of neurons have negative output, these neurons will have no incentive to adjust their weights and hence, the Network will have limited ability to "learn". PRelu addresses this issue, given that it doesn't have zero-slope parts.

SoftPlus: Finally, the softplus function is a smooth approximation to the PReLU activation function, and is sometimes used in the Neural Networks in place of PReLU. It is actually closely related to the sigmoid function.In particular, when $x \rightarrow \breve{}\infty$, the two functions become identical.

$$\sigma_{soft} = \log(1 + \exp x), \tag{3.8}$$

However, these activation functions have different properties, making them useful for different Network architectures. For example the ReLU function has beneficial properties. Networks trained with the rectifier function almost completely avoid the problem of vanishing gradients, as the gradients remain proportional to the node activations [Goodfellow, Bengio, and Courville 2016]. Contrary for the sigmoid function, containing very high values of z result in values of y that are saturated, i.e., extremely close to 1, which causes problems for learning [Jurafsky and Martin 2009]. Rectifiers don't have this problem, since the output of values close to 1 also approaches 1 in a nice gentle linear way.

3.1.2 The XOR problem

When sending data to a unit, it goes through the activation function which reduces the sum of the input values to a 1 or 0 value,or at least a value very close to. If we consider the task of computing elementary logical functions of two inputs, like AND, OR and XOR we get the truth tables, as shown in Figure 3.3
On the surface this appears to be a simple problem, however, as shown by Minsky and Papert in 1969 [Minsky and Papert 1969]this becomes an issue for a Neural Network, that is only based on a single perceptron. The difference is that a perceptron is purely linear and a unit is non-linear. If we look at a two dimensional problem for the different truth-tables in Figure 3.4 we see the the possible logical inputs (00, 01, 10, and 11) and the line drawn by one possible set of parameters for an AND and an OR classifier. If we look at the AND example we see, that when $x1 = 1$ and $x2 = 1$ then

$y = 1$ and in all other cases it is 0, so if you wanted to separate all the ones from the zeros by drawing a single line, you would just draw the line as shown in the graph.

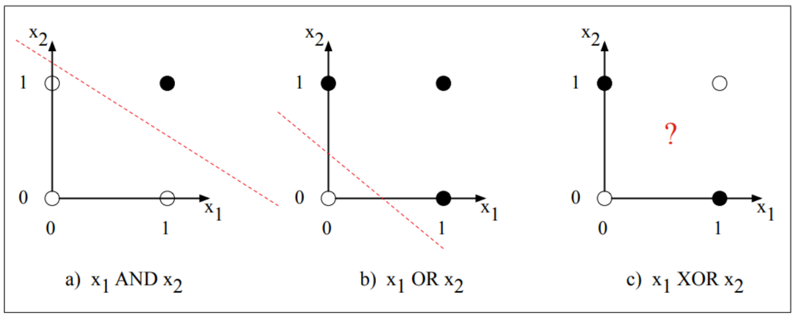


Figure 3.4: *Truth-tables of elementary logical functions of two inputs for AND, OR and XOR [Jurafsky and Martin 2009]*

This class can be separated with a single line. They are known as linearly separable patterns, meaning that classes of patterns with a n-dimensional vector can be separated with a single decision surface.
That is how the perceptron works, it draws a boundary to separate the binary values. When the perceptron is trained, it will have the weights value adjusted to form the line shown. Since the output of perceptron is a linear function, the two classes must be linearly separable in order for the perceptron Network to function correctly.
Let's see what happens with XOR problem that gives $y = 1$ if $x_1 \neq x_2$ and $y = 0$ otherwise. Notice that there is simply no way to draw a line that separates the positive cases of XOR (01 and 10) from the negative cases (00 and 11). Hence, we conclude that XOR is not a linearly separable function. Instead, we will have to draw multiple lines and thus require multiple perceptrons. In n-dimension if you can draw a hyperplane to separate the binary values, then you can use perceptrons to solve that problem.

3.2 FeedForward Neural Network

Having provided the necessary background knowledge behind units/perceptrons and how they work, next we will need to elaborate on the functioning of the **FeedForward Neural Network or FNN**. A FeedForward Network consists of multiple layers in which the units are only connected in one way. FeedForward Networks are often applied to classification tasks, such as the recognition of a certain shape in an image. There are three kind of layers, the input, hidden and output layer. However the characteristic and core of a FNN is the hidden layer, which consist of units described in 3.3. The output of each layer are passed to units in the hidden layer which sums over all the input units until the final layer gives the last output. Figure 3.5 shows an FNN where the input layer represents the data that is fed into the Network, followed by one or more hidden layers, and an output layer.
We can think of a fully-connected Network as a function $F : \mathbb{R}^n \to \mathbb{R}^m$ that maps the input $(x_1, .., x_n)$ to the output $(y_1, .., y_m)$ We are now considering a structure of numerous neuron ordered in different layers. Therefore, F can be split into a chain

of simpler functions $f^{(i)}$, where $f^{(i)}$ correspond to activation/output of the neurons in the i'th layer.

$$F(x) = f^{(out)}(..f^{(i)}(..f^{(1)}(x)..)..), \tag{3.9}$$

Where the first layer after the input layer is represented by $f^{(1)}$ and final layer as $f^{(out)}$. The input layer is usually not counted when enumerating layers, but we will call it the 0'th layer.
In a Neural Network we only know the input and the output from the final layer. The activation $f^{(i)}$ of the layers in between is not shown which is why they are called hidden layers. This is one of the reasons why machine learning is seen as a 'magic black box'. A Network consisting of more than one hidden layer is called *deep Neural Network* We define the model depth as the number of hidden layers in the Network, thus, not counting the input and output layer. It is common for hidden layers to be much larger than the input and output layer in deep Neural Networks, because having larger dimensions give the best prediction [Goodfellow, Bengio, and Courville 2016]. The number of weights for deep Networks becomes therefore approximately proportional to the square of the count of nodes per layer in these large hidden layers.
Instead, we can think of layers as we described the neurons. Now just with a bias b as a vector and the weight as a matrix W, consisting of the combination of the weight vector w_i and bias b_i for each unit i. From a visual point of view as illustrated in Figure 3.5, we can represent each element of matrix W as W_{ji} where it connects the ith input x_i unit to the jth hidden unit h_j. By representing it with a single weight matrix W, the output of the hidden layer h is thus:

$$h = \sigma(Wx + b) \tag{3.10}$$

where σ is applied to a vector elementwise, so $\sigma[z_1, z_2, z_3] = [\sigma z_1, \sigma z_2, \sigma z_3]$.
The dimensionalities of these vectors and matrices are enumerated as $n_0, n_1 and n_2$ for respectively the input, hidden and output layer. Here $x \in \mathbb{R}^{n_0}$, $h \in \mathbb{R}^{n_1}$ and $b \in \mathbb{R}^{n_1}$, since each hidden layer can take a different bias. The weight matrix therefor has dimensionality $W \in \mathbb{R}^{n_1 x n_0}$.
The matrix multiplication from equation 3.10 is thus:

$$\begin{aligned} h &= \sigma\left(\sum_{i=1}^{n_0} W_{ji} * x_i + b_j\right) \\ &= f^{(i)} = \sigma(W^{(i)} * f^{(i-1)} + b^{(i)}) \end{aligned} \tag{3.11}$$

where $f^{(i-1)}$ is the output from the $(i-1)^{th}$ layer. Like the hidden layer, the output layer also has a weight matrix $\mathbf{U}$ and sometimes a bias, but we will look at a simple example without the bias for the simplicity sake. The weight matrix for the output layer is multiplied by its input vector from the layer before which is the hidden layer h to produce the intermediate output z.
The output of the hidden layer h is thus:

$$z = \mathbf{U}h \tag{3.12}$$

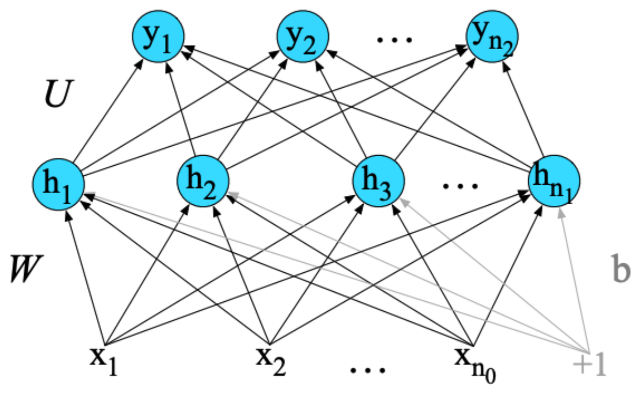


Figure 3.5: *A simple FeedForward Network, with one input layer (not counted as layer), hidden layer and one output layer [Jurafsky and Martin 2009].*

Since z is a vector of real-valued numbers and needs a classification of probabilities, we can use normalization function such as the softmax to get a probability distribution. The output of the hidden layer h is thus:

$$y = \text{softmax}(z) \tag{3.13}$$

In this way the layers in FNNs are able to model not only arbitrary functions but also separate data-sets that are not linearly separable. This linearity is only broken by the activation function which are contained in the hidden layers, which act as a non-linear transform that distorts the input. It is done in such a way that its classes become linearly separable by the output layer as explained in the 3.1.2.

3.2.1 Universal Approximation Theorem

One of the most striking properties about neural networks is that they can compute any function at all and can be approximated by any Borel measurable function [Hornik 1991]. A simpler explanation is that the Universal Approximation Theorem states that a Neural Network with one hidden layer can approximate any continuous function for inputs within a finite-dimensional space to another with any desired non-zero amount of error, provided that the Network is given enough hidden units. However it does not state how large this Network needs to be to solve the given problem, so this is still done by trial and error methods.

3.3 Training of a Neural Network

A FeedFoward neural networks are mostly used for supervised learning where the data to be learned is not sequential nor time-dependent. This implies that we know the correct output y for each observation x. What the system produces, is an estimate of $\hat{y}$ (the true y). The goal of the machine learning approach is to find a weight and bias configuration for each layer that captures the *essence* of the presented data set, that make the estimate of y for each training observation as close as possible to $\hat{y}$. Hereby, the Network is very much dependent on the training data-set. It should

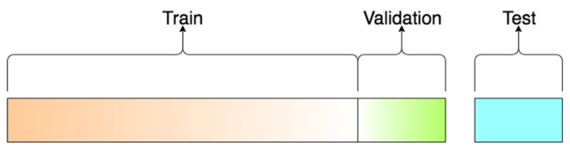


Figure 3.6: *All data used for training a ML model, split up in a training, validation, and test set*

ideally include the all kind of combinations of possible inputs. This can be difficult to do in practice and the available data-sets are therefore typically split randomly into three categories: *training*, *validation*, and *test* set.

The **training set** is the actual data set that we use to train the model (weights and biases in the case of a Neural Network). The model sees and learns from this data. To do that we will need a loss function that models the distance between the system output and the expected output. The commonly used technique to train the model is by some variation of Gradient Descent Gradient Descent (GD). GD algorithms try to minimize a certain loss or cost function with respect to a given weight configuration. Since we will not train in this thesis, we will not give a further explanation. The **validation set** is used to provide an unbiased evaluation if a model fits on the training data-set. This tells how to change the weights and biases overall behaviour of the Network, also known as tuning hyper parameters. The evaluation becomes more biased as skill on the validation data-set is incorporated into the structure of the model. To reduce the risk of over fitting, the optimization should be stopped as soon as the error on the validation set is not decreasing any more. Generally, over-fitting occurs when a model learns the training data set too well, performing smoothly on the training data set but can not seem to work on a different sample [Kriesel 2007] This is one of other methods applied in machine learning in order to achieve a generalization over previously unseen data points. Because the information of the validation set is leaking into the Network via the early stopping criterion, a third data set is needed to evaluate the actual performance and generalization of the Network. This third set is called the **test set**. The test set is a benchmark used to evaluate the model and only used once a model is completely trained. However, the focus in this project is to accelerate an already trained model and does therefore not have to be trained any further.

3.4 Our FNN model

Now that we have the background, we can start looking at the provided model which is written in Python using the `Pytorch` library.

3.4.1 *Pytorch*

PyTorch is an open source machine learning library based on the Torch library used for applications such as computer vision and NLP [Ketkar 2017]. It is an open source library that is maintained mostly by Facebook's AI research group. Unlike other popular frameworks like TensorFlow, which use static computation graphs, PyTorch uses dynamic computation, which allows greater flexibility in building complex architectures. It uses core Python concepts like classes, structures and conditional loops, which is easier to understand intuitively. This makes it a lot simpler than other frameworks, such as TensorFlow that incorporates their own programming style. We will quickly go through some features of the PyTorch used in the given script, to understand how the model is built.

3.4.2 *Torch*

PyTorch defines a class called `Torch.Tensor` which contains data structures for multidimensional tensors and mathematical operations. It stores and operates on homogeneous multidimensional rectangular arrays of numbers. "PyTorch Tensors are similar to NumPy Arrays, but can also be operated on a CUDA-capable Nvidia GPU. Additionally, it provides many utilities for efficient serializing of Tensors and arbitrary types, and other useful utilities"[Paszke et al. 2019].

3.4.3 *Torch.nn*

The `Torch.nn` PyTorch auto grad makes it easy to define computational graphs and take gradients, but raw auto grad can be a bit too low-level for defining complex Neural Networks. This is where the nn module comes in handy. This model uses `nn.module` and `nn.parameter`. The `nn.module` performs operations on tensors. Modules are implemented as sub classes of the `torch.nn.module` class. All modules are callable and can be composed together to create complex functions. the `nn.parameter` is a tensor that sub-classes the Variable class. The difference between a variable and a parameter comes in when associated with a module. When a parameter is associated with a module as a model attribute, it gets added to the parameter list automatically and can be accessed using the 'parameters' iterator.

The provided FNN model takes in an input x and a target y. The weights and data are generated randomly. To understand all the calculations going through I started with a very small data-set to make sure I understood each process going through the model. This FeedForward Neural Network model consists of two layers shown in Listing 3. This model is a part of a bigger pipeline where the models are trained to recognize

```python
    def _forward_first_layer(self, x):
        batchsize = x.shape[0]
        h = x.mm(self.W0.reshape(self.num_networks * self.hidden_size,
        self.input_size).t())
        h = h.reshape(batchsize, self.num_networks, self.hidden_size)
        hz = self.prelu_z_slopes[None, :, None] * h * (h < 0) + h * (h >= 0)  #
        parametric relu
        hr = self.prelu_r_slopes[None, :, None] * h * (h < 0) + h * (h >= 0)  #
        parametric relu
        return hz, hr

    def _forward_second_layers(self, hz, hr):
        #dual second layer linear transformations
        z = (hz * self.Wz[None]).sum(axis=-1)
        r = (hr * self.Wr[None]).sum(axis=-1)
        return z, r

    def _combine_z_and_r(self, z, r):
        z = 2.0 * torch.sigmoid(self.z_scale[None] * z) - 1.0
        r = F.softplus(r)
        return r * z

    def _ensemble_predictions(self, y):
        y = y.clamp(-self.max_predict, self.max_predict) # clip prediction of each
        network
        y = y.mean(axis=-1) # average over networks
        return y
```

Listing 3: *The different layers and functions used in the FNN Network*

features that indicate an upcoming increase or decrease in the market pricing and bid accordingly. Deep Learning methods, while known in general to be extremely successful in terms of accuracy, also carry a curse of heavy computations with them. We will therefor in the next section implement it with SME and test if we can keep the accuracy and accelerate the inference.

Chapter 4 *Implementation*

This Chapter describes the implementation and gives an overview over the translation of the FNN model to SME. The purpose is to show the structure and thoughts throughout the process of the development of the final SME model. We will highlight the most important technical considerations that were made during the implementation and how the structure was set given the theoretical background from the previous Chapters. This will be illustrated through explanations and code snippets.

4.1 The architecture of SME_ML

Having the theoretical background behind FNN and SME, we will in this section introduce the implementation of the FNN model in SME and describe some of the essential parts. The code that was developed in this work **ML_SME_FPGA** can be found on Github [Moussa 2020]. The result is generalized such that each function can be used for different purposes on an FPGA. The ultimate goal is to find methods for transpiling, that can be generalised to different FNN problems. The python code provided from the HFT company who wants to remain anonymous, was made in python with the *Pytorch* library, which I thereafter translated to C#, using almost no built-in libraries. This was due to the fact that we wanted to understand the data structure of all operations, such that it would be easier to implement with SME. The main intention of each language is different and therefore the transpiling of a process in the given Python code to a SME process might not be completely trivial. However, the automatic parallelization of the network is still useful for users wanting to run the model on a FPGA.

Before getting into the concrete implementation detailing the model, let us restate the purpose of the FNN model and how we want to go from a Python model to a full implementation on a FPGA: Based on an input sequence $\mathbf{x}$ and a set of trained weights $\mathbf{w_i}$, a model is made to predict a $\mathbf{y}$ in Pytorch, which we will recreate in SME. We will do this by being aware of timing and restricted clock cycles such that the calculations runs correctly and independent of each-other. The final model works in the following way:

1. **Takes in** the input sequence $\mathbf{x}$ and the weights $\mathbf{w_i}$.
2. **Runs** the FNN model
3. **Compares** the SME- with the Pytorch -output

The whole architecture is divided into several sections, but to execute this you need to load your data in the `Deflib/data` file and run the `FNN` folder. The `FNN` calls in all the other modules, which can be found in **ML_SME_FPGA**. In the following sections we will describe some specific implementation details that are worth noting.

4.2 Matrix multiplication in SME

Each module is designed as an individual simulation setup so that it can be tested separately from the rest of the system. A further explanation about how to test the modules can be found in Chapter 5. To give an idea of how each function is structured in the core of the FNN, we will look at a concrete example; the `Matmul` function. To understand the development of the process, we will go through the steps of going from the python to SME model. This consists of the following main steps:

- Translation from python to C#

```python
h = x.mm(self.W0.reshape(self.num_networks * self.hidden_size,
self.input_size).t())
```

Listing 4: *Matrix multiplication in python using the `Pytorch` library among other operations*

- Write in the SME framework
- Re-structure functions to run independently

4.2.1 FNN in C#

For this example we will only focus on this specific line from the given FNN model shown again in listing 4. This function calculates the matrix multiplication in the first layer of the model. As we can see in listing 4, before the matrix multiplication, there is a need for a reshaping and transpose of the function, but we will only focus on the matrix multiplication for now. I had to write the function out in C#. This is generally not necessary to do first, but beneficial, since it is needed for the testing. Another reason and something to be aware about is that this is done because in SME in specific situations, calling C# libraries will not translate directly to VHDL, which is the goal. This happens when using the `SimpleProcess`. This process is driven by the global hidden clock with the OnTrigger function that is triggered once in each clock cycle [Johnsen, Thegler, Skovhede, et al. 2021a]. The `Matmul` function is shown in Listing 5. The translation of all lines to the necessary functions such as reshape, transpose, sigmoid ect. was done and tested with several examples to make sure it worked for different cases. I manually tested all functions to start with. This was my way to incorporate learning C# programming.

The next step was recreating the FNN model in C# so it looked as close to the python script. A snippet of the model from C# is shown Listing 6. Specific data was generated and tested a reasonable amount of times on both models to make sure that they both gave the same output.

4.3 Write to SME

The Sigmoid function was made to get introduced to the SME framework, which is shown in A.1. There are many ways to build a function in SME and this was definitely also the case for the `Matmul` process. Several block diagram were made. The final block diagram is shown in Figure 4.1 containing the processes and busses that was necessary for the function to take in data, calculate correctly in the right order and give an output. Because the matrix multiplication is a combination of multiplication and addition, where the operations are dependent on each other, we need other functions to hold on data and forward them. This is where the different processes come in.

```csharp
public static double[,] matmul(double[,] X, double[,] Y)
{
    int x_row = X.GetLength(0); // find dimensions
    int x_col = X.GetLength(1);
    int y_row = Y.GetLength(0);
    int y_col = Y.GetLength(1);
    double[,] result = new double[x_row, y_col];

    for (int i = 0; i < x_row; i++) // loops over row and col
    {
        for (int j = 0; j < y_col ; j++)
        {
            for (int k = 0; k < x_col; k++)
            {
                result[i,j] += X[i,k] * Y[k,j];
            }
        }
    }

    return result;
}
```

Listing 5: *Matrix multiplication written out in C#*

```csharp
public static double[,] matmul_mat(double[,] x, double[,,] W0)
{
    var hh = Functions.matmul(x,
        Functions.transpose(
            Functions.reshape(W0,
                (int)Parameters.num_networks * (int)Parameters.hidden_size,
                (int)Parameters.input_size
            )
        )
    );

    return hh;
}
```

Listing 6: *The C# version of listing* 5

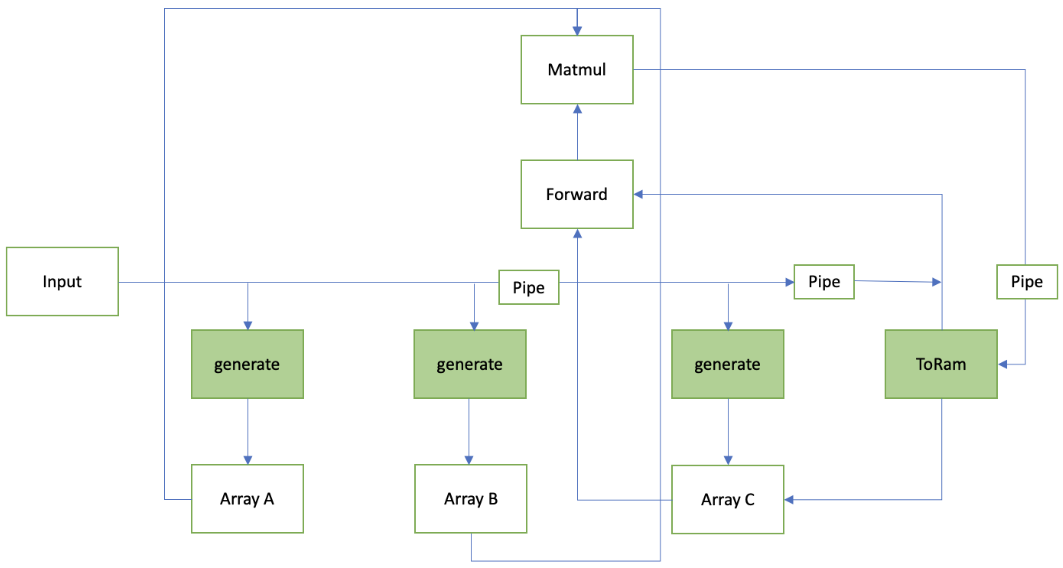


Figure 4.1: *Block diagram of the `Matmul` module. The colored boxes represent the non-clocked processes, since they just are empty 'boxes' with the only purpose of changing the bus type going through them.*

In the following sections we will explain the different processes from the pipeline, used in the implementation.

4.3.1 Generate

The generate function takes the data in their given sizes and and outputs as a flat array, since the *Ram* only takes flat data. The real shapes will be defined when addressing the data i.e. in `MatmulIndex`. This process has an input-bus which holds the base address and the output bus reads the data.

Implementation

To implement the unit we create a SME process. In SME, a process cannot make matrices from lists, as lists are dynamically allocated. However, we can make them as arrays as these can be statically allocated. It needs to be clocked, meaning the unit will activate on a rising clock edge, as it will be part of a closed loop circuit, when we later connect the units. If no unit is clocked in a closed loop, there is no way of knowing, where to begin sending signals and if translated to hardware, one would end up getting a short circuit, which would destroy the chip. Because the process is clocked, when it reads the next address input, it will actually contain the address calculated in the previous clock cycle, as the bus has not been updated yet. This would then be the correct address in the current clock cycle. This is because the bus has not been updated yet. This would then be the correct address in the current clock cycle.

```
        protected override void OnTick()
        {
            output.Enabled = index.Ready;

            if (index.Ready)
            {
                output.Address = index.Addr;
            }
        }
    }
```

Listing 7: *The generate process*

4.4 Matmul and Matmulindex

Matmul

The `Matmul` is implemented as a pipeline, where each sub computation is performed in a single clock cycle. The `Matmul` process takes in the Block RAM that holds the matrices, A and B used for the computation $C+ = A * B$. When C is calculated it gets sent through the `Forward` process and will continue into the `Matmul`. When the calculation is ready it will add C to the next A and B in the matrix. These ports are named Array_A, Array_B, Array_C respectively which are all input busses, meaning the data are stored in these array.

Implementation

This process takes four busses and outputs the calculated value. Three of the input busses are the matrices Array_A, Array_B, Array_C. It also consists of one input-bus *input_pipe* which is a base holding the base address. To give an idea of how a process looks like we will show the whole `SimpleProcess` once including how we define the busses and the structure of the process. The real relevant part here is from line 28, where we define an `OnTick`. This is the part that runs over one clock cycle. If the next input is ready we will calculate the matrix multiplication. With this, we can check if takes data in. From now on I will only show the `OnTick` functions, since they are more relevant. Additionally, since all data needs to be handled as flat lists for the matrices, we have to define another process that keeps track of the real matrix shapes. This function is called `MatmulIndex`.

MatmulIndex

`MatmulIndex` keeps track of the indices in the matrices. It can be thought of as a register, which holds the address of the current instruction. This is very important to have to keep track of where we are in the matrix. It is also here the dimensions of the matrices are being taking into account. The input busses contains of *controlA* and *controlB*. The two control busses both contain a `bool` value which checks whether the dimensions of the input data is correct. The output busses takes the addresses from

```
        protected override void OnTick()
        {
            if (running == true)
            {
                outputA.Ready = true;
                outputB.Ready = true;
                outputC.Ready = true;
                started = true;


               // Console.WriteLine($"i{i} j{j} k{k}");
                outputA.Addr = i * widthA + k;
                outputB.Addr = k * widthB + j;
                outputC.Addr = i * widthA + j;

                k++;

                if (k >= widthA)
                {
                    k = 0;
                    j++;
                }

                if (j >= widthB)
                {
                    j = 0;
                    i ++;
                }

                if (i >= heightA)
                {
                    running = false;
                }
            }
```

Listing 8: *The Matmulindex*

all the matrices, A, B and C. The last *controlout* bus specifies the sizes and validity of the respective matrices.

Implementation

All the control busses are made from the `IndexControl` bus which consists of the following fields: a boolean indicating whether the data is valid, a string holding the base address, an int indicating the access stride, ints specifying the size of the matrix. The main restriction with the `Matmul` function is the multiplication being depend on the accumulation, which is performed within a single clock cycle. To make sure we hold the right data at the correct time we add a process to the pipeline called `Forward`.

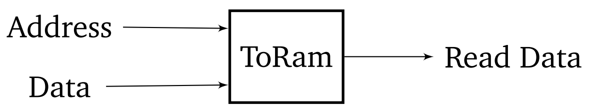


Figure 4.2: *The ToRam Unit.*

4.4.1 ToRam

The ToRam Unit is like Generate, but for the write port to the memory. This is where the final calculations are stored. The FPGA can either read or write to the Memory Unit in a single cycle. The Unit has two inputs: `v_input` (the data) and `index` (the address). It has a single output: `output`, where it writes the data. The ToRam Unit and its connections can be seen in Figure 4.2.

Implementation

The ToRam process takes the Address and Data and should all contain a `int` value. The memory part should be an array, and should read and write On each clock tick, the process should check if the `Index` flag is set, in which case it should read the value on the Address bus, and output the value stored in memory at the read address.

4.4.2 Pipe

Since parts of the Matmul function are dependent on its previous calculation, there will be issues with holding the same positions in the indexing when running over each clock cycle. It is possible to write an indexing process in a way so it reads and write at the same time, nevertheless, this will take longer time when running the code. This is not very efficient, as the clock rate of the processor is determined by the longest path in the processor. A path in a design is the components that a signal goes through, until it reaches a 'ready' state. A ready state in a FPGA tells that the signals are safely stored in the registers. A longer path implies a lower clock rate. So to increase the clock rate, there is a need to decrease the longest path in the processor. For this we introduce `Pipes`.
Pipes are registers in the processor, where all the values computed so far are temporarily stored. It takes all of its inputs, and holds them until the next clock tick, where it will forward the values it is holding. This ensures that the data does not have to travel as far, until it has reached a ready state.

Implementation

Implementing the Pipe Unit should be done with an SME process. To implement the pipe function, we write the same function for as for the $genereate$ function. On each clock tick, the process should check if the `Ready` flag is set, in which case it should read the value on the Address bus, and output the value stored in memory at the read address.

```
        protected override void OnTick()
        {
            if (new_input.Ready && new_input.Addr == old_input.Addr)
            {
                v_output.Data = v_inputNew.value;
            }
            else if (old_input.Ready)
            {
                v_output.Data = v_inputOld.Data;
            }
            else
            {
                v_output.Data = 0;
            }
        }
```

Listing 9: *Forward process*

4.4.3 Forwarding

Since the Matmul calculation output depends on the previous output we need to store the different outputs and therefor need a process that can compare the different addresses and results and forward the final output. What the the Forwarding Unit does is looking at the address *new_input* and checks if the calculated value corresponds with the previous calculated address *old_input*. If they correspond, it will forward the calculated value from the *Matmulindex* function to the Matmul stage. The structure of the simplified processor with the forwarding unit can be seen in Listing 9.

Implementation

Implementing the Forwarding Unit should be done with an SME process. As it interacts with the Matmul stage, it should be put in the Matmul stage. The unit consists of five busses. The first four busses are input busses which we take in. They consist of two busses, which are the pipes that holds the *old_input* and *new_input* addresses. The next two busses are the calculated values of the old and new output and the last bus is an output-bus that forwards the value.

4.5 The FNN model in SME

For the rest of the needed functions in the network, the main challenges has been making sure that the right data and indices were correct. However they were build on the same principles, where the main difference in all of them was the structure of the processes sending the indices around depending on size and dimension. The processes around the main functions in the FNN models, which are used more than once are saved in the module **Deflib**. This also where the data, simulation and C# version of the FNN models is saved. The rest of the relevant processes needed each have their own module.

When I first tested all the functions, they all depended on each other, leaving the debugging process very long and hard. Therefor there was a need to pipeline them all in such a way such that they could be modified and tested independently. All modules in the FNN simulation have been designed to be pipelined. This was an important design consideration because each module consists of many individual calculations. If the system had to wait for each calculation to finish, it would increase the runtime considerably. Each individual module setup consists of one or more `SimpleProcesses` and at least one `SimulationProcess` which generates input data for the network. `SimulationProcess` will be elaborated in Chapter 5. The pipelined structure results in that in each clock cycle, all sections of the modules, are in use. The pipelined system also means that we cannot calculate something in one clock cycle and expect it to still be there a couple of clock cycles down the line. This means that if a result from a calculation needs to be used later on, we either have to save it into RAM or we need to send it through the pipeline until it is needed. With data that is always needed at a specific point in time, it makes sense to send it through the pipeline. Understanding how a module can be build and connected lead us to the rest of the modules and their purpose in the FNN_SME model. The raw connection between them is illustrated in fig. 4.3. A lot of work were put into each module making sure that forwarded the right data and sizes and making sure that the calculations where executed at the correct time. The fully connected FNN model can be seen in Figure 4.4. The instructions on the addresses need to be given explicitly, such that everything gets send around in the right order. This means that we no matter how much we try to generalize the modules, the addresses needs to be changed if this is desired with the model.

Each module consists of a process, program and simulation file. In the process file there are two different processes. The first is indexing of the input. An important notice when working with SME is that the dimensions are fixed to an extend. The biggest problem is that the RAM cannot change after the hardware has been generated. Processes such as index calculation, are done dynamically, and could easily be changed during run-time, since they receive all of the dimensions on the control buses. If one were to truly fix all of the sizes, additional optimizations could be introduced, since we don't have to have the entire precision given by a 32-bit number, when you only need to count to 16.

Transpose

The transpose module is the only function not containing a direct process function since the function in itself changes the address of indices by transposing them. To give a general idea of how the index process looks for most of the modules, an example of the `TransposeIndex` process is shown in Listing 10. The most relevant part in the indices getting addressed from line 1-18. The rest are statements forwarding the information if the indexing holds.

```
protected override void OnTick()
{
    if (running == true)
    {
        started = true;
        j++;
        if (j >= height)
        {
            j = 0;
            i++;
        }
        outputA.Addr = i * height + j;
        outputB.Addr = j * width + i;
        if (i >= width - 1 && j >= height - 1)
        {
            running = false;
        }
    }
    else
    {
        if (controlA.Ready == true)
        {
            started = true;
            running = true;
            width = controlA.Width;
            height = controlA.Height;
            i = j = 0;
            outputA.Ready = true;
            outputB.Ready = true;
        }
        else
        {
            if (started == true)
            {
                controlout.Ready = true;
                controlout.Height = controlA.Height;
                controlout.Width = controlA.Width;
                controlout.OffsetA = controlA.OffsetA;
                controlout.OffsetB = controlA.OffsetB;
                started = false;
            }
            else
            {
                controlout.Ready = false;
            }
            outputA.Ready = false;
            outputB.Ready = false;
        }
    }
}
```

Listing 10: *Transpose index, addressing a*

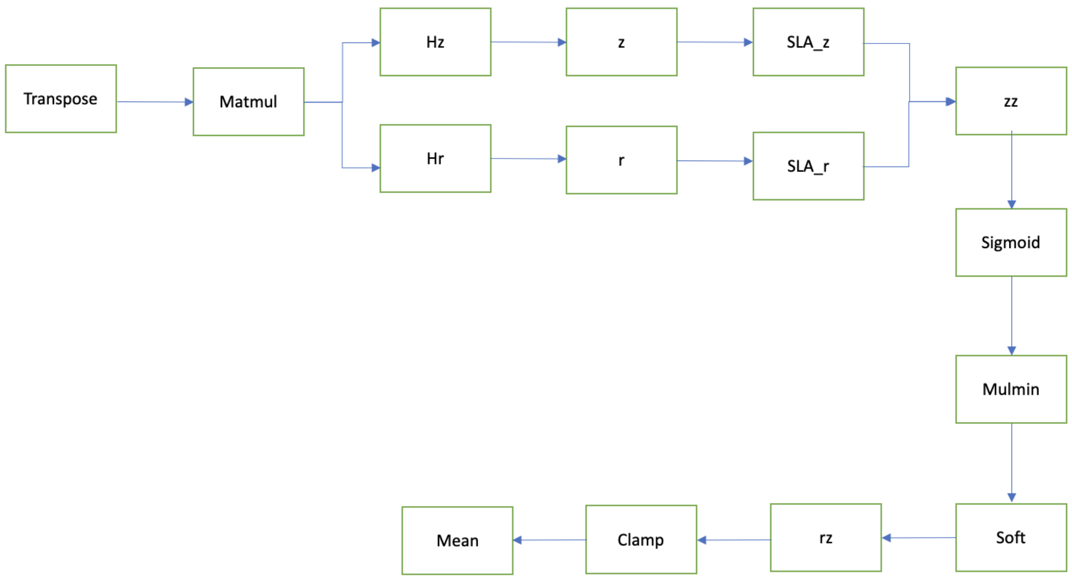


Figure 4.3: *Block diagram of all modules used and their order in the FNN model in SME*

4.6 Quantization of Neural Networks

We are interested in comparing the results from the Pytorch model with the SME model making sure that the functions are written correctly. However, the given data from the company is using floating points for their data while the SME framework supports floats when simulating, but does not yet have floating points implemented for deploying on a FPGA. One solution to compare the Pytorch and SME model, would be to quantazise the data in Python. *Pytorch* has made it possible to quantazise data while training the model. Quantization is a technique which stores tensors at lower bit widths than floating point precision. A quantized model performs some or all of the operations on tensors with integers rather than floating point values. This allows for a more compact model representation and the use of high performance vectorized operations on many hardware platforms [**quantization_pytorch**]. Quantization is primarily a technique to speed up inference and only the forward pass is supported for quantized operators. So to evaluate the data we would need to train the models for quantization. Our focus is not really about training the model in the most efficient way, but rather using it as a tool to be able to get integers out. The *Pytorch* library arguments that it supports integer 8bit (INT8) quantization compared to typical floating point 32bit models allowing for a 4x reduction in the model size and a 4x reduction in memory bandwidth requirements [**quantization_pytorch**]. Depending on the whole pipeline structure there are different approaches to quantazise the model: post training dynamic quantization, post training static quantization, and quantization aware training. Since this is just a small part of a bigger structure, the simplest version would be enough to see if it works.

I chose to play with the python model to start with, since this library seemed very

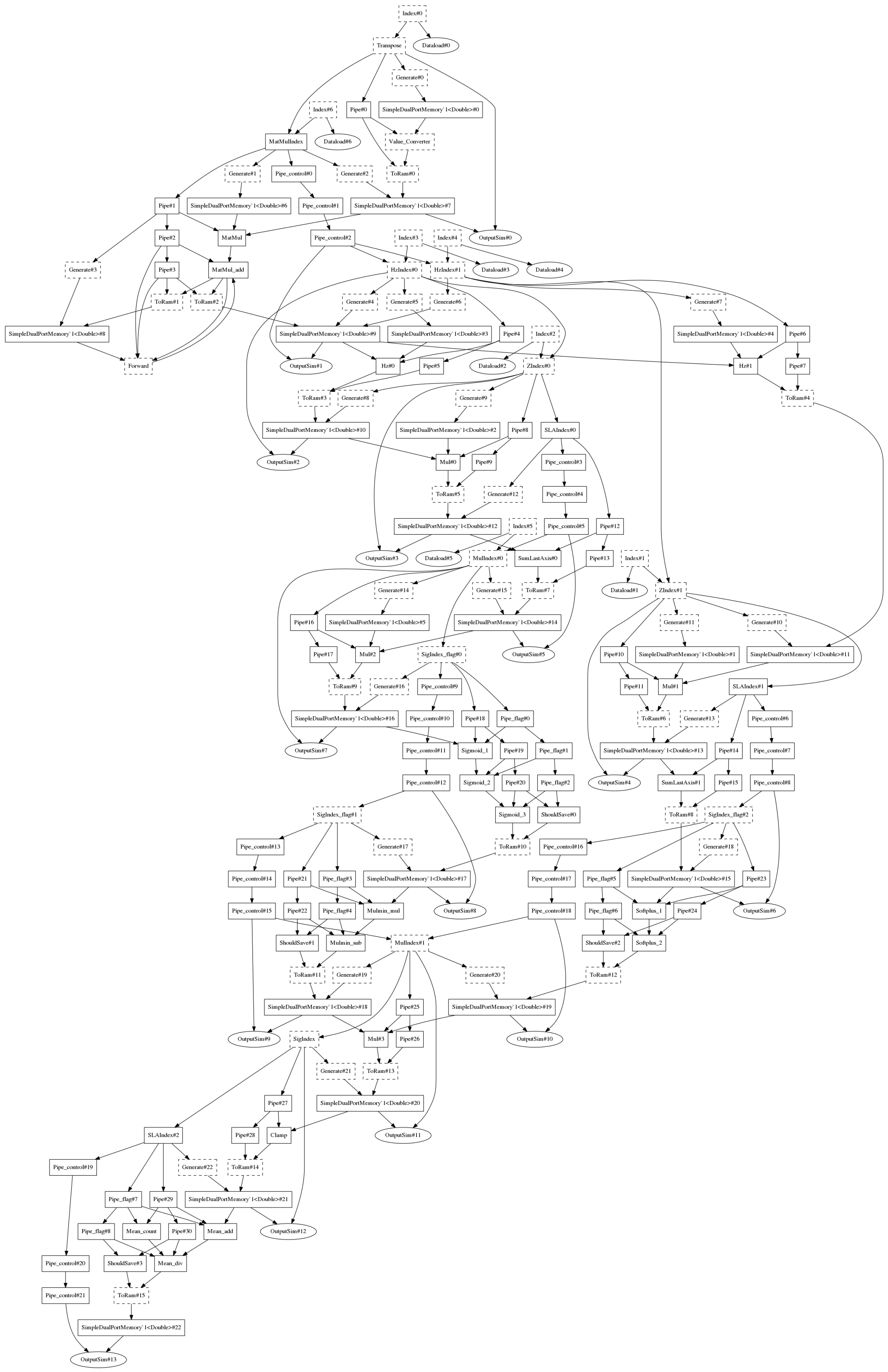


Figure 4.4: *Block diagram of the whole module This diagram is not including all the busses used, but shows the connection of all the processes used*

straightforward and easy, but I must have missed something while working with it. The instructions stated that applying the quantization on the model and then train them would be what was needed to get a set weighted integers. However after several tries the model wouldn't seem to save it any differently. I used time on changing the way of training the model and used all the different quantization approaches without any luck. Since this took too long to figure out and time was getting short, I decided to try to look into another way of getting the SME model to work with floating points.

4.7 ONNX

An interesting thought that came up during this implementation was the question about saving *Pytorch* models and upload them directly in C#, without translating manually and thereby save a lot of time. With the PyTorch framework, it is possible to train a model, save it and download it as an ONNX file to run locally with Windows machine learning in C#. This would save time for future work going from one framework to another and from there rewrite the C# code into SME. In that way we could also test other models containing the same operations we have implemented in SME and compare how fast they execute compared to each other. This is where ONNX comes in handy. ONNX is an open format built to represent machine learning models. ONNX defines a common set of the building blocks of machine learning and deep learning models. It generates a common file format to enable machine learning developers to use models with a variety of frameworks, tools, run-times, and compilers. We managed to save and download the model but could not open a translated `Pytorch` version in C#. Several attempts on using ONNX to convert different deep learning models was made, but without success.

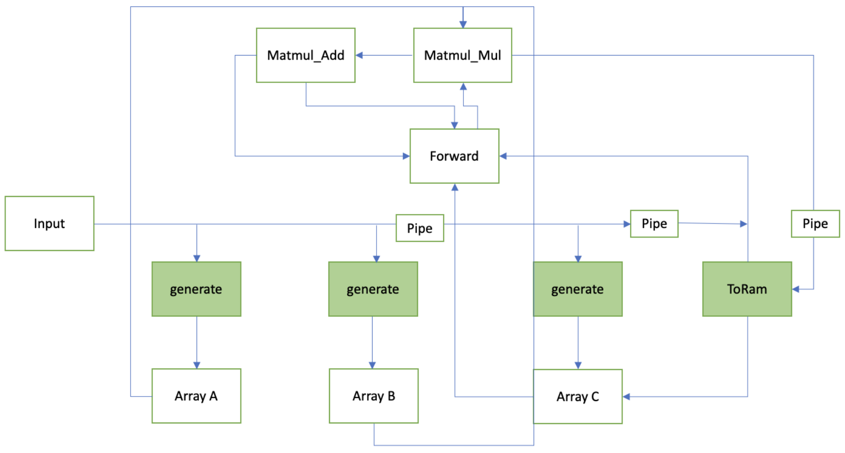


Figure 4.5: *Block diagram of the restructured* `Matmul` *module. The* `Matmul` *process is splitted up to* `Matmul_Add` *and* `Matmul_Mul`

4.8 SME and floating points

Since the time was running out I decided to save this idea of optimizing the performance and just focus on getting this piece of code working. unlike with CPUs/GPUs, FPGAs are more limited in the available calculation methods i.e. the SimpleProcess [Johnsen, Thegler, Skovhede, et al. 2021a]. We can not just import a math library to do advanced functions. Certain calculations have a drastic better performance than others, such as division, and choosing wrong will reduce the performance of the system [Johnsen, Thegler, Skovhede, et al. 2021a]. Since we are targeting Xilinx boards we are restricted to the Intellectual Property (IP) blocks that they provide for floating point numbers. To be able to do calculations consisting of more than one operation in the `SimpleProcess` for the different modules used in the FNN model, we had to do some restructuring to fit it into the possible IP blocks provided by Xilinx. This mean we will have to split the processes up into doing only one calculation.

As an example we will look at the `Matmul` function again. The original version of this process which calculates the `Matmul` consists of different operations such as accumulation and multiplications happening at the same time in the same process. However if this should work for floating points we will need to split it up to such that the FPGA more easily can connect all operations going on. This is therefor done by splitting the `Matmul` Process in two processes; `Matmul_Add` and `Matmul_Mul`. On each clock cycle, one value is read from the A and B matrix, which are streamed to the Multiplier. The Accumulator keeps adding its input, until the address for matrix C changes, where it will write the value to C. To make sure that all data gets sent at the right clock-cycle there needed to be added some flags which is set every time a result needs to be saved. We also would need more pipes to hold the data until it is ready to send to the next process. The new structure of the matrix multiplication is shown as a block diagram in fig 4.5.

4.9 Simulation process

Now, having the structure of the SME feed forward neural network model, we need to test whether or not it gives the same results as the `Pytorch` model. For each process there is a simulation part to test the functions and follow the results. Notably, the final structure is the result of an iterative process, where the structure of testing changed multiple times during the progress of building it. The first simulation to test the functions was built in a way that would encompass all the other processes, because nothing was pipe-lined and would therefore cause frequent errors. However, we managed to pipeline the structure, such that each module is designed as an individual simulation setup so that it can be tested separately from the whole system. When the modules are connected to the entire FNN network system, one `SimulationProcess` generates data and verifies the result. Besides generating input data, the `SimulationProcess` uses selfmade C# libraries, to calculate the results of the simulation as verification for the SME results.

Each module is a folder containing a **process.cs**, **program.cs** and **simulation.cs** file. The simulations file generates the expected output and the test verification happens in program file. For instance, when calculating the `Sigmoid` function, the Testing Simulator will create input data generated from the calculation before, which is from the `zz` then. While the `Sigmoid` is calculated, the Testing Simulator will calculate the actual sigmoid function using standard C#. This way, when `Sigmoid` get computed, it is simple to check whether or not these results are as expected. The output will be a bool of true or false depending on if the results match. Often during development, we ran into errors where the calculations were correct, but the timing was wrong, which meant that the signals arrived unaligned causing the results to be out of sync. By having a standard C# calculation based on the current input data, it was easy to see how the timing of the communication of data was misaligned.
The Simulation is constructed using the following steps: A process, that takes in the data, converts the size and dimensions, since the Ram only takes flattened data and a process that simulates each function and compares it to the actual answer from the model. With this in place, we can test each function and connect all of them together. Ultimately building the final FNN and test it concurrently.

4.9.1 LoadStage

The `LoadStage` is a function that loads the data in, flattens it and indexes it to the correct size and dimensions. It controls whether the index holds their correct size. The `LoadStage` is shown in Listing 11 and consists of three other other processes:

- DataLoad
- TestIndexSim
- Index

```
        public LoadStage(int size, string CSVfile, int row, int col)
        {
            var load_control = Scope.CreateBus<IndexControl>();
            var load_index = Scope.CreateBus<IndexValue>();
            control = Scope.CreateBus<IndexControl>();

            output = new SimpleDualPortMemory<double>(size);

            var generate_load = new Dataload(size, CSVfile, load_index,
            output.WriteControl);
            var load_sim = new TestIndexSim(load_control, row, col);
            var load_ind = new Index(load_control, load_index, control);

            expected = generate_load.input;
        }
```

Listing 11: *LoadData process that loads the input and address the data with their correct sizes*

The **DataLoad** reads the CSV file, the complete size of the data and the value of the index. The output is a flat array which is ready to connect to a given address. Since all the data comes in a flat array, we need to create a process which has the right dimensions of the data to ensure that the sizes hold up. This is where **TestIndexSim** comes in. In **TestIndexSim** we overload with up to three dimensions including an index-control. The index control makes sure to output an error if the sizes does not hold up. When we actually run the simulations the sizes get transferred to the other processes with their given sizes, which can then handle the data for their true sizes. The **index** keeps track of where in the program we are located. It can be thought of as a single register, which holds the address of the current instruction. Further, the index process holds the instruction address. The input bus contains the address of the next instruction and the output bus with the current address.

4.9.2 simulation.cs

When the data is loaded in with their right sizes and dimensions, the next step would be to run them through the simulation processes. Taking the structure of the `Matmul` Function as the starting point, there needs to be a simulation function to test out the results. Before performing matrix multiplication the model reshapes and transposes the data. The simulation process takes in the model which is rewritten in C# and calculates the predicted outcome. In the **program.cs** file there are two parts: first part takes in all the SME processes written for that specific module and pipelines them to get the whole SME `Matmul` function, this part is called the `MatmulStage`. The other half of the **program.cs** takes in the data, the FNN network from both the C# version and the SME and compares them inside the `OutputSim` process. the **OutputSim** is the process that compares the calculated output from SME with the expected output calculated using C# functions. To give a simpler visualisation of the program file, a

```
//expected output of Matmul
var matmul_expected =
Deflib.Functions.Flatten(Deflib.Generate_data.matmul_mat(Deflib.dataMatrix.x,
Deflib.dataMatrix.W0));
// generate data
var matmul_x_ready = new Deflib.TestIndexSim(control_x,
(int)Deflib.Parameters.Batchsize,(int)Deflib.Parameters.input_size);
var prelu_W0_ready = new Deflib.TestIndexSim(control_W0,
(int)Deflib.Parameters.input_size, (int)Deflib.Parameters.num_networks*
(int)Deflib.Parameters.hidden_size);
// SME output of Matmul
var matmul = new MatmulStage(control_x, control_W0, array_x, array_W0);

// simulate and compare outputs
var outsimtra = new OutputSim(matmul.control_out, matmul.ram_out_1,
matmul_expected);
```

Listing 12: *Small snippet of code of the `program.cs` file calculating the matrix multiplication output for the expected and actual output*

snippet of code is seen in Listing 12. Here, it calculates the expected Matmul output, generates the SME data, calculates the SME Matmul output and compares those two at the end with `OutputSim.cs` which prints a **True** if the results matches.

Chapter 5
Results

In this chapter, the results of the thesis will be presented, touching upon two key areas. The proposed methodology from the previous chapters will be adopted and tested on the data of the focal company. Conducting this experiment will provide further insight into the speed and performance of the FPGA and how it performs in a real life setting.

Id	Data	input size	hid size	# networks	max predict	Batch size
1	Test size	4	7	5	1	1
2	Actual sizes	256	96	16	1	1

Table 5.1: *Parameters with different sizes used to test the FNN model in SME*

Name	Clock rate [MHz]	Logic	Registers	Block RAM	DSP
clamp	191.4975	197	419	0	0
hzhr	74.6436	711	1089	1.5	10
load_data	596.3000	993	2367	9.5	56
matmul	51.2242	596	1292	9.5	9
mean	17.7135	1162	1372	0	3
mulmin	51.6022	417	1037	0	4
rz	90.6536	273	526	0	4
sigmoid	17.7135	1933	1267	0	3
softplus	31.1536	1188	1037	0	3
sumlast	45.7603	846	2317	1	6
transpose	884.9557	249	403	16	2
z_r	77.9301	545	1041	3	10
zz	90.6536	273	526	0	4
FeedForward	17.7135	9121	8468	20.5	110

Table 5.2: *Small parameter set. Timing and utilization post Synthesis as reported by Vivado targeting a ZedBoard.*

5.1 Results

To test the processes quickly, a small data set of the different weights were randomly generated and used throughout the construction of the SME model. Since we are not interested in the FNN predictions, but rather if the FNN model SME aligns with the results, using these random values will be sufficient. The first parameter set is one among other small test data to check if all values matched and to test that all stages work by keeping track of the results, since the implementation started without the automated setup of matching the predicted output with the calculated. parameter set 2 are the actual parameter sizes for the network used by the high frequency trading firm. If all executions provides a True, this means that the expected output matches the SME output. The SME model was tested with two different parameter and input sets and we made sure to get a match of the C# and SME results throughout the simulation. This is a deterministic problem, since this can be tested for several parameter sizes. However by changing the dimensions would cause a change of the index addresses, since we are working with a flat array.

Having the simulation work and matching, the next step would be to generate VHDL. In **program.cs** at the end of the code, we can generate VHDL code. As mentioned in chapter 2, SME can be transpiled into VHDL and as such provides a high level approach for hardware design. This will be processed in Vivado [*Vivado Design Suite*

name	Clock rate (MHz)	Logic	Registers	LutRAM	Block RAM	DSP
clamp	184.6722	215	418	48	0	0
hzhr	74.0686	657	1117	48	6	10
load_data	47.6077	4198	2458	2360	384	56
matmul	51.2243	3673	1991	2288	384	11
mean	17.7135	1162	1372	24	0	3
mulmin	51.6022	433	1036	48	0	4
rz	86.9490	259	525	72	0	4
sigmoid	17.7135	1949	1266	48	0	3
softplus	31.1536	1204	1036	48	0	3
sumlast	45.7603	852	2329	48	4	6
transpose	61.0500	785	1427	0	768	6
z_r	74.0686	505	1081	0	12	10
zz	86.9490	259	525	72	0	4
FeedForward	17.71	17515	10695	7223	768	117

Table 5.3: *Large parameter set. Timing and utilization post synthesis as reported by Vivado targeting a ZedBoard.*

n.d.]. Vivado is a tool for synthesizing and implementing hardware designs for Xilinx FPGAs. Vivado supports the VHDL language. This simulation done in VHDL is the closest to the actual hardware, as it visualize all of the different wires, timing and components used in the FPGA.

In this thesis the target board was a ZedBoard [*ZedBoard* 2021]. A ZedBoard is a development board, which contains a Xilinx Zynq system on chip. A Zynq chip consists of a processing system and a programmable logic Farhad Fallahlalehzari 2021. Once the project have been created in Vivado, the behavioral simulation should be run. Since there was not enough time to work with the FPGA itself, the implementation was done in collaboration with Carl-Johannes Johnsen and we will therefore not go into further details about the FPGA, but rather analyse the provided results. These numbers are post synthesis, which means that they are estimates. I.e., given a "perfect" FPGA, we would be able to reach those clock rates and those resource utilizations. After running place and route, we assume that the clock-rate will be lower, which makes it slower. However, utilization should become better, as Vivado can do further optimizations post place and post route. The reason for not doing place and route is due to the BRAM utilization. If any resource is >100% it cannot be done.

The most relevant metrics in the given report are: Logic, LUTRAM, Registers, Block RAM and DSPs, which can be found in 5.4. Block RAM Block ram (BRAM) are memory blocks, which contain more memory than the registers The timing and utilization numbers gathered from the different parameters can be seen in Table 5.2 and . We were able to achieve a clock cycle rate with the lowest bound on 17.714 MHz. The utilization of FPGA resources i.e, the number of Logic used in the implementation for the whole model was 9121 and 17515, respectively for parameter set 1 and 2. This gives a percentage of 17.14% and 32.92% of the available logic on the ZedBoard. The

Resource	Utilization 1	Utilization 2	Available	Utilize 1	Utilize 2
Logic	9121	17515	53200	17.14%	32.92%
LUTRAM	327	7223	17400	1.88%	61.47%
Registers	8468	10695	106400	7.96%	6.79%
BRAM	20.50	768	140	14.64%	548.57%
DSP	110	117	220	50.00%	53.18%

Table 5.4:
Table representing the resource utilization for the whole FNN model on a small FPGA

ratio between how much board uses are around and the ratio between the two parameter sets seems pretty different in size. If the time was there, it would have been interesting to investigate how each of the parameters in Table 5.1 affects the resources used.

The number of clock cycles determines how long it takes to run the entire data set through. The clock-rate is measured in *Mhz*, which is how many clock cycles it can run per second. To have a better understanding of the performance of each process on the FPGA, one can look into the different components seen in Table 5.2 and 5.1. This shows how much of the different resources are available on the very small board. From Table 5.4, there are room for data-set 1. For data-set 2 most of the resources use less space than available but not in *BRAM* when it comes to the data set 2. . In Table 5.2 and both *LUTRAM* and the *BRAM* is mostly used for the `Transpose` and `FeedForward` processes. Since the `Transpose` takes in all the data as a input and outputs all the data transposed, this takes a lot of space in the memory. It is half the size for the `Load_data` and `Matmul`, because all the data only goes in one way. This is due to when connecting all modules together, the input from one stage is the output from the previous.
In general it takes longer to execute with the larger input data x, which makes the network itself larger. Since you can divide by the size of x in the end, to get clock cycles per prediction, and hence time per prediction, then it should not have a big effect on the resulting network. In addition, a larger input x should not have an effect if it is set up correctly. The problem right now is that it runs the whole x through, which means that it must be able to accommodate the whole x and the whole transposed x, which takes up quite a lot of memory. If we ran the batch sizes right, it would be possible to only have room for a single batch. To fix the need for space in BRAM one can either exchange the board with a larger version to be able to run all the data through, or make smaller batches of the data so it stores less inside the board itself while executing.
One could bulletproof test the model, by testing this on several dimensions or similar models, which also was one of the goal with, but due to time restrictions did not become a reality.
One can also compute an estimate of the execution time. This can be done by counting the number of clock cycles used by SME during the simulation and multiplying them

	t_{Python}	t_{SME}	Δt
Test size	0.86 s	$0.034\mu s$	25294.12
Actual size	1.41 s	66.86 ms	21.09

Table 5.5: *Running time for the whole FNN model executed with Python on a CPU and Vivado on a ZedBoard.* t_{Python} *was measured using the* `time` *module in python,* t_{SME} *denotes the time from the simulation in Vivado and* Δt *denotes the scaling factor*

with the time needed for a single clock cycle. At $17.71MHz$ one clock cycle takes $56.45ns$ to run. This is the lowest bound for the clock rates and means this takes the longest. In both tables for the processes, the `mean`, `sigmoid` and `FeedForward` function takes the longest. This is probably because of the use of division. One of the hardest operations in the FPGA is using division and this could probably be optimized if the process was pipe-lined even more such that the steps in calculating the mean was split in more processes running simultaneously. Using the reciprocal instead of division could be an option to spread the utilization of the different on the ZedBoard. One could also be splitting the data into batches again such that the mean evaluates less data at a time. Another way that would increase the running time would be using quantization on the input predictions.

For parameter set 1, with an input data set of 4000 prediction takes 597 clock cycles. If we use these numbers, we can compute an estimate of the execution time. We do this by counting the number of clock cycles used by SME during the simulation and multiplying them with the time needed for a single clock cycle. Thus, 597 times $56.45ns$, gives a running time on $0.034\mu s$ in total for the parameter set 1. For parameter set 2, to run the whole FNN, having 256,000 predictions takes 1,184,385 clock cycles in total, which will take 1184385 times $56.45ns$ giving a running time on $66.86ms$ in total. If we divide the running times with the predictions we get how long it takes to run a single prediction which is $261.1847ns/pred$. We can find the final running times for the SME and Python implementation of the whole FNN, where the comparison is shown in Table 5.5. The Python code was executed on an Apple Macbook Pro carrying an M1 chip. For more information about the machine see Table 5.6. We see that for all cases, the SME implementation performs better than the Python implementation. The reason the scale is so immense for the test size is due to python having a "heavy" startup time. I.e. it's quite fast if discarding the warm-up time. However this should be taken lightly. The Python implementation is running directly from the computer and is not set to utilize the full processor, which can be done. We managed to design and implemented the FNN model in SME and successfully generated a VHDL code which was then successfully run through Vivado. From the results, we see that we have accelerated the FNN model in general and with an optimization on almost 21 times faster this seems like a great update for the high frequency trading company. However the inference could be optimized quite a lot. There could have been used more time on rewriting the different processes such that there were more focus on the clock-cycles, by using more pipe-lining on the parts that uses to much memory.

CPU	8-core CPU 3.2 GHz
ARM	ARMv8.4-A
Xillix Vivado	2020.2
Python	version 2.7.16
Pytorch	version 1.8.0
FPGA board	Zedboard
FPGA Chip	Xillix Zynq Z020

Table 5.6:
Specifications of the Apple Macbook Pro carrying an M1 chip, running the FNN model. This is along with the version numbers of the programs used.

Chapter 6
Conclusions & Further Work

This chapter rounds up this thesis and gives some suggestions towards eventual future work. It briefly discusses how the developed model could be used in other aspects of machine learning and how we could improve SME to work with other high productivity languages as well as how the network could be improved with respect to computational performance.

6.1 Conclusion

Throughout this project the theory behind the FNN model implementation has been covered, starting from understanding the basic properties of the FPGAs. Hereafter the syntax of Synchronous Message Exchange (SME) was introduced, where the theory covering the implementation was explained. An introduction to Feed-forward neural networks and the provided script was described. Finally the given knowledge was then used to design the architecture of the FNN model in SME. The new design was then implemented in SME and we managed to successfully run the FNN test program. The generated VHDL was then successfully run in Vivado and it can be concluded from chapter 5 that the simulated speed for the provided parameters took $66.86ms$ compared to the python code which took $1.41s$ with a speedup on almost 21 times more.
With this proof of concept we have shown that with SME and no beforehand knowledge in hardware and programming it is possible to implement code down to FPGAs. This also shows that FPGAs are good alternatives in the field of machine learning, at least for inference. A wider application of the concepts on a bigger pipeline of artificial intelligence to make the full potential of faster inference would be an interesting next step.

6.2 Future Work

6.2.1 Specific ML package to FPGA

This idea was also chosen by the SparNord foundation and I received a scholarship to go to University of California, Berkeley and work on this as a start-up idea. There has been a lot of positive feedback and requests on expanding the deep learning library. The idea of a simpler work tool for implementing machine learning on FPGAs is catching companies interest everywhere, the demand on faster models is increasing and the growth of PyTorch users are expanding. It could therefore be interesting to extend the SME- Pytorch library which could optimize the algorithms and maybe get more FPGA customers and users. This is also where ONNX could be a great bridge for exactly this. A lot of time on translating a Neural Network model from one language to another would be needed with ONNX. This would speed up the process of developing more machine learning libraries in SME. Furthermore, one wouldn't be limited just to use Pytorch, but would have the flexibility of every framework that can target ONNX, which also targets TensorFlow. This would be able to target both of the two big Neural Networks vendors.

6.2.2 Performance Improvements

To increase the speed of the processor there could have been used more time on rewriting the different processes such that there were more focus on the clock-cycles. This would allow the electrons to run over less distance and therefor fasten the speed up for each clock-cycle. This would also allow the processor to work on multiple instructions in a single clock cycle and by that increase the speed of the FPGA by adding more parallelism by pipeline the processes even more. To test the performance of the processor a larger workload, it would be interesting to use a larger amount of parameters of data to get closer to a real ML problem.

6.2.3 ONNX and quantization

Another interesting approach to avoid translating different frameworks and languages to C# would be looking at incorporating ONNX to save time on translating models, so the focus can go to optimizing the pipe-lining of the model in SME and accelerating the run time. Since SME does not have floating points incorporated yet, some manual translation needs to be done in VHDL, by putting each process in an IP block, quantization in general would be helpful. Both because it would save time to use int in SME but also because Pytorch speeds up the performance and could hence give an even faster run-time on the FPGA.

Chapter A
Appendix

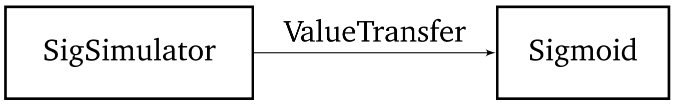


Figure A.1: *The rectangles represent individual processes, which lies in a file of its own. The solid arrow represents a bus which we call `ValueTransfer` in this example.*

A.1 SME example of a sigmoid function

To show how SME works, a guide explaining every step is given, so a person with almost no coding experience is able to try it out. we are going to implement a simple function that is already made in *C#*. It takes a value in as an input and spits a result out. We will implement the sigmoid function which is described in Sec. 3.2. Before you can run this example, make sure you have installed:

- .Net Core SDK with at least version 2.0
- VS Code (or another editor) and the C# extensions

as they work well on all operating systems:
For this quick example we are going to start a project in a folder we create called **SIGMOID**, which is where all shown files are going to be placed. To do this we are going to open up a terminal window and navigate to the desired folder destination which could be the Desktop as an example and invoke the command

```
$ cd Desktop
$ dotnet new console -n SIGMOID
```

next we go into the folder by

```
$ cd SIGMOID
```

and add the necessary SME libraries

```
$ dotnet add package SME -- version =0.4.0 - beta
$ dotnet add package SME . GraphViz -- version =0.4.0 - beta
$ dotnet add package SME . Tracer -- version =0.4.0 - beta
$ dotnet add package SME . VHDL -- version =0.4.0 - beta
```

There should now exist 3 items inside the SIGMOID folder: Program.cs, SIGMOID.csproj and a folder called obj. If all went well we should now be able to begin with our project.

Since all functions has to be build through busses and processes, it can easily end up with complicated systems. The best way to get an overview is to make a drawing over your system. Even though this system is simple we will still make a drawing to understand how it works.

As the picture shows we will need to add two more files. One for the simulation process generating values and one process defining the sigmoid function. We will call them respectively SigmoidSimulation.cs, Busses.cs and Sigmoid.cs. The Program.cs file will be our main file where we connect all the processes together.

A.1.1 Program.cs

Opening up the Program.cs file, we are going to add a couple of lines, as shown in Lst. 13. First, the namespace in all files should be same. Program.cs contains the Main() method of the project and is the entry point of any C# program. To use the SME library we import it with the "using SME" command as shown in the second line. Within the Main() method, the first function we meet is the using function, this is just to ensure the resources are properly cleaned up after the simulation. Hereafter we see the simulation object, which as the name implies, is responsible for the simulation of the logic unit.

We define `simulation data` as a bus, since it will pass the value from the simulation generating data to where-ever we choose. The same counts for the `sigmoidresult`, because we need to pass the results as an output. Then we need a `simulator`, which we will define shortly as a process, where we insert the simulation-data. Lastly we have the `sigmoid` function which takes the input data and gives data to the `sigmoidresult` which then outputs the results. We can configure the simulation with the sim object, which uses fluent syntax. It should be noted that Run() should always be the last method called.

A.1.2 Sigsimulator.cs

The simulation file is shown in Lst. 14. Since the `bus.cs` file only contains two lines, we will put it in the same file as `sigsimulator.cs` The first definition inside the simulation process creates the bus `Valuetransfer`, which will as the word imply, transfer the values we simulate in the Sigsimulator. Notice that we define that `ValueTransfer` in the Sigsimulator to be an output as `m_output` such that we can take the simulated data and transfer it over to the sigmoid function, see figure 14. Then we define the sigsimulator to take in the `ValueTransfer` as an output value. On line 19 - 22 we declare our output bus. The way it is written is just pure SME syntax.
We want a new clock cycle to occur, so we use the line await ClockAsync(). In a simulation process any .NET library is allowed and is not going to get transpiled to a VHDL file. Therefore we can print the output of the AND gate to console to see whether or not it works correctly.

A.1.3 Sigmoid function

Lastly we are going to define the sigmoid function itself. We open the file Sigmoid.cs. We see how the process is defined in 15. Since the Sigmoid process is only going to execute once per cycle, we are going to inherit from the SimpleProcess class. In line 8-11 define a `m_input` Bus that takes an input in and a `m_output` Bus that calculates

```
1   using System;
2   using SME;
3
4   namespace SIGMOID
5   {
6       class MainClass
7       {
8           public static void Main(string[] args)
9           {
10              using (var sim = new Simulation())
11              {
12
13                  var simulationdata = Scope.CreateBus<ValueTransfer>();
14                  var sigmoidresulst = Scope.CreateBus<ValueTransfer>();
15
16                  var simulator = new SigSimulator(simulationdata);
17                  var sigmoid = new Sigmoid(simulationdata,sigmoidresulst);
18
19
20                  sim
21                  .Run();
22              }
23
24          }
25      }
26  }
```

Listing 13: *The Program.cs file, which contains the Main() method for the project*

the result. We declare the busses and then at line 21 we define the actual Sigmoid function and print it out to see the results.
Now we go back to terminal and run the code.

```
$ dotnet run
```

this will output a VHDL folder with VHDL files, with one named 'makefile'. We can use a VHDL simulator to test it out. I use the GHDL simulator to do this. If you have downloadet GHDL down on your computer, you can navigate to the place where the VHDL folder is and run the following command in your terminal:

```
$ make
```

```
using System;
using SME;

namespace SIGMOID
{

    /* Defining the bus*/
    public interface ValueTransfer : IBus{
        double value {get; set;}
    }

        /*defining sigsimulator*/
       public class SigSimulator : SimulationProcess{

        [OutputBus]
        private ValueTransfer m_output;


        public SigSimulator ( ValueTransfer output)
        {
            m_output = output ?? throw new ArgumentNullException(nameof(output));
        }

            public async override System.Threading.Tasks.Task Run()
        {
            await ClockAsync();
            for (int i = 0; i < 10; i++)
            {
                m_output.value = i;

                await ClockAsync();
            }
        }
    }
}
```

Listing 14: *The SigmoidSimulator.cs file, which generates values from 1 to 10*

```csharp
using System;
using SME;

namespace SIGMOID
{

    public class Sigmoid: SimpleProcess {
        [InputBus]
        private ValueTransfer m_input;
        [OutputBus]
        private ValueTransfer m_output;

        public Sigmoid(ValueTransfer input, ValueTransfer output)
        {
            m_input = input ?? throw new ArgumentNullException(nameof(input));
            m_output = output ?? throw new ArgumentNullException(nameof(output));
        }

        protected override void OnTick()
        {
            var tmp = 1 /(1 + Math.Exp(m_input.value));
            Console.WriteLine(tmp);
            m_output.value = tmp;
        }
    }
}
```

Listing 15: *The Sigmoid.cs file, which contains the sigmoid function and the bus that will pipe the data*

Bibliography

[1] Anuj Agarwal. "High frequency trading: Evolution and the future". In: *Capgemini, London, UK* (2012), p. 20.

[2] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. "Performance comparison of FPGA, GPU and CPU in image processing". In: *2009 International Conference on Field Programmable Logic and Applications*. 2009, pp. 126–131. DOI: *10.1109/FPL.2009.5272532*.

[3] Mike Brogioli. "Chapter 6 - The DSP Hardware/Software Continuum". In: *DSP for Embedded and Real-Time Systems*. Ed. by Robert Oshana. Oxford: Newnes, 2012, pp. 103–111. ISBN: 978-0-12-386535-9. DOI: *https://doi.org/10.1016/B978-0-12-386535-9.00006-8*. URL: *https://www.sciencedirect.com/science/article/pii/B9780123865359000068*.

[4] Rod Burns et al. "Accelerated neural networks on OpenCL devices using SYCL-DNN". In: *Proceedings of the International Workshop on OpenCL*. 2019, pp. 1–4.

[5] Jason Cong et al. "Understanding Performance Differences of FPGAs and GPUs". In: Apr. 2018, pp. 93–96. DOI: *10.1109/FCCM.2018.00023*.

[6] Farah Fahim et al. *hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices*. 2021. arXiv: *2103.05579* [cs.LG].

[7] Jian Fang et al. "In-memory database acceleration on FPGAs: a survey". In: *The VLDB Journal* 29.1 (2020), pp. 33–59.

[8] Farhad Fallahlalehzari. *Introduction to Zynq Architecture*. *https://www.aldec.com/en/company/blog/144--introduction-to-zynq-architecture*. 2021.

[9] Virgilio Gianluca. "Is High-Frequency Trading a Threat to Financial Stability?" In: (2017).

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

[11] hls4ml Contributors. *hls4ml*. *https://fastmachinelearning.org/hls4ml/*. 2021.

[12] C. A. R. Hoare. "Communicating sequential processes". In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 00010782. DOI: *10.1145/359576.359585*. URL: *http://portal.acm.org/citation.cfm?doid=359576.359585*.

[13] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural networks* 4.2 (1991), pp. 251–257.

[14] Carl-Johannes Johnsen, Alberte Thegler, Kenneth Skovhede, et al. "Accelerating Molecular Dynamics with the Lennard-Jones potential for FPGAs". In: (2021).

[15] Carl-Johannes Johnsen, Alberte Thegler, Kenneth Skovhede, et al. "SME: A High Productivity FPGA Tool for Software Programmers". In: *arXiv preprint arXiv:2104.09768* (2021).

[16] Carl-Johannes Johnsen, Alberte Thegler, Brian Vinter, et al. "SME: A High Productivity FPGA Tool for Software Programmers". In: *IEEE TRANSACTIONS ON COMPUTERS, SPECIAL ISSUE ON COMPILER OPTIMIZATIONS FOR FPGA-BASED SYSTEMS* (2020).

[17] David H. Jones et al. "GPU Versus FPGA for High Productivity Computing". In: *2010 International Conference on Field Programmable Logic and Applications*. 2010, pp. 119–124. DOI: *10.1109/FPL.2010.32*.

[18] Dan Jurafsky and James H. Martin. *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition*. Upper Saddle River, N.J.: Pearson Prentice Hall, 2009. ISBN: 9780131873216 0131873210.

[19] Nikhil Ketkar. *Introduction to PyTorch*. Oct. 2017, pp. 195–208. ISBN: 978-1-4842-2765-7. DOI: *10.1007/978-1-4842-2766-4_12*.

[20] David Kriesel. *A Brief Introduction to Neural Networks*. 2007. URL: *available%20at%20http://www.dkriesel.com*.

[21] Leslie Lamport and Fred B. Schneider. "The "Hoare Logic" of CSP, and All That". In: *ACM Transactions on Programming Languages and Systems* 6.2 (1984), pp. 281–296. ISSN: 01640925. DOI: *10.1145/2993.357247*. URL: *http://portal.acm.org/citation.cfm?doid=2993.357247*.

[22] Christian Leber, Benjamin Geib, and Heiner Litz. "High frequency trading acceleration using FPGAs". In: *2011 21st International Conference on Field Programmable Logic and Applications*. IEEE. 2011, pp. 317–322.

[23] John Lockwood et al. "A low-latency library in FPGA hardware for High-Frequency Trading (HFT)". In: Aug. 2012, pp. 9–16. DOI: *10.1109/HOTI.2012.15*.

[24] Jan Lönnberg and Anders Berglund. "Students' understandings of concurrent programming". In: *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research-Volume 88*. 2007, pp. 77–86.

[25] Warren Mcculloch and Walter Pitts. "A Logical Calculus of Ideas Immanent in Nervous Activity". In: *Bulletin of Mathematical Biophysics* (1943), pp. 127–147.

[26] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.

[27] Amira Moussa. *Github code to ML_SME_FPGA*. 2020. URL: *https://github.com/amir0135/ML_SME_FPGA*.

[28] Muryshkin Evgeny. *QuokkaEvaluation*. *https://github.com/EvgenyMuryshkin/QuokkaEvaluation*. 2018.

[29] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: *http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf*.

[30] Wei QingJie and Wang WenBin. "Research on image retrieval using deep convolutional neural network combining L1 regularization and PRelu activation function". In: *IOP Conference Series: Earth and Environmental Science* 69 (June 2017), p. 012156. DOI: *10.1088/1755-1315/69/1/012156*. URL: *https://doi.org/10.1088/1755-1315/69/1/012156*.

[31] Martin Rehr, Kenneth Skovhede, and Brian Vinter. "BPU Simulator". In: *Communicating Process Architectures 2013*. Ed. by Peter H. Welch et al. Nov. 2013, pp. 233–248. ISBN: 978-0-9565409-7-3.

[32] Esben Skaarup and Andreas Frisch. "Generation of FPGA hardware specifications from PyCSP networks". PhD thesis. Master's thesis, University of Copenhagen, Niels Bohr Institute, 2014.

[33] Michael Stonebraker. "The case for shared nothing". In: *IEEE Database Eng. Bull.* 9.1 (1986), pp. 4–9.

[34] Berkeley University of California. *chisel*. *https://www.chisel-lang.org/*. 2012.

[35] Mário Véstias and Horacio Neto. "Trends of CPU, GPU and FPGA for high-performance computing". In: Sept. 2014, pp. 1–6. DOI: *10.1109/FPL.2014.6927483*.

[36] Brian Vinter and Kenneth Skovhede. "Bus Centric Synchronous Message Exchange for Hardware Designs". In: *Communicating Process Architectures 2015*. Ed. by Kevin Chalmers et al. IOS Press, Amsterdam, The Netherlands, Aug. 2015, pp. 257–268. ISBN: 978-XXX.

[37] *Vivado Design Suite*. *https://www.xilinx.com/products/design-tools/vivado.html*. [Online; accessed 25-July-2017].

[38] Kyongsik Yun, Alexander Huyen, and Thomas Lu. "Deep neural networks for pattern recognition". In: *arXiv preprint arXiv:1809.09645* (2018).

[39] *ZedBoard*. *https://www.xilinx.com/products/boards-and-kits/1-elhabt.html*. 2021.