Accelerating DNA basecalling of Nanopore reads on FPGAs

J. Haenen

TU Delft

Accelerating DNA basecalling of Nanopore reads on FPGAs

by

J. Haenen

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday October 17, 2023 at 10:00 AM.

Student number:	4953266	
Project duration:	February 13, 2023 – October 17, 2023	
Thesis committee:	Prof. dr. H. P. Hofstee,	TU Delft, supervisor
	Dr. ir. Z. Al-Ars,	TU Delft
	Dr. J. P. Gonçalves,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

Genomics has revolutionized our understanding of evolution, hereditary diseases, and more. The advent of long-read DNA sequencers i.e. Oxford Nanopore Technologies' innovations, has opened many new research potentials in genomics. These sequencers produce significantly longer DNA reads, facilitating novel applications. However, this technological leap brings challenges, particularly in accurate basecalling which is the process of converting raw sequenced measurements into digital base pair sequences. While advances in basecalling accuracy have been steadily improving over the years, the computational intensity remains a bottleneck in genomic analysis workflows, demanding costly high-end GPUs for probabilistic neural network models.

The main problem this thesis addresses is the implementation of an accelerated hardware solution for the compute-intensive process of basecalling long-read sequences. The thesis presents an FPGA-based implementation of the computationally demanding Long Short-Term Memory (LSTM) layers within the basecalling network known as Bonito. However, due to the lack of floating-point arithmetic units available on the FPGA, the FPGA implementation could not achieve competitive performance compared to GPUs.

While the FPGA implementation falls short of GPU performance, it serves as a possible stepping stone toward developing an ASIC solution for implementing the Bonito LSTM layers or potentially implementing the entire Bonito model. An ASIC implementation has the potential for superior performance up to 9 times faster than a GPU implementation while additionally being cost-effective. This suggests that ASICs hold promise as a future direction for accelerating long-read sequence basecalling, allowing for faster and more affordable genomics research.

Preface

Working on this thesis presented a fair share of challenges and disappointments. Despite of this difficult journey I am very proud of the results we have achieved. It provided me with an opportunity to expand my knowledge about hardware design and genomics.

I would like to express my gratitude to all the participants who played a significant role in the results and completion of this thesis. At first I would like to express my gratitude to my supervisor Prof. Dr. Peter Hofstee for his guidance and support throughout this research. His knowledge and insights have been very valuable in shaping this work. Also I would like to thank Dr. Zaid Al-Ars, Tanveer Ahmad, and the whole ABS Group of TU Delft for their constructive discussions and contributions.

J. Haenen
The Hague, September 2023

Contents

1 Introduction

Genomics is an important field of science enabling research to understand better the evolution of humanity and other organisms to curing hereditary diseases and much more. With the advent of long-read DNA sequencers developed by i.e. Oxford Nanopore Technologies over the past decade, a new large step has been made in genome sequencing allowing for much larger preserved DNA reads up to 500 times longer than was possible before; ushering in a new generation of sequencing that allows for a multitude of new applications. However, with the introduction of new technology, there are always difficulties in the early stages which is not different for long-read sequencing. While improved greatly over the years, the accuracy of long-read sequencing has been a challenging facet where only recently accuracy has been high enough to be utilized for non-experimental genomics analysis.

Long-read sequencing can be subdivided into two phases: measuring the DNA using chemical processes, and converting the measured signals into digital representations of the base pairs of the measured DNA. The conversion of the measurement to the base pairs is called basecalling. While both phases contribute to the accuracy of long-read sequencing, the main challenge currently lies in the measurement output of the sequencer. Due to a multitude of factors, these measurements can not be deterministically converted to base pairs, therefore, requiring probabilistic models to perform these conversions. However, the problem is that due to the continued increase in accuracy over the years, these probabilistic models have become larger over those years, dramatically increasing computational intensity. Currently, basecalling has the longest process duration in long-read genomics analysis workflows while additionally requiring expensive high-end compute hardware to run those probabilistic models.

This thesis looks at accelerating the basecalling process for Oxford Nanopore Technology long-read sequencers using FPGAs. The basecalling models are identified and one model, called Bonito, is selected to accelerate. The Bonito model has neural network layers, called long short-term memory (LSTM) layers, that take up the majority of the computing time of the models. This thesis provides an FPGA implementation for the LSTM layers that are present in the Bonito model. Moreover, the thesis analyzes the potential benefits of converting the FPGA implementation to an ASIC, which can theoretically achieve comparable results to high-end graphic cards.

1.1. Problem definition and research questions

The main problem this thesis addresses is the implementation of an accelerated hardware solution for the compute-intensive process of basecalling.

To address this problem, we focus on answering the following research questions.

1. What are the main computational bottlenecks faced by basecalling applications?
2. What hardware architectures and designs are suitable for eliminating these bottlenecks?
3. What is the expected performance gain of using these architectures for our application?

1.2. Thesis outline

Chapter 2 of the thesis looks at the background of the thesis project, discussion long-reads, and the long-read pipelines. Moreover, it discusses why it has been chosen to accelerate the basecalling process of the long-read pipelines. Chapter 3 first looks at the state of the art of long-read basecallers and discusses the reason for selecting the Bonito model. After that, it explores the Bonito and explains its architecture. Chapter 3 closes off by exploring the different possible methods to implement the Bonito model. Chapter 4 discusses LSTMs and outlines how the FPGA implementation of the LSTMs of the Bonito model is implemented. Chapter 5 evaluates FPGA implementation, the FPGA implementation is evaluated on three aspects: FPGA resource utilization, performance, and validity. Last, Chapter 6 discusses the potential of implementing the FPGA implementation on an ASIC and discusses other future avenues to be further researched or implemented to enhance the current implementation.

1.3. Contributions

This thesis has multiple contributions, these contributions are as follows:

- A state-of-the-art analysis of long-read basecalling implementations providing an overview of the development of basecallers.
- Documentation of the Bonito model and its inner workings which is not provided by the maintainer Oxford Nanopore Technology. Moreover, the models are evaluated for performance.
- An evaluation of possible implementation methods for Bonito on FPGAs.
- A VHDL implementation of the LSTM layers of the Bonito model. Compared to existing LSTM implementation in literature, this thesis implementation deals with significantly larger layer sizes and multiple layers.
- A evaluation of the VHDL implementation and a comparison to the GPU counterpart implementation of Bonito.
- A explorative study that looks at the possibilities and potentials of mapping this LSTM implementation to an ASIC.

2

Background

Since the end of the 20th century, the field of genomics has been on the rise, with various factors contributing to its development. First, it became evident that many diseases can be traced back to the malfunctioning of a single gene was largely incorrect. In almost all cases, there is a complex interaction of many different genes involved. With genomics, the focus shifted from solely examining individual genes to additionally studying the interconnectedness of genes [1].

A large field of genomics is sequencing, which is the process of determining the precise order of nucleotides in a DNA (bases A, C, T, and G) or RNA molecule (bases A, U, T, and G), to enable the study of genetic information and various biological processes. Currently, many facilities utilize 2nd generation sequencing, also known as short-read sequencing, which has been in existence since 2005. However, around 2014 Oxford Nanopore technology introduced a sequencer capable of reading much longer reads, approximately 50 to 500 times larger. This technology is referred to as long-read sequencing or 3rd generation sequencing. The longer reads provided by this technology offer numerous advantages in genomics. Nonetheless, long-read sequencing has presented several challenges that need to be addressed for its widespread viability and adoption in the field [1].

This chapter will first analyze the difference between short and long reads and the challenges of long reads. After that, the long read pipeline will be analyzed to identify potentials to use for acceleration on an FPGA (Field-Programmable Gate Array). Subsequently, a stage from the pipeline will be chosen to use for further acceleration.

2.1. Short reads and long reads

Long-read sequencing has emerged as a promising solution to address the main challenge encountered in short-read sequencing methodologies. By producing longer contiguous reads of the genome, long-read sequencing significantly simplifies the genome assembly process compared to short reads. Genome assembly refers to the reconstruction of all the fragmented pieces of the genome before they are separated from each other. Short reads typically consist of an average of 200 base pairs per read, while Oxford Nanopore long reads offer an average of 100 thousand base pairs per read. Considering the human genome, this implies that approximately 450 million short reads are required for assembly, whereas only 900 thousand long reads are needed [2]. The apparent advantage of long reads becomes evident as the genome is split into fewer DNA pieces, simplifying the puzzle of reassembling them into a complete genome, facilitating the correct alignment of long repeat sequence regions, and determining haplotypes [3]. A haplotype is a specific combination of genetic variations on a chromosome inherited as a unit [1].

However, long-read sequencing faced multiple challenges in its early stages, including being slow, error-prone, and expensive. Nevertheless, improvements have been made over the past decade [2]. In 2015, sequencers had an accuracy of approximately 60%, with large variability, making early genome assembly highly challenging or nearly impossible [4]. Today, state-of-the-art PacBio and Oxford Nanopore Technologies (ONT) sequencers have achieved an average accuracy of around 99% in their highest accuracy modes [2]. This improvement in accuracy has played a crucial role in enabling more reliable and efficient genome assembly using long-read sequencing technologies.

While long-read sequencing with current technology still suffers from not high enough accuracy, it is undoubtedly that with current trends long-read sequencing will replace short-read sequencing, and that short-read will mostly be applied with hybrid approaches [5][6][7].

2.2. The long-read pipeline

The long-read analysis can be viewed as a pipeline. However, there is not one definitive pipeline as there are multiple different applications to apply to the DNA sample. The following are the most common pipeline stages:

- DNA sample collection and preparation
- Sequencing the DNA
- Basecalling the sequenced data
- Analysis
  - Read alignment and variant analysis
  - De novo genome assembly

While there are undoubtedly more analysis applications the listed workflow above is currently the most well-proven analysis method that benefits by using long-sequencing [2]. It is important to note that the first three steps (collecting/preparation, sequencing, and basecalling) are fundamental to the long read pipeline and are as such present in each pipeline no matter the application [2][8]. The following sections go more in-depth on the pipeline stages.

2.2.1. DNA sample collection

This is a bio-chemistry stage of the pipeline, which includes gathering the DNA, isolating it, and preparing it to be sequenced by the nanopore sequencer. This stage will be briefly described as it does fit entirely into the scope of the thesis.

Human DNA is gathered from patients or test subjects by often collecting their blood, however, other methods like saliva, muscle tissue, etc. are possible but less common. The blood of the patients is chemically cleaned by removing the red blood cells, as they contain no DNA, and keeping the white blood cells which do contain DNA. These white blood cells are destroyed with salt so that the DNA can come out and then the DNA is chemically isolated [9]. The process of isolating long-reads compared to short-reads is more complex as the isolation process has to be more delicate in long-reads. In long-reads, High Molecular Weight DNA Isolation (HMW-DNA) is applied to isolate the DNA, HMW-DNA allows for reads isolated in length over 50 kbases [10].

After the DNA is isolated, it has to be prepared by a technique called library preparation. This library preparation can be used for multiple applications but the main applications are:

- **Applying motor and attachment proteins**, these proteins help the DNA strains travel to the nanopores in the sequencer and attach to the pores, to be pulled through. To attach these proteins to the DNA, a piece of adapter DNA is attached to the DNA to be pulled through. This adapter DNA allows the proteins to attach to the DNA [11].
- **Barcoding** which is a cost-reducing technique where barcode sequence is added to the DNA, so that multiple patients' DNA can be batched to safe cost, and later after sequencing can be separated by looking for the barcode sequence for each patient in the result [12].
- **Chemically treat** the DNA to be reinforced to result in longer reads or by improving accuracy [11].

2.2.2. Sequencing DNA

Nanopore sequencers are long-read sequencers designed by Oxford Nanopore Technologies. While there is a lot of chemistry and physics involved in these sequencing techniques, a nanopore is essentially a hole where the DNA can be pulled through, see Figure 2.1. The DNA travels to the nanopore by means of the motor protein attached after sampling, when it arrives at the nanopore the DNA is caught and split from double-strand DNA (dsDNA) to single-strand DNA (ssDNA), after which one of the split strands is to be pulled through the pore [13]. The nanopore is pulled through the pore with speeds of around 260 base pairs per second (bps) or for bigger sequencers 400 bps. At the same time, a sensor attached on both sides of the nanopore measures the bases at the same sampling speed as the DNA goes through the pore, so 260 and 400 samples per second. This sensor runs a current through the DNA and by measuring the corresponding impedance of the DNA, this impedance can be converted into an analog signal as seen in Figure 2.1 [12]. Matching the rates of sampling and pulling remains a challenge as it leads to *"blurry"* measurements. The nanopores can be arranged in a 2-dimensional array allowing the nanopore sequencing process to be greatly parallelizable, greatly improving throughput.

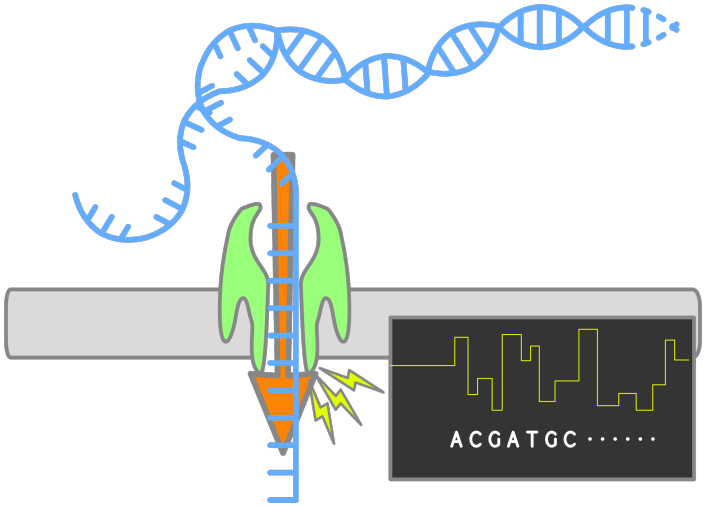


Figure 2.1: Visualization of DNA going through a nanopore and measured to an analog signal [14]

2.2.3. Basecalling

After the DNA has been sequenced, the raw sequenced data have to be basecalled. The output of the sequencer is an analog signal, which is not useful to analyze for DNA variations, etc. To make the sequenced data useful, it has to be converted to base pairs (A, C, T, G), as seen sketched in Figure 2.1, this process is called basecalling [13].

Compared to short-read sequencing, long-read basecalling is a much more challenging process. As mentioned before, this challenge mainly lies in the speed at which the DNA strain is pulled through the nanopore during sequencing. If the speed at the strain goes through the port is too high the signal of individual bases may influence the neighboring base pairs, conversely, if the speed is slow a stuttering signal is induced making it apparent as if there are multiple repeating bases even though there is only one. Therefore, this varying speed introduces non-deterministic results out of the sequencer requiring also probabilistic models and algorithms to process it [15].

Currently, this basecalling is the most computationally heavy process in the pipeline and is thus the slowest stage currently in the pipeline [2][8]. While it is very dependent on the hardware used to perform basecalling and not a lot of studies can be found on speed analysis, the time to basecall can lay between 5 to 10 hours which is significantly slower than sequencing [2][8]. There are efforts to perform real-time synchronous basecalling and sequencing, however, this will currently result in a loss of accuracy or using a lot more resources to keep up with the throughput of the sequencer [13]. For instance, the fully saturated nanopores on an Oxford Nanopore Promethion 48 (the largest nanopore sequencer) require

around 4 A100 GPUs using lower accuracy and 20 in the highest accuracy configuration [16][17]. This is very expensive and power consuming, totaling as of the time of writing at $60,000 to $300,000 in cost and 1200W to 6000W in power consumption [18], while the Promethion itself costs around $600 [2].

2.2.4. Post-sequencing analysis

The following sections discuss a small sample of the main and most well-proven post-sequence analysis applications.

Read alignment and variant analysis

Variant calling is the process of finding variations by comparing the sequenced genome with a database of genomes. This is one of the primary applications of genome analysis [2].

However before the variations can be detected between samples and the database, the sample reads have to be aligned to the database reads, this process is called read alignment. In the initial stages of long-read analysis, this alignment used very complex and sensitive algorithms. However, the current reads are accurate enough that the current alignment process does not require complex algorithms and the process can be achieved relatively quickly and is most of the time baked-in with basecallers as a post-processing step [8]. Additionally, dynamic programming algorithms are introduced that allow base-by-base alignment [2].

There are a couple of main variations that can occur between the sample and database, these are isolated single nucleotide variations, insertions, deletions, and shortened and extended DNA repeat regions [19]. The process of variant calling has been improved significantly by using long-reads over short-reads as the longer-reads allow better detection of repetition length as those most often spanned over multiple short-reads, which makes the alignment with the reference genome unreliable and not feasible for short-read sequencing [2].

Traditionally variant calling has always been performed by comparing samples to databases, newer implementations also deploy deep-neural networks to detect variations which makes it connectionless to a database [20], which allows also the possibility of creating more portable devices. Current implementations have a runtime for variant calling ranging from 1 to 2 hours [8].

De novo genome assembly

De novo genome assembly is the process of reassembling the sequenced genome. This process is as mentioned before made easier by the long-read sequencing, as there are fewer reads necessary to reassemble the genome. The most applied technique to assemble genomes is by utilizing string graphs where entire reads are compared to each other using minimizers and MinHash techniques [2].

However, the challenges in this pipeline stage are not algorithmic but again depend more on the accuracy of the sequencer and basecalling process. Currently, most of the time in de novo assembly is spent on correcting the reads from the sequencer [21]. The process of assembling a telomere to telomere (begin to end of the chromosomes) genome error-free has only recently become possible by utilizing every possible new technique to improve read accuracies [2].

2.3. Basecalling challenges

Considering the discussed long-read pipeline, it seems that the most interesting part is the basecalling stage, due to its time bottleneck and accuracy implications. DNA preparation and sequencing clearly fall out of the domain of Computer Engineering, and thus pose little potential. However, compared to the post-sequencing analysis methods it seems that the main challenges still lay in the low accuracy of the basecalling and sequencing process, so by further improving the basecalling process a lot of post-sequencing analysis methods will benefit too. Additionally, this leads to another point that basecalling is the core stage of all long-read analysis, as all subsequent tools require a basecalled sequence, not the analog signal. Lastly, the basecalling process is currently very slow, it takes up a large chunk of the entire long-read pipeline. Therefore improving this part of the process by accelerating will be a great benefit for a lot of users of these pipelines.

3

Exploring basecallers

This chapter will describe the exploration of basecallers. It will look at the state-of-the-art techniques used to perform basecalling, compare several basecalling solutions to each other, and end by selecting the most suitable basecaller to consider for acceleration. This selected basecalling solution will in turn be analyzed further to examine its bottlenecks and strengths. Finally, there will be looked at suitable methods to implement this selected basecaller on an FPGA.

3.1. State-of-the-art of basecalling

While discussing the pipeline, it was already mentioned that the process of basecalling for Oxford Nanopore sequencers is not a deterministic process. This is due to the fact that the sample interval and the speed that the DNA strain goes through are not always synchronous, resulting in base pair readings blurring other base pairs and stuttering resulting in false repetitions [15]. For this reason, the state-of-the-art implementations are currently exclusively built on stochastic or probabilistic models, being neural networks or hidden Markov models [2].

While numerous basecallers have been made in the past decade, only a few will be selected to be highlighted here to demonstrate and compare different architectures of neural networks. The current space of basecallers can be divided into 4 categories:

- Neural network implementations
  - Recurrent neural networks
  - Convolutional neural networks
  - Transformers
- Other implementations
  - Hidden Markov models

3.1.1. Recurrent neural networks

A recurrent neural network (RNN) is an artificial neural network specifically designed to handle sequential or time series data. These deep learning algorithms are extensively employed for solving ordinal or temporal problems, including language translation, natural language processing (NLP), speech recognition, and image captioning [22]. The problem of basecalling can be closely mimicked to a speech recognition problem. Converting an analog signal to a sequence of speech-to-words for speech recognition is very similar to converting the analog signal from the nanopore sensor to bases (letters) A, C, T, G during sequencing [23].

What sets RNNs apart is their inherent ”memory” capability, enabling them to leverage information from preceding inputs to impact the current input and output. In contrast to traditional deep neural networks that treat input and output as independent entities, recurrent neural networks rely on the previous elements within the sequence to determine their output [22].

Oxford Nanopore, in its initial stages, introduced three distinct models for their sequencing technology: Albacore, Guppy, and Scrappie. Among these models, Albacore was designed as the earliest (2017) and used a CPU implementation, while later (2018) Guppy and Scrappie were developed by using GPU implementations. Guppy was the most accurate while also being a stable model at the time, while Scrappie was more experimental and served as a testing ground for new features and enhancements that were being considered for integration into Guppy [24].

These implementations by Oxford Nanopore first used only a single layer of RNNs, which resulted in still very low read accuracy of around 88% [25]. This accuracy is too low to perform any useful post-sequencing analysis [2]. Around the same time Oxford Nanopore was developing its GPU implementations, another basecalling architecture was developed by Haotian Teng et al. called Chiron [26]. This architecture modeled a new architecture by introducing newer developments from the research of speech-to-text applications. Chiron combined deep learning with RNNs; this meant that the raw nanopore signal was first fed into convolutional neural network (CNN) layers before going into the RNN layers. By utilizing a CNN, the model extracts useful features of the raw signal instead of feeding just the signal into the RNN directly. By training the model, the CNN can be trained to extract features of the raw signal that provide better information to the RNN layer. Haotian Teng et al. state that using both CNN layers and RNN layers is crucial in basecalling networks otherwise accuracy will drop significantly [26].

Additionally, Chiron introduced a statistical post-processing layer called connectionist temporal classification (CTC) decoder to get to the final basecalled sequence output. The CTC decoder solves the variable speed problem of the DNA strain going through the nanopore better. Essentially what this layer adds is that a new label is added to the existing A, C, T, and G labels; this label is a null label that allows the RNN to output nothing if it is most likely that the input signal stuttered. With this new label, the CTC decoder predicts the most likely sequence given the output of the RNNs, however, it also takes previously predicted sequences into account [27]. While the accuracy increased, the model was however very slow; 100x slower than the Oxford Nanopore implementations [25]. However, Oxford Nanopore developed another model called Flappie based on Guppy that used the CTC decoder without the CNN layers [25]. Next to this, Flappie introduced a *"flip-flop"* mechanism which means that the output from a layer is reversed when it is put in the consecutive layer [28]. While the reason is never stated in literature why the exact reason for this *"flip-flop"* architecture is developed, it is most likely that this is due to the measurement of the sensor being bi-directional. This means that the DNA strain could have passed through the nanopore in either direction causing the sensor reading to be reversed. It is likely that continuously flipping the outputs between layers increases accuracy as it is analyzed from both directions. This model eventually replaced Scrappie in being the experimental line and Scrappie fell out of development [29]. A year into development, Flappie was also archived suggesting that it is most likely that it has been incorporated into Guppy [13][25], however due to Guppy being proprietary software it is impossible to be certain.

In 2020, Oxford Nanopore made an open-source version of a basecaller called Bonito/Dorado. Bonito and Dorado are the same model, the only difference is that Bonito is written in Python using PyTorch while Dorado is written in C++ [30][31]. For ease of reading only Bonito will be mentioned, however, most comments will apply to both and it will be specified when something is different for Dorado. While there is little to no literature attached to these models, by examining the commit history of the code repositories the model's development has been very analogous to the development mentioned earlier with Guppy and its experimental models (Flappie, Scrappie) [30]. However, the current implementation of Bonito has shifted in 2022 to a model very similar to that of Chiron, although the reason for this design shift is not documented it is clear that it is a good architecture as currently, Bonito is the most accurate basecalling model to date by achieving a read accuracy of 93% [32]. However, there are changes compared to the Chiron CNN-RNN-CTC architecture: the *"flip-flop"* mechanism from Flappie is introduced and the CTC layer has been changed to a conditional random field (CRF) decoder as a post-processing layer [30]. Instead of the CTC predicting the most likely sequence given the previous prediction, the CRF decoder is more granular predicting the most likely base given its neighboring bases [33]. This CRF is considered superior over CTC from benchmarking improving performance over 4% [32]. Altogether, Bonito is currently the leading basecalling tool available having little competition, Bonito's model is speculated to be copied over into Guppy [32]. It has high accuracy but is still considered slow [34], however, its higher accuracy enabled better post-sequencing analysis leading it to be the most popular [30].

3.1.2. Convolutional neural networks

Other models try to perform basecalling by removing the RNN from the models and creating full CNN models. There are three prominent models developed, CasaulCall, URnano, and MinCall, that use full CNN networks which arose around 2020 [32]. The main problem these models tried to solve was that the RNN models were slow [35][36][37]. The main bottleneck lies in the recurrent nature of RNNs, meaning that the computation of the subsequent inference is dependent on the previous inference making unrolling and parallelization complex [35]. CNNs on the contrary are very parellizable making them in terms of speed very competitive, but they compromise on accuracy with MinCall having a very low 63% read accuracy, whilst CasaulCall and URnano having 88% and 90% accuracy respectively [32]. The read accuracies of URnano and CasaulCall were competitive at the time with URnano being on par with Guppy at the time [37].

CasaulCall and MinCall are very similar to the Bonito implementations with the RNN layers stripped away, the architectures are roughly a CNN-CTC network [35] [36]. URnano uses a slightly different approach instead of having the neural network outputting the sequence of bases, the neural network outputs a base mask. This base mask is fed into the label transformer algorithm which performs a per base conversion instead on a per sequence basis with CTC [37]. As seen with the CRF which also uses a more per-base approach, this approach leads apparently to better accuracy compared to CTC [32].

Currently, none of these three networks are in active use nor development, making the analysis from Haotian Teng et al. quite plausible that basecaller neural networks require both CNNs and RNNs [26]. The most likely reason for these models to be abandoned is that the accuracy is too low compared to RNN networks and in the case of URnano being actually 2x-3x slower than Guppy in 2020 [37].

3.1.3. Transformers

A transformer model is a type of neural network that acquires contextual understanding and meaning by analyzing the relationships among sequential data elements, such as the words in a sentence. Utilizing a set of mathematical techniques known as attention or self-attention, transformer models can discern subtle connections between remotely positioned elements within a sequence and can identify how they influence and rely on each other. These sophisticated algorithms enable the transformer to grasp long-range dependencies in the data, leading to more accurate contextual comprehension [38].

By this description, transformers seem quite similar to RNNs, however, solve some problems attached to RNNs. RNNs are trained sequentially, this leads to two problems that transformers do not suffer from. First, sequential training means that parallelizing the inference is more difficult for RNNs. Another problem in RNNs is exploding and vanishing gradients, due to the recurrent properties of RNNs training results in feedback loops. These feedback loops make it difficult to train RNNs as it is difficult to keep the gradients bounded as the gradients tend to explode to infinity or vanish to zero over time [39]. Due to the attention algorithms, transformers do not have a recurrent structure meaning they are highly parallelizable [40]. Moreover, this means that the aforementioned problems of RNNs do not apply to transformers [41]. Another benefit of transformers is that pre-trained models can easily be used for other tasks, resulting in less training time [41].

Currently, there are two prominent transformer models for basecalling SACall and CATCaller [32]. Both these models are based on a CNN-transformer-CTC model [42][43]. While SAcall is based on the transformer model, CATCaller is based on a lite-transformer model [32]. This lite-transformer is different in that the convolutions, called lightweight convolutions, in the transformer layer are sharing weights. These lightweight convolutions reduce the model complexity allowing lite-transformer models to be run on edge devices [44]. While these models claimed to be designed for speed, SACall is not faster than Guppy [42] and CATCaller claims to be 4x faster than SACall it does not make any speed comparison to Guppy [43] making comparisons very difficult. Accuracy-wise the transformer models seem to be not competitive. As from the benchmark result of Marc Pagès-Gallego et al., RNNs are superior to transformer models [32]. Since the transformer models are still using CTC decoders, these results might be more competitive if they use a CRF decoder instead as the current RNN basecallers.

3.1.4. Hidden Markov models

Hidden Markov Models (HMM) are statistical models that relate a sequence of observations to a sequence of hidden states. It is useful for predicting future observations or classifying sequences based

on the hidden process generating the data [45]. While a lot of literature claims that hidden Markov models were used as the first basecallers for long read analysis [2][25][26][32], it seems to be difficult to find any HMM implementations for basecallers. One HMM basecaller could be found called Nanocall [46]. The model is trained using Expectation Maximization (EM) which is very standard for HMMs. At the end of the HMM inference, Nanocall applies a Viterbi decoder to get the output. The Viterbi decoder is similar to a CTC decoder, however, it only considers the current sequence to predict the sequence output instead of including neighboring sequences [46]. The inference of HMMs and thus Nanocall are very lightweight computations compared to neural networks.

It seems however that hidden Markov models have completely fallen out of favor and current implementations have completely shifted towards neural networks. While hidden Markov models are easier to accelerate, the accuracy in basecalling with neural networks is much higher for neural networks than for hidden Markov models [2].

3.1.5. Choosing a model

After comparing and discussing different models and architectures, it was decided to choose the Bonito model to accelerate on an FPGA to demonstrate Fletcher and Tydi, two technologies developed by the Accelerated Big Data Systems (ABS) group from TU Delft. Fletcher is a framework that helps to integrate FPGA accelerators that use an Apache Arrow back-end [47]. Bonito switched over at the end of 2022 to the POD5 file format from the older FAST5 file format [30]. The POD5 file format stores data in Apache Arrow tables, this means that data is stored in a columnar format allowing for more efficient data readouts for acceleration purposes [48]. POD5 using Apache Arrow means that more tools are available, like Fletcher, as Apache Arrow is general-purpose and popular while FAST5 is just designed for Nanopore sequencing. Bonito is currently the only major basecaller using POD5 making it ideal to implement with Fletcher.

Next to the POD5 argument, Bonito is currently the most accurate basecalling solution while also relatively being speed competitive [32]. Bonito is also the most widely used next to Guppy but Bonito is an open-source basecaller and Guppy is proprietary making it impossible to know how Guppy is implemented exactly. Additionally, all the other basecallers from literature have no published source code or have been unmaintained for several years making it not compelling to create an accelerator from them. Additionally, full CNN models are not very suitable to accelerate on FPGAs as it is hard to beat GPUs with CNNs since GPUs have dedicated hardware for tensor calculus.

3.2. Deep-dive into Bonito

Since Bonito is the chosen model to accelerate, this section will dive deeper into the neural network components that Bonito consists of and benchmark the model to identify the most computationally heavy components of the network. After this analysis, multiple implementation options will be explored to decide the best way to implement the Bonito on an FPGA.

3.2.1. Analyzing Bonito

As mentioned before, Bonito is a model by Oxford Nanopore and it uses a CNN-RNN-CRF model. However, this classifier CNN-RNN-CRF does not describe the model completely as there are many more granular components present in the model. Moreover, Oxford Nanopore does not provide any literature or documentation about the design decision or how the model works, therefore, the model is *"reverse-engineered* in this section to provide such documentation. By delving into the config files of the models, it is possible to construct an overview of the model, as seen in Figure 3.2 [30]. There are three models provided by Bonito: Fast, High Accuracy, and Super Accuracy (presumed from the abbreviation SUP). The Fast and High-accuracy models can be used for real-time basecalling depending on the amounts of nanopores present in the sequencers (sequencer throughput) and the computational throughput of the basecaller. The super model is generally used in a catch-up mode meaning the output of the sequencer is buffered [13]. Moreover, there are two models for each accuracy model for a sequencing speed of 260 base pairs per second (bps) and 400 (bps), these models do not differ from each other architecturally, the only difference is that the weights are different.

A convolutional block consists of multiple subfunctions, as seen in Figure 3.1. While the most computationally heavy and important part is the convolution, a batch normalization, and a swish activation

function are performed after the convolution. The batch normalization is used to speed up training, it allows for faster training speeds by normalizing the output from the convolution to network-wide normalization making it not too high or too low to be processed by subsequent layers [39]. The swish activation function is not standard but is more recently becoming popular over the more standardized ReLU activation function [49]. Activation functions are often simple but very crucial to allow for non-linearity in the neural network as all other components like convolution, RNN, etc. are linear equations [39]. Swish is a type of sigmoid-weighted linear unit (SiLU) activation that looks like $Swish(x) = x \cdot \sigma(\beta x)$ where β is a trainable parameter which allows for tweaking the linearity of the activation function; if $\beta \to 0$ the function becomes linear by the increasing β the sigmoid function will be more pronounced. The convolutions are all 1-dimensional convolutions with a convolution kernel of size 16. As seen in Figure 3.2, the convolution sizes the input up to the size of the RNNs, starting from 1 and ending up at 96, 384, or 1024 depending on the model size.

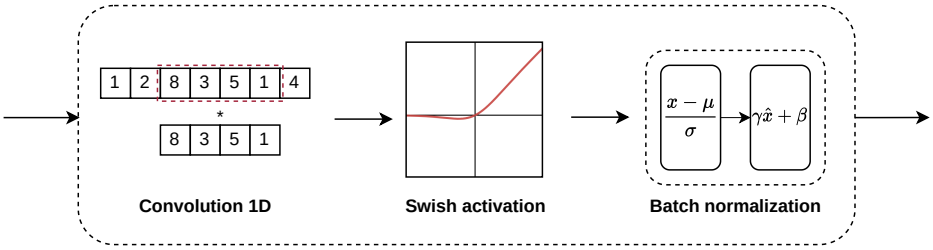


Figure 3.1: Architecture of convolutional block in the Bonito models

The RNNs in the models are called LSTMs (long-short-term memory), LSTM is a variant of RNN as traditional RNNs are barely used anymore. Traditional RNNs suffer a lot from the vanishing gradient problem because they only keep track of short sequences and forget older inputs; this is called short-term memory. LSTMs add an additional long-term memory which allows older inputs to affect the current data thus meaning that these older gradients do not vanish. The long-short term memory allows LSTMs to analyze data on a granular scale but still can keep correlation between distant data [39]. The LSTMs have a size of 98 in the fast model, 384 in the high accuracy model, and 1024 in the super model; these sizes do not change between layers. As mentioned before with Flappie, Bonito also uses the *"flip-flop"* mechanism which means that the output will be reversed between layers.

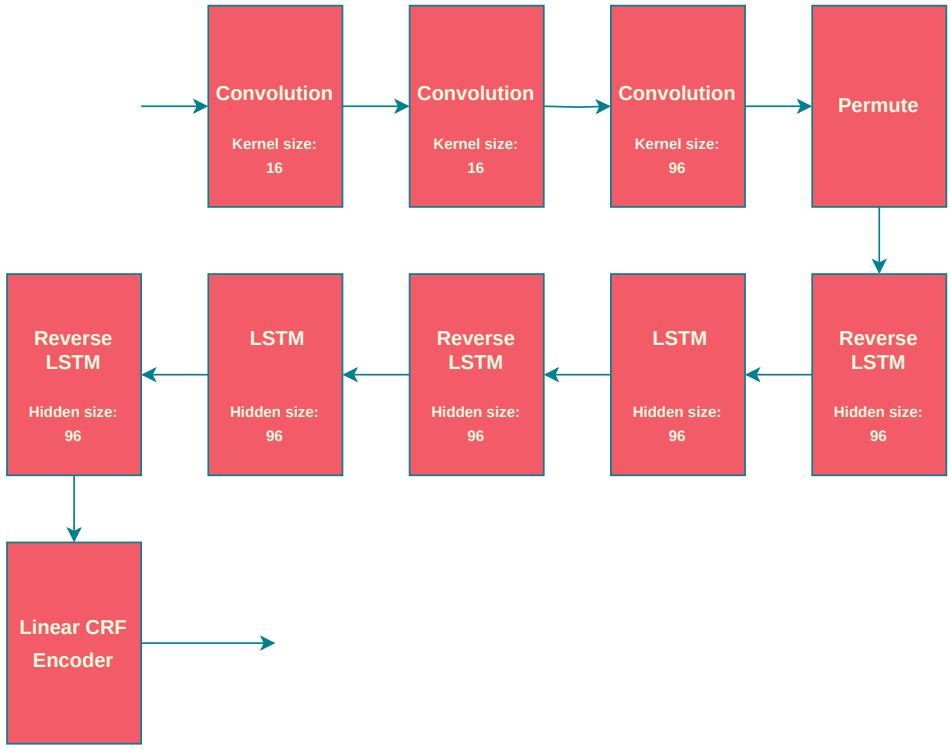
After the LSTMs, the output is again reduced using a linear layer, also called a fully connected layer (FC). The reason for this fully connected layer is not clearly known due to the lack of documentation. Normally, FC layers let all inputs interact with each other to create more interdependency between the output of the previous layer, and the output is reduced back to the training labels. However, here the output goes into another linear layer, the linear CRF Encoder, which consists of an FC layer and a CRF encoder. The FC layer scales up the data so that the CRF encoder can output scores in a linear-chain CRF. This linear-chain CRF can be used to calculate the log probability of a particular (aligned) output sequence [50]. As a guess the first linear layer is to classify the output of the LSTMs for the CRF decoder as the first stage of the linear CRF decoder is also an identical linear layer, hence it does not make sense that it is to reduce data size for the CRF decoder.

Additionally, there are some small layers like clamp and permute that are not very computationally heavy. The clamp layer clamps the output between boundary values so that the output is not too large or too small for the next layers. These boundary values are defined in the model config and thus not trained. The permute layer turns the input tensor around axes, it is not very clear from documentation or literature why one would apply this layer, however.

3.2.2. Benchmarking Bonito

The amount of parameters seems to be heavily dependent on the size of the LSTMs used. The difference between models is only significant in the sizes of the LSTM. The super model has 7× more parameters than the high accuracy model which in turn has 14× more parameters than the fast model. To further attest to this claim, the model has been benchmarked. The benchmark was run on a server with an *Intel Xeon E5-2620* CPU and a *NVIDIA RTX 2080Ti* GPU. Figure 3.3 shows the trace of the

Fast: 427,984 parameters 260/400bps



HAC (High accuracy): 6,213,296 parameters 260/400bps

SUP (Super): 43,610,800 parameters 260/400bps

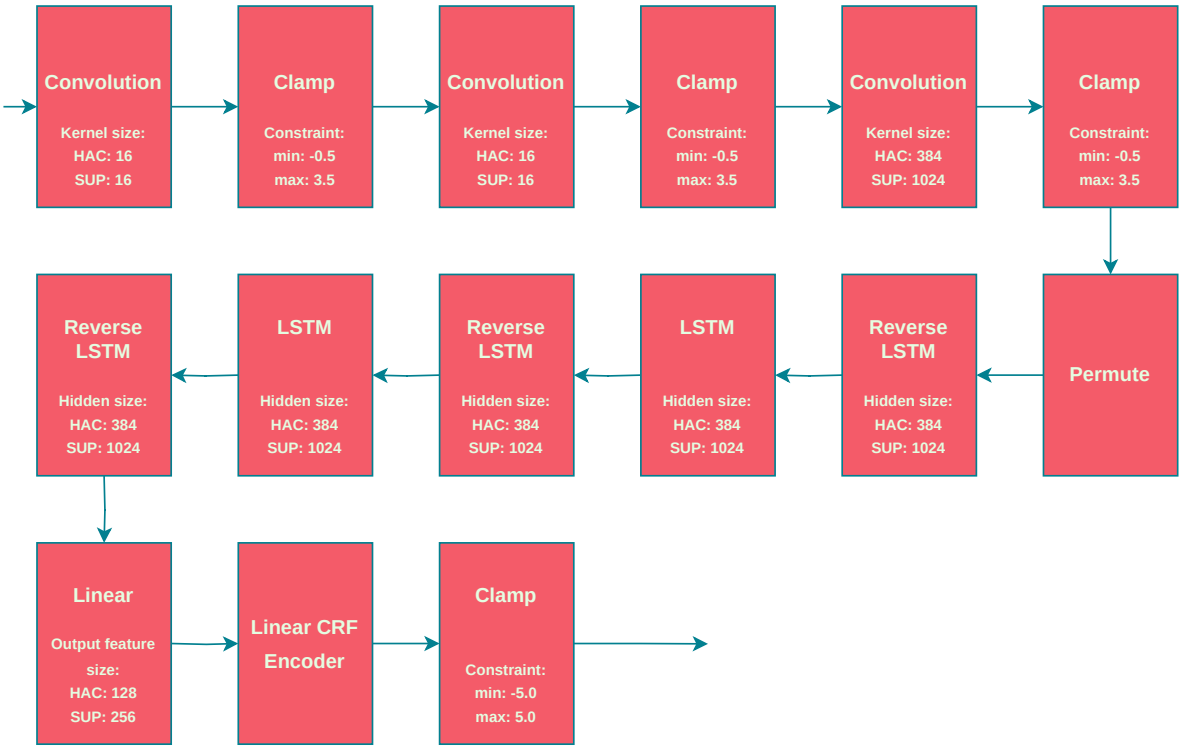


Figure 3.2: Overview of bonito models

runtime of a single inference through the Bonito model. The trace clearly shows that nearly all the processing time of the inference is spent on the RNN calculations, demonstrating clearly that the LSTM is the bottleneck in the calculations.

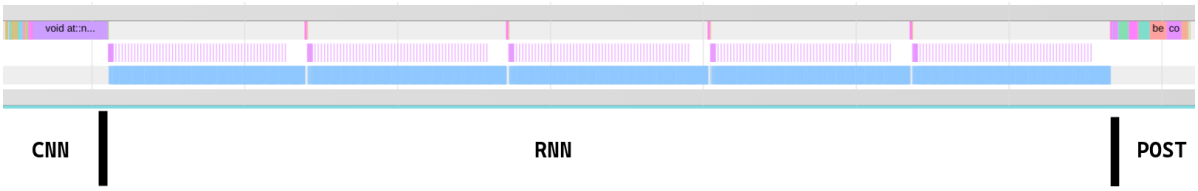


Figure 3.3: Trace of an inference of the Bonito network

Next to the trace, it is also interesting to see how the models compare to each other. The result of this benchmark can be seen in Table 3.1. Dorado has been chosen to be benchmarked as well and clearly from the table it is apparent that it is much faster than Bonito. This speed difference is not surprising, however, as mentioned by the maintainers of both Dorado is for practical use while Bonito is more for training purposes and model development and analysis [30][31]. This is mainly because of Bonito using Python which is easier to develop for and experiment with than C++ used by Dorado but Python being an interpreted language means it induces more overhead. The models between Dorado and Bonito are as said before equivalent. The super model is also significantly slower than the other models which scale not linearly with the number of model parameters. This indicates that computationally the throughput decreases faster for larger LSTM sizes.

Table 3.1: Performance comparison of Bonito and Dorado

	Bonito		**Dorado**	
	Duration (H:M:S)	**Reads per second**	**Duration (H:M:S)**	**Reads per second**
Fast model	00:24:53	53.6	00:03:09	424.5
High accuracy model	00:33:21	40.0	00:17:31	76.2
Super model	03:00:41	7.4	01:46:32	12.6

3.2.3. Implementing Bonito

There are multiple avenues to explore to implement Bonito from scratch. The first possibility is to build the entire model from scratch in VHDL or Vivado High-Level Synthesis (HLS). While this would give the most control over the project's implementation, it would also be a lot of work and require a lot of research in each layer to understand how to implement it while also making it performant. AMD has however introduced over the years a lot of frameworks and tools to use to more easily implement neural networks on FPGAs, an example being FINN [51]. FINN is a tool that allows generating dataflow-style architectures on an FPGA based on pre-trained neural networks. While that description fits nicely with Bonito being a pre-trained model, there are some caveats as some layers are not supported mainly being RNNs. Since the core of the models are RNNs, this means that this makes FINN not feasible. Moreover, FINN does not support the swish activation function and CRF layer, however, this is much less of a concern since these are not very complicated layers; the CRF layer is essentially an FC layer, which is supported, with a quite lightweight scoring algorithm after it.

Next to FINN, there is still another tool available to help implement it called Vitis-AI, also by AMD. Vitis-AI is more of a blanket term for a collection of AI IPs, libraries, tools, and models. The main workflow of Vitis-AI is using a pre-built model from the so-called *"Model Zoo"* provided by AMD. This model can be adjusted to be more in line with the model to be implemented but this adjusting is very limited, so models may need to be nearly one-to-one and can differ only in size for some layers [52]. After selecting a good model implementation is quite easy, as the models from Model Zoo have already been deployed and tested on FPGAs. While Vitis AI states that RNNs are supported, all links leading to documentation and models on GitHub are broken. It seems that the RNNs from Vitis AI have been

dropped since going to version 3 in 2022. This means that using Vitis AI and Model Zoo is not possible to implement Bonito.

Before going all the way down to low-level hardware design VHDL/HLS, a last attempt was explored to use a higher-level approach by combining Cλash and HaskTorch. Cλash is a high-level synthesis language like Vivado HLS but instead of HLS using C++, it uses Haskell. Haskell in turn has a Torch implementation called HaskTorch which makes it easy to run Bonito in Haskell as Bonito is built using PyTorch which is a Torch implementation for Python. While running the model is possible, it seems to be very hard to make HaskTorch and Cλash work together as HaskTorch is designed really for CPU and GPU implementations. After a lot of trying this solution seemed to be also a dead end.

A final possibility without going to a from-scratch implementation is looking into the AMD VCK5000 Versal FPGA. Versal is an FPGA card specifically designed for neural network acceleration by offering special hardware blocks on the FPGA for neural networks. Access to this FPGA has been granted by the *Heterogeneous Accelerated Compute Clusters* at ETH Zürich to use for this project. While on the product it specifies that it can be used for CNNs and RNNs [53], if one looks deeper into the documentation there is no trace to be found about RNNs. The documentation only specifies how to use the special hardware blocks for CNNs and those blocks seem to only have CNN capabilities meaning that utilizing this FPGA for RNNs would be equivalent to any other FPGA of equal resource capability.

It seems now that every high-level implementation gets stuck on not being able to implement LSTMs, it seems to be important to explore why this limitation exists before starting a from-scratch implementation. While the lack of RNN support in models can be due to the higher popularity of full CNN networks, there might also be more reasons for there being such low support. After research, it seems that the internal storage (SRAM) of the FPGA seems to be the main bottleneck on FPGAs. This means that a lot of the time LSTM implementation needs to stream from DRAM which is significantly slower [54][55]. Therefore, it is important to find out the means to make this memory management work on an FPGA, but for that first, a better understanding must be acquired to find out why LSTMs are more memory intensive than i.e. CNNs.

This also marks a turning point in the thesis that the focus shifts away from creating a demonstrator using Fletcher and Tydi to implementing the LSTM part of Bonito. The decision was made to only focus on the LSTM because clearly there is a bottleneck that potentially can be solved but also the LSTM has a lot less research compared to other layers i.e. CNN, FC, etc. CNNs for instance can be easily generated so by dropping the Fletcher and Tydi part not a lot of interesting research potentials remain there.

4

LSTM accelerator design

This chapter will examine the entire design process of implementing the LSTM layers of Bonito on an FPGA. First, the LSTM layer will be analyzed theoretically to understand the workings of LSTMs. After that, an FPGA will be selected for use in implementation based on the requirements imposed by the LSTM layer. Before going into the low-level implementation of the hardware components, first an abstract overview is shown of the design system architecture and the components will be abstractly outlined. After the overview, each component will be described in more detail including design challenges and iterations.

4.1. Background of LSTMs

Long-term short-term memory (LSTM) is a variant of RNN that can overcome the vanishing gradient problem. Instead of an RNN only having short-term memory hidden layer (h_t), LSTMs additionally add a long-term memory called cell state (c_t) to the neuron. The long term memory is used to store information over longer time steps. The hidden layer h_t and the cell state c_t are the recursive components of the LSTM meaning that at the next inference, they are reinserted into the LSTM cell; see Figure 4.1. The long-term memory is updated by a function of a forget gate f_t and an input gate i_t. The forget gate decides which information from the previous long-term memory should be ignored and which should be kept. The input gate combined with the cell input gate $\tilde{c}_t$ determines which information from the current input should be added to the long-term memory. The cell input gate $\tilde{c}_t$ is often denoted in code as g_t due to the tilde and confusion with the cell state c_t. The hidden layer h_t, the short-term memory, is a function of the long-term memory c_t and an output gate o_t. The output gate decides how much the short-term memory and input information should be outputted. LSTM uses both sigmoid and hyperbolic tangent activation functions [56].

The input x_t, hidden layer h_t, and cell state are vectors. The hidden layer and the cell state vectors have to be always equal in size and they parameterize the output size of the LSTM. The input vector can differ from the hidden layer and the cell state vectors denoting the input size. Whilst Figure 4.1 only shows the activation function σ or $tanh$, the gates' decision-making cannot only be performed by these functions as they are not parameterizable, meaning they cannot be altered and trained. Therefore, before the activation each gate performs an inner product between a weight vector stored in the gate and the input and hidden vector. After the inner product optionally a bias vector can be added to the product result [56].

Whilst the inputs of the cells are vectors, the cells themselves produce a scalar value. Therefore, there have to be as many cells equal to the output size of the LSTM to obtain the full output vectors. By combining all those cells together, the inner products of the gates of each cell combined can be modeled as a matrix-vector multiplication for each gate. The resulting gate equations can be seen in Equation 4.1 to 4.4, which shows that there are 8 matrix-vector multiplications; 4 multiplications between the input vector and input weights and 4 between the hidden vector and hidden weights. These weight matrices have dimensions of $hidden\_size \times input\_size$ and $hidden\_size \times hidden\_size$ for the input weights and hidden weights respectively. Subsequently, the gates use a hyperbolic tangent or a sigmoid (Equation 4.7) to bind the vector output of the matrix-vector multiplication to the interval

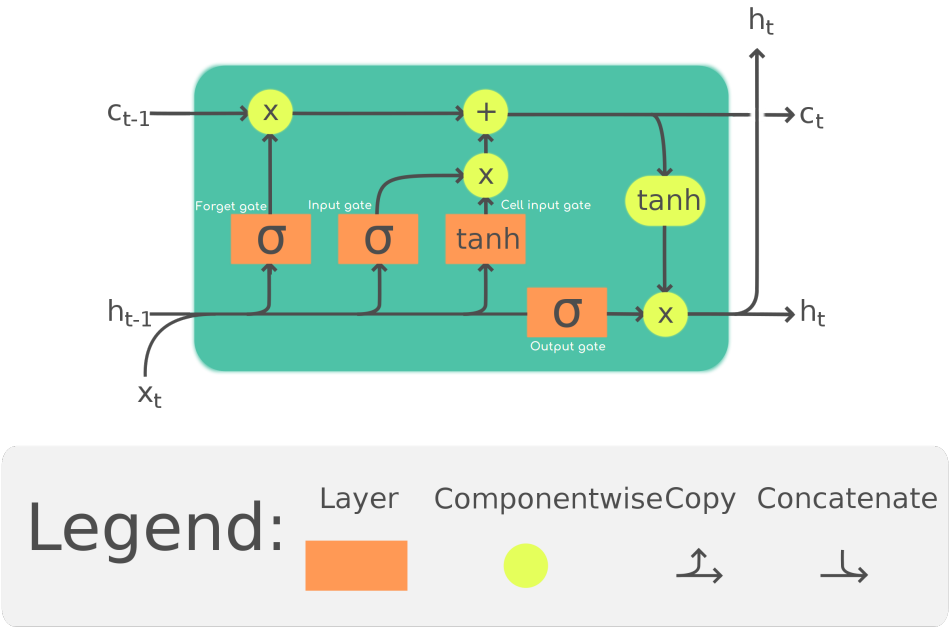


Figure 4.1: Visualization of an LSTM cell [57]

$(-1, 1)$ and $(0, 1)$ respectively. Equation 4.5 and 4.6 show the equations to get the final short-term memory h_t (output) and long-term memory c_t. These equations use Hadamard products ($\odot$) which is an element-wise product between the gate vectors. This is because, in a cell after the gate operations, all operations are using local values of the cell therefore when combining multiple cells this results in a Hadamard product [56].

$$i_t = \sigma(W_{i\_input}x_t + W_{i\_hidden}h_{t-1} + b_i) \tag{4.1}$$

$$f_t = \sigma(W_{f\_input}x_t + W_{f\_hidden}h_{t-1} + b_f) \tag{4.2}$$

$$\tilde{c}_t = tanh(W_{\tilde{c}\_input}x_t + W_{\tilde{c}\_hidden}h_{t-1} + b_{\tilde{c}}) \tag{4.3}$$

$$o_t = \sigma(W_{o\_input}x_t + W_{o\_hidden}h_{t-1} + b_o) \tag{4.4}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{4.5}$$

$$h_t = o_t \odot tanh(c_t) \tag{4.6}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.7}$$

These equations above demonstrate well why this layer is significantly more compute-intensive compared to a convolutional layer. By analyzing the equations by counting atomic operations, where the activation functions are considered atomic but the matrix-vector multiplications are split into a multiplication and an addition, it is possible to see how intensive it is. The number of operations in a matrix-vector multiplication is $n(2m-1)$, for the Bonito model the input size is equal to the hidden/output size, therefore, this simplifies to $2n^2 - n$ where n is the layer size, scaling this up to 8 the matrix-vector multiplications this results in $16n^2 - 8n$ cycles for the 8 matrix-vector multiplications. After that, there are per cell 4 bias additions, 5 activation functions, 3 multiplications (Hadamard product), and 1 addition; since these are per cell these operations are all equal to n resulting in $13n$. Combining the results each inference has $16n^2 + 5n$ operations, meaning that there are 16.9 million atomic operations for each inference in the Bonito super model where the layer size is 1024.

Memory requirements and FPGA selection

In terms of FPGA implementation, the most important thing to validate before implementing anything is if these weights, biases, etc. can be stored and accessed timely at all in FPGAs. As seen from the theoretical analysis of the LSTM layer, the memory requirements are quite demanding, especially for FPGAs where on-chip storage is scarce, of around 50MB on a high-end FPGA (*AMD Alveo U250*) [58]; compare that to a high-end CPU having 290MB (*AMD Ryzen Threadripper PRO 5995WX*) [59] and a high-end GPU having 70MB (*NVIDIA A100*) [60]. However, off-chip storage on FPGA is often plenty, but the bandwidth is in turn limited on the off-chip memory.

For an LSTM to function, there are ten different data stores required: an input data store, a hidden/output data store, and eight weight data stores for each gate. The hidden and input data sizes are equal since in the Bonito model the input and hidden vector dimensions are the same. Using the knowledge of the LSTM and Bonito, the memory requirements per LSTM layer can be calculated as seen in Table 4.1. The input/hidden storage size seems to be very high for a single vector input and output, as for a single 1024 vector the storage should only be 2KB. However, the table value is different as most often in neural networks the input is batched into multiple vectors making the input essentially a matrix. In Bonito, this batch is additionally split into chunks. This chunking is not a simple further subdivision of the reads from the sequencer, this is because, with the chunks, the neighboring chunks overlap partially with each other. Due to the lack of documentation on Bonito, the exact reason for this overlap is not clear. Bonito has for all its models specified a chunk size of 1.000 with an overlap of 50. This chunk size is used in Table 4.1 to determine the memory requirements of the input and hidden storage, the batch size would multiply this number even further however the batch size numbers lay above 100 which would move up the memory requirements over 100MB of on-chip memory which is practically impossible on an FPGA at the time of writing.

Table 4.1: Memory requirements of a Bonito LSTM layer (16 bits)

Model	Input/Hidden Size	Input & Hidden Storage (MB)	Weight Storage (MB)
Super model	1024	4.0	16.8
HAC model	384	1.5	2.4
Fast Model	98	0.38	0.15

With the memory requirement available, selecting an FPGA to implement for is possible. After researching, it seems that there are two main FPGA contenders for the implementation, those being: the *AMD Alveo U280* and the *AMD VCK5000 Versal*. Table 4.2 shows the corresponding memory specification of these FPGAs. The reason for them being the main contenders is that the Alveo U280 is the FPGA with the most internal memory available at the *Quantum & Computer Engineering Cluster* at TU Delft while the Versal as mentioned before is available via the *Heterogeneous Accelerated Compute Clusters* at ETH Zürich which enables easier implementation of the other layers of Bonito for future work.

Table 4.2: Memory specification of the AMD Alveo U280 [61] and AMD VCK5000 Versal [53]

	Alveo U280	VCK5000 Versal
Internal SRAM (MB)	41	24
Internal SRAM bandwidth (TB/s)	30	24
External DDR (GB)	32	16
External DDR bandwidth (GB/s)	38	103
External HBM (GB)	8	N/A
External HBM bandwidth (GB/s)	460	N/A

Since the first target is to implement the Super model of Bonito, none of these FPGAs can internally store 5 layers of this model. However, with the Alveo U280, it is possible to deploy a double buffering scheme for the weight memories. This double buffering means that whilst one layer is calculated the next layer is loaded in, this prevents the calculations from stalling between layer calculations due to the weights fetching from DRAM. The Alveo additionally offers an HBM which is DRAM memory with a higher bandwidth which can help by loading the next layer weights during calculations faster. However,

due to the large chunk size, it is expected that the normal DDR memory is fast enough. For double buffering, there is only a need for $2 \times 16.8 + 4 = 37.6MB$ of storage since the input and hidden/output storage can be flipped after each layer. Considering this, the choice for FPGA has fallen on the Alveo U280.

4.2. System architecture

Before starting an implementation, an initial abstract system architecture is laid out. An overview of this abstract architecture can be seen in Figure 4.2. This architecture is based on the implementation using a layer-for-layer approach; as mentioned in the memory requirement exploration, the only way to implement the Super model of bonito on the available FPGAs is by using a layer-for-layer approach. This layer-for-layer approach entails that for each subsequent layer, the weights have to be read from an external memory. However, an LSTM layer is additionally stateful, meaning that the calculations are dependent on a state determined from previous calculations. The state in the LSTM layer consists evidently of the long-term memory/cell state (c_t) and short-term memory/hidden state (h_t). This state cannot be discarded in layer transitions as this will invalidate the calculation in the next iteration of calculations in the layers. Consequently, on a layer transition, the state of the new layer should be read from the external memory while the state of the finished layer should be written back to the external memory. Next to the weights being loaded in, the biases of the gates in the LSTM cells should additionally be loaded into cells from the external memory.

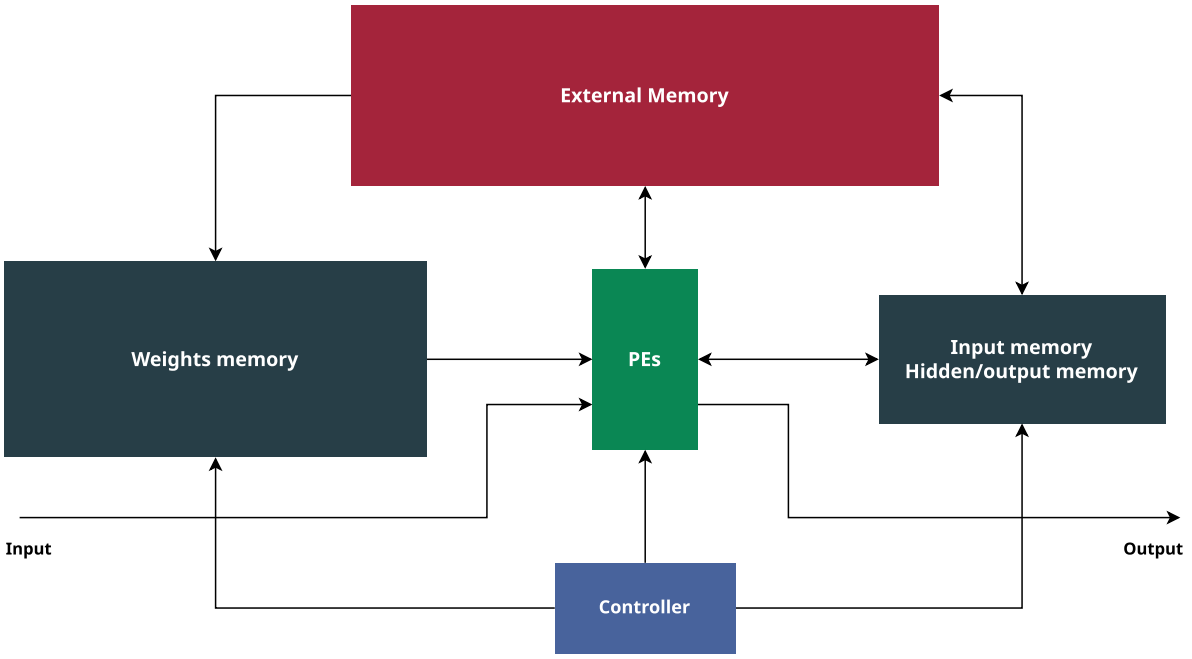


Figure 4.2: Abstract overview of system architecture

The following list abstractly outlines each component from the overview in Figure 4.2:

- **Processing Engines (PEs)**: The PEs are the computational units of the implementation each PE will be equivalent to an LSTM cell. The PEs additionally store the biases and the cell state.
- **External memory**: This is an off-chip memory, the memory type DDR or HBM memory will be decided upon later depending on the computational throughput of the implementation. The external memory will store all the accompanied data for each layer: the weights, the biases, and the layer state.
- **Input/Output stream**: These are not necessarily components but the input of the first LSTM layer is streamed in from outside the FPGA and subsequently the output of the last layer is streamed out of the FPGA. The streams are both possible input/output of the PEs.
- **Input and hidden/output memory**: This is on-chip memory, this memory consists of two memories an input memory and a hidden/output memory. An input vector from the input memory will be fed to the PEs and the output vector of the PEs will be written to the hidden/output memory, with exceptions in the first and last layer as mentioned before. Additionally to the input vector,

the last output vector will be reinserted into the PEs as a hidden vector, hence the name of the memory hidden/output memory. The input and hidden/output memory will not be a fixed memory but more a functionality applied to 2 memories. The reason being that the output of one layer is the input of another layer, this means that the hidden/output memory will function as an input layer in a subsequent layer and for the input memory vice versa.

- **Controller**: The controller is an over-arching component that enables synchronization between data transfers from the various memories, i.e. the weights input of the PEs should be synchronized with the corresponding input and hidden data put into the PEs. Moreover, it keeps track of when to switch between layers and commands the necessary components to retrieve the next layer data (weights, state, and biases) from the external memory while additionally managing that the state is written back to the external memory.

The architecture is designed from scratch as there is not a lot of literature on LSTM implementation on FPGA, and the ones that exist are of very small sizes. An architecture that exists that inspired this project's design a little bit is an LSTM implementation by H. Wang et al. [62], however, their architecture is different in a couple of aspects:

1. They only use a one-layer model therefore the entire switching out on layer transitions is not present.
2. Their LSTM layer is a bi-directional layer meaning that the input data is analyzed from both directions in the layer. This is not present in the Bonito model, however, the computational approach can be applied to a uni-directional LSTM as the computational approach is the same in both directions.

Besides this, their implementation inspired the architecture on a computational level but everything is essentially designed from scratch.

4.3. Component design

This section examines how the previously outlined components in the system architecture are implemented. The design process of the implementation is split into three sections:

- **Processing engines** which discusses the core LSTM calculating units and how they can be mapped to the hardware available in the chosen FPGA.
- **Memory management** which discusses the arrangement of the internal memories and external memories and how they are those memories can be connected to each other.
- **Control and final architecture** which discusses the final ensemble of all the components together. Moreover, it outlines the design of the control units that link and synchronize the memory management and PEs together.

The FPGA implementation designed in this thesis is developed using the Xilinx Vivado toolchain, including simulations, synthesis, and implementation. All inter-component communication is designed using the AMBA AXI4 specification. AXI4 is used mainly due to it providing a consistent, fully specified, and popular component interface specification and additionally because all the IPs[1] of Xilinx in Vivado are designed using this component interface [63], thus allowing for easy communication between those IPs and self-made components. Moreover, the design process of each component has gone through the following list of design steps in nearly all cases:

1. Outlining design requirements of the component
2. Sketch on paper a basic architecture of the component
3. Implement the component in VHDL
4. Write a testbench in VHDL to test the validity of the component and check if it correctly interacts with other connected components by emulating those other components.
5. Synthesizing the component to obtain the resource consumption on the FPGA.

Naturally, these steps will be iterated over if redesigns are necessary due to possible excessive resource consumption. While additionally, these steps can be quite time-consuming as each little component has to be tested, it will result in much less work when finishing up the design in the end and ensuring the final validity of the design. Moreover, in VHDL testing a smaller design is orders of magnitude times faster than testing the whole design, so this will reduce the amount of test iterations of the complete final architecture.

4.3.1. Processing engines

The processing engines are the core calculating part of the LSTM implementation. Processing engines (PEs) do not generally store the data to be processed but most often stream in data and subsequently stream out transformed output data, the exception being that the long-term memory is stored in the PEs, however, this is considered a state and not an output product. Before starting the implementation of the PEs, it is important to first take into consideration the data format to be used in the system architecture. After selecting a data format, there will be looked at ways of mapping the LSTM algorithm onto the selected FPGA given its constraints.

Data format

The data format is a crucial design decision during FPGA development to try to obtain the right balance in resource consumption and accuracy. The Bonito model used a 32-bit floating-point, also called single precision, data format at the beginning of this thesis project. However, during the course of the thesis, this actually changed to using a 16-bit floating-point (half precision) data format. The arithmetic units of a GPU are mainly designed for floating point operations, while GPUs can perform integer operations this

[1] IP stands for intellectual property, however in the context of Vivado it is better to interpret it primarily as a library component written by Xilinx (or any other company) that is additionally their intellectual property. In this thesis, it will mean a pre-made component of Xilinx unless stated otherwise.

is most often not at the same level of performance. Since there are very few restrictions on FPGAs, it is possible to utilize data formats other than floating-point, i.e. most predominantly fixed-point. In the fixed point format, as the name implies, the decimal point is fixed between an integer part and a decimal part, where both these parts have some amount of bits assigned to them. With a floating-point, the decimal point can float and consist of two parts: a fractional part and an exponent part which can be represented as $\textit{fraction} \times 2^{\textit{exponent}}$. Floating-point arithmetic has more accuracy as the representable range through the floating decimal point is much larger than that of fixed-point. However, the resulting hardware generated for floating-point arithmetic is often much larger than fixed point due to the following aspects:

- A floating-point operation often consists of multiple atomic arithmetic operations. For example in a floating-point multiplication, the exponents have to be added while the fractional parts have to be multiplied.
- Floating points have to be aligned. This can be in the case of an addition that the exponents of the inputs should be aligned having to additionally shift the fractional part. Additionally, in the output of a floating-point operation, the resulting fractional part has to be realigned so that the most significant bit is equal to a 1.
- There might also be rounding circuitry. There are multiple rounding modes in floating-point, while historically a truncation was applied resulting in no extra hardware, with the introduction of the widely-used floating-point standard IEEE754 the default requires some rounding, however, there is still the possibility for using different modes [64].

Conversely, in fixed-point arithmetic, these points are not present or less hardware-intensive [65]. Fixed-point addition and subtraction are equal algorithmically to integer addition and subtraction. Fixed-point multiplication requires somewhat more hardware to perform multiplication, it can be done by doing integer multiplication however the output has to be scaled depending on the inputs. However, FPGAs, like the one used in this project, most often have dedicated hardware on-chip to perform fast arithmetic called digital signal processors (DSPs). These DSPs are fixed hardware blocks that cannot be reconfigured and are generally faster and more resource efficient than if one would design something themselves using reconfigurable resources. These DSPs can perform fixed point arithmetic using little or no additional supporting reconfigurable resources, while this is certainly not the case with floating point [66]. Therefore, it is most often encouraged that designers would first explore the possibility of using fixed-point compared to floating-point. Typically in applications that are mapped from CPU or GPU implementations to FPGA, the CPU or GPU implementations use floating-point as that is by far the most common way on those platforms to implement real numbers. However, there is a high possibility that those numbers do not even require the large range provided by floating point, making them have a small or no error when they are converted into a fixed point format. For this reason, the same will be explored about the data of Bonito to verify if a conversion to fixed-point is possible.

To verify if converting the data in the Bonito network from half precision to fixed point requires not only the weight data from the LSTMs of Bonito but also the input data requires verifying. Getting the weight data out of the model is not hard as these are already separately stored. The data to be analyzed is retrieved from Bonito's super model, however, it is assumed that these results will not differ extremely for the HAC and fast model. Before doing any conversion, the original data is analyzed to determine the range of the data, see Table 4.3. The first thing to notice from the statistics of the original data is that the data is signed, this means that one bit in the fixed-point data format has to be reserved for a sign indicating positive and negative numbers. Furthermore, it is apparent that 95% of the weight data is in the order of magnitude of 10^{-1}, while the maximum or minimum value does not exceed the value of 5. Utilizing this knowledge, using a fixed point representation of 16-bits with one sign bit, 3-bits for the integer part to allow for the values exceeding 4, and the rest (12-bits) for the fractional part seems for an optimal fixed-point format; this format can also be denoted as `s16/12` for signed, a total of 16-bits, and a 12-bit fractional part. The process of analyzing this conversion is performed by loading in the data in Python and using a library to convert the data to a fixed-point format, in this case `s16/12`. Then convert it back and subtract the absolute value of the conversion from the original to obtain the absolute error, as seen in Table 4.4. The results of the conversion are promising as 25% of weights have no error after conversion, except for the input weights of layer 1.

Table 4.3: Statistics of the original Bonito weight data

		Original				**Absolute**					
	Weights	Mean	SD	Max	Min	Mean	95th percentile	3rd quartile	median	1st quartile	5th percentile
Layer 0	input	-1.16E-03	0.093	1.555	-2.240	6.81E-02	1.76E-01	9.42E-02	5.32E-02	2.45E-02	4.76E-03
	hidden	-6.82E-04	0.148	3.109	-2.453	1.09E-01	3.03E-01	1.50E-01	8.09E-02	3.67E-02	4.76E-03
Layer 1	input	-3.61E-04	0.129	2.818	-2.607	9.32E-02	2.61E-01	1.27E-01	6.90E-02	3.13E-02	6.06E-03
	hidden	-1.36E-04	0.157	3.287	-2.785	1.15E-01	3.18E-01	1.59E-01	8.68E-02	3.95E-02	6.06E-03
Layer 2	input	2.33E-04	0.125	4.859	-2.973	9.01E-02	2.50E-01	1.23E-01	6.70E-02	3.06E-02	5.94E-03
	hidden	2.49E-05	0.162	3.973	-3.527	1.18E-01	3.28E-01	1.62E-01	8.75E-02	3.97E-02	5.94E-03
Layer 3	input	7.71E-05	0.124	2.775	-4.445	9.05E-02	2.46E-01	1.24E-01	6.85E-02	3.15E-02	6.15E-03
	hidden	-9.08E-04	0.147	2.924	-2.799	1.06E-01	2.96E-01	1.46E-01	7.95E-02	3.64E-02	6.15E-03
Layer 4	input	-1.39E-04	0.128	3.617	-3.908	9.34E-02	2.54E-01	1.28E-01	7.07E-02	3.25E-02	6.34E-03
	hidden	5.61E-04	0.141	4.172	-3.451	1.02E-01	2.82E-01	1.40E-01	7.66E-02	3.51E-02	6.34E-03

Table 4.4: Absolute error of the conversion of the Bonito weights to a 16-bit fixed point with 3-bit integer part and 12-bit fractional part

s16/12

	Weights	Mean	SD	Max	Min	95th percentile	3rd quartile	median	1st quartile	5th percentile
Layer 0	input	9.83E-05	7.13E-05	2.44E-04	0	2.14E-04	1.53E-04	1.07E-04	3.05E-05	0
	hidden	8.48E-05	7.23E-05	2.44E-04	0	2.14E-04	1.22E-04	9.16E-05	0	0
Layer 1	input	8.99E-05	7.21E-05	2.44E-04	0	2.14E-04	1.37E-04	9.16E-05	0	0
	hidden	8.27E-05	7.23E-05	2.44E-04	0	2.06E-04	1.22E-04	8.39E-05	0	0
Layer 2	input	9.09E-05	7.20E-05	2.44E-04	0	2.14E-04	1.45E-04	9.16E-05	0	0
	hidden	8.21E-05	7.24E-05	2.44E-04	0	2.06E-04	1.22E-04	7.63E-05	0	0
Layer 3	input	9.07E-05	7.19E-05	2.44E-04	0	2.14E-04	1.45E-04	9.16E-05	0	0
	hidden	8.56E-05	7.22E-05	2.44E-04	0	2.14E-04	1.22E-04	9.16E-05	0	0
Layer 4	input	8.98E-05	7.20E-05	2.44E-04	0	2.14E-04	1.37E-04	9.16E-05	0	0
	hidden	8.70E-05	7.22E-05	2.44E-04	0	2.14E-04	1.22E-04	9.16E-05	0	0

Moreover, the maximum error is still 1 order of magnitude smaller than the 5th percentile of the original data and the 95th percentile of the error is 3 orders of magnitude smaller than the 95th percentile of the original data. Other fixed-point formats have been checked, however, reducing the integer bits to 2 results in a very high maximum error which seems to be unpreferable as these large values have a lot of impact on these small numbers, so having a high error on them can lead to very erroneous results of the whole network.

While the fixed-point results of the weights are very promising, verifying the input data is on the contrary very challenging. This is mainly due to the input data of the Bonito network not being the input of the LSTMs as the input of the network is going first through the CNN layers, etc. Intercepting the data after these initial layers proved to be challenging as well because the Bonito network internally is constructed quite complexly due to multi-threading which makes it quite challenging to alter the network or debug the network to retrieve input samples. Another possibility to validate the impact caused by conversion to fixed-point is to fully run the Bonito network with half-precision and another run with fixed-point and check the output accuracy by means of validation tools provided by Bonito. However, as mentioned before GPUs do not regularly use fixed-point, therefore there is no support in PyTorch [67] meaning that is virtually impossible to validate the conversion this way. Lastly, as a last resort is converting the input in the LSTM to fixed-point using the library used for the weight analysis and then converting it back to half precision. This way it is possible to at least introduce the errors caused by the conversion, however, the calculations are still in half precision. While this method did work, it was so slow that even a single inference took more than 3 minutes, after which the process was killed because this would not be a possible method, as it would take way too much time to run the entire network. This concluded the path of using fixed-point as it would be impossible to verify that converting the network to this data format would not decrease the accuracy too much, even though it showed a lot of potential.

The results of the fixed-point analysis mean that the only other option is to use floating-point. There is another solution which is quantizing the weights into 8-bit or even 4-bit integers. However, this is only applied to the weights and is mainly utilized to reduce I/O bandwidth as the weights are scaled to float point by multiplying the integer weights with a floating point scaling factor which means that calculations are still performed in floating point [67]. Add to this that the implementation would most likely be computationally bound and not I/O bound due to the chunk size as mentioned before, this means that this quantization is not necessary and only increases the computation time and thus the entire inference time.

However, in floating-point there are two different 16-bit formats, the first one ”half precision” has already been discussed while the second one is bfloat16. A bfloat16 is different from half precision in that a bfloat16 has the same size exponent part as a single precision 32-bit floating-point which means that its fractional part is smaller and thus less accurate however it has a larger representable range. A half precision has an exponent size of 5 bits and a fraction of 10 bits, while a bfloat16 has an exponent size of 8 bits and a fraction of 7 bits. While it seems that the fraction part is more important than the exponent part looking back at Table 4.3, however, it depends on the range of the input data. Thus at first face value, it seems that half precision is the best format, however, switching over to a bfloat16 later is not complicated as this only means tweaking some settings and does not result in any component redesign.

Outlining design

With the data format chosen, being 16-bit floating point, it is possible to look at designing the architecture for the processing engines. As mentioned before, the processing engines should transform the input data using the LSTM algorithms to obtain the correct inference data. From the theoretical analysis of the LSTMs, recall equations 4.8 to 4.13, the LSTM inference can be divided into two phases: a matrix-vector multiplication phase (Equation 4.8 to 4.11) calculating the gate outputs and a post-processing phase (Equation 4.12 and 4.13) combining the gate outputs to obtain the cell state and hidden state.

$$i_t = \sigma(W_{i\_input}x_t + W_{i\_hidden}h_{t-1} + b_i) \tag{4.8}$$

$$f_t = \sigma(W_{f\_input}x_t + W_{f\_hidden}h_{t-1} + b_f) \tag{4.9}$$

$$\tilde{c}_t = tanh(W_{\tilde{c}\_input}x_t + W_{\tilde{c}\_hidden}h_{t-1} + b_{\tilde{c}}) \tag{4.10}$$

$$o_t = \sigma(W_{o\_input}x_t + W_{o\_hidden}h_{t-1} + b_o) \tag{4.11}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{4.12}$$

$$h_t = o_t \odot tanh(c_t) \tag{4.13}$$

To more clearly describe these separate phases, the equation can be rewritten as Equations 4.14 to 4.27. These clearly show the matrix-vector phase in Equations 4.14 to 4.21 and the post-processing phase in Equations 4.22 to 4.27.

$$i_{acc\_input} = W_{i\_input} x_t \tag{4.14}$$

$$i_{acc\_hidden} = W_{i\_hidden} h_{t-1} \tag{4.15}$$

$$f_{acc\_input} = W_{f\_input} x_t \tag{4.16}$$

$$f_{acc\_hidden} = W_{f\_hidden} h_{t-1} \tag{4.17}$$

$$\tilde{c}_{acc\_input} = W_{\tilde{c}\_input} x_t \tag{4.18}$$

$$\tilde{c}_{acc\_hidden} = W_{\tilde{c}\_hidden} h_{t-1} \tag{4.19}$$

$$o_{acc\_input} = W_{o\_input} x_t \tag{4.20}$$

$$o_{acc\_hidden} = W_{o\_hidden} h_{t-1} \tag{4.21}$$

$$i_t = \sigma(i_{acc\_input} + i_{acc\_hidden} + b_g) \tag{4.22}$$

$$f_t = \sigma(f_{acc\_input} + f_{acc\_hidden} + b_f) \tag{4.23}$$

$$\tilde{c}_t = \sigma(\tilde{c}_{acc\_input} + \tilde{c}_{acc\_hidden} + b_{\tilde{c}}) \tag{4.24}$$

$$o_t = \sigma(o_{acc\_input} + o_{acc\_hidden} + b_o) \tag{4.25}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{4.26}$$

$$h_t = o_t \odot tanh(c_t) \tag{4.27}$$

From these rewritten Equations 4.14 to 4.27, a list of necessary arithmetic components can be assembled for each phase can be described, allowing the construction of a single processing engine, this list is as follows:

- Matrix-vector multiplication phase
  - Multiply accumulator
    - Adder
    - Multiplier
- Post-processing phase
  - A sigmoid unit
  - A hyperbolic tangent unit
  - Adders
  - Multipliers

First, the two phases will be designed separately after which those design results will be put together and connecting them.

Matrix-vector multiplier

Firstly, there will be a look at implementing the matrix-vector multiplication units. Calculating a full matrix-vector multiplication in a single cycle, especially of the Bonito sizes, is impossible; this means that the matrix-vector calculation has to be segmented. Looking at the resources provided by the Alveo U280, there are 9024 DSPs available. Dividing those DSPs over the 8 matrix-vector calculating units means that each matrix-vector multiplication unit has access to 1128 DSPs. Looking at the Bonito model, the super model is a matrix-vector multiplication of a matrix of size 1024×1024 and a vector of size 1024. This shows great potential for completely unrolling a single axis of the matrix-vector multiplication, with unrolling meaning that a single axis is calculated in one cycle, provided that each index uses only one DSP. Whilst in the matrix there are two possible axes to unroll, see Equation 4.28 and 4.29, the Equations have lines added to them to illustrate the segmentation of the calculation over time where each line equals a clock cycle[2]. The most optimal axis of the matrix to unroll is the vertical axis as seen in Equation 4.28. The reason for this is that the resulting calculation in a cycle in Equation 4.28 has no dependency whilst the resulting calculation in Equation 4.29 is completely dependent on each other. The result of this is that Equation 4.28 are simple parallel multiply accumulations, whilst Equation 4.29 results in multipliers and massive serial adder trees to combine all the multiplication results which due to serial nature probably slow down clock speed considerably.

$$\left[\begin{array}{c|c|c} a & b & c \\ d & e & f \\ g & h & i \end{array}\right] \times \left[\begin{array}{c} x \\ \hline y \\ \hline z \end{array}\right] = \left[\begin{array}{c|c|c|c|c} a \times x & + & b \times y & + & c \times z \\ d \times x & + & e \times y & + & f \times z \\ g \times x & + & h \times y & + & i \times z \end{array}\right] \tag{4.28}$$

$$\left[\begin{array}{ccc} a & b & c \\ \hline d & e & f \\ \hline g & h & i \end{array}\right] \times \left[\begin{array}{c} x \\ y \\ z \end{array}\right] = \left[\begin{array}{c} a \times x + b \times y + c \times z \\ \hline d \times x + e \times y + f \times z \\ \hline g \times x + h \times y + i \times z \end{array}\right] \tag{4.29}$$

The initial design is based on the fixed-point format as it showed great promise. Implementing a fixed-point design proved to be relatively easy due to the fixed-point packages released in VHDL-2008, this package allows the use of fixed-point formats in VHDL code without resorting to standard logic vectors. While this design was in the near stages of completion, it was still scrapped in favor of the half precision floating-point format for the reason mentioned in the data format section. While the VHDL-2008 library also provided a similar floating-point package, this package is not very publicly endorsed on fora due to the fact that it does not support any pipelining, alternatively, it is considered better to use the floating-point IPs provided by Xilinx to perform arithmetic operations. As mentioned before, the IPs include an AXI4 streaming interface allowing them to be applied more easily to the PEs in the design. Additionally, this IP has a configuration page that allows for easy setting of the exponent and fraction part of the floating-point to be calculated, this means that switching between different formats over the whole system design is fairly easy as long as the bit-width of the data format does not change. The floating-point IP does not support a multiply-accumulate configuration, which would be the most ideal implementation for a matrix-vector multiplication. However, the IP does support a fused multiply-add (FMADD) configuration, see the left side of Figure 4.3. In this configuration, the IP has two DSP configurations medium usage using 1 DSP and full usage using 3 DSPs, the medium usage will compensate for fewer DSPs by replacing them with LUTs [68]. Since for the super model it is only possible to use 1 DSP, the medium usage seems ideal. However, an issue arises due to the latency of the IP, the IP has a minimum latency of 1 clock cycle, which means that the FMADD result is only available after 2 clock cycles. This is not a problem if the implementation only requires FMADDs as the IP allows for pipelining but in this case, where an accumulation is required it results in a stall every other cycle as the result is not ready every cycle to be reinserted in the FMADD.

Due to the latency reason, the FMADD-configured IPs are not usable, making it necessary to opt for a new design. This design utilizes the floating point IP two times: once configured as a multiplier and another configured as an accumulator, see right of Figure 4.3. However, this design puts forward a decision to which IP the DSP is assigned. Looking at the resource consumption of both IP configurations it is best to assign the DSP to the multiplier as it saves 332 LUTs while assigned to the accumulator only

[2] In the result of Equation 4.28 the cycle segmentation are illustrated by | + | and not just a single line.

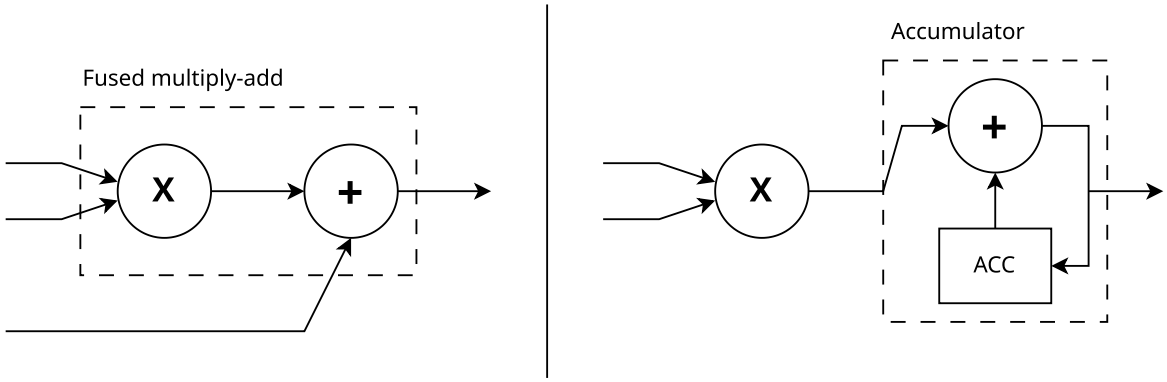


Figure 4.3: The fused multiply-add unit (left) compared to the multiply-accumulate unit (right)

38. This difference in LUT consumption means that the DSP will be used for the core operation and a multiplication requires more hardware than an addition. The accumulator IP will continuously accumulate until the AXI4 last signal, indicating the last transaction, is triggered on the input, the accumulator will reset once it outputs the last signal itself.

Post-processing phase

The post-processing phase follows after the accumulation is finished and it combines those accumulation results from each gate output to calculate the cell state (c_t) and hidden state (h_t), Equations 4.30 to 4.35 show the post-processing these relating equations.

$$i_t = \sigma(i_{acc\_input} + i_{acc\_hidden} + b_g) \tag{4.30}$$

$$f_t = \sigma(f_{acc\_input} + f_{acc\_hidden} + b_f) \tag{4.31}$$

$$\tilde{c}_t = \sigma(\tilde{c}_{acc\_input} + \tilde{c}_{acc\_hidden} + b_{\tilde{c}}) \tag{4.32}$$

$$o_t = \sigma(o_{acc\_input} + o_{acc\_hidden} + b_o) \tag{4.33}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{4.34}$$

$$h_t = o_t \odot tanh(c_t) \tag{4.35}$$

The set of Equations 4.30 to 4.35 result in the following list of atomic operations to be performed by the post-processing design.

- 8 additions
- 3 multiplication
- 3 sigmoid activations
- 2 hyperbolic tangent activations

The first step in designing the post-processing phase is transforming the set of Equations 4.30 to 4.35 into a timeline to demonstrate sequentially all the arithmetic operations that are performed and how they depend on each other. A visualization of this timeline can be seen in Figure 4.4. With this timeline, a more optimal list of required floating-point operations can be assembled by analyzing how many parallel operations are performed, this list is as follows:

- 3 adders
- 2 multiplication
- 2 sigmoid activations
- 1 hyperbolic tangent activations

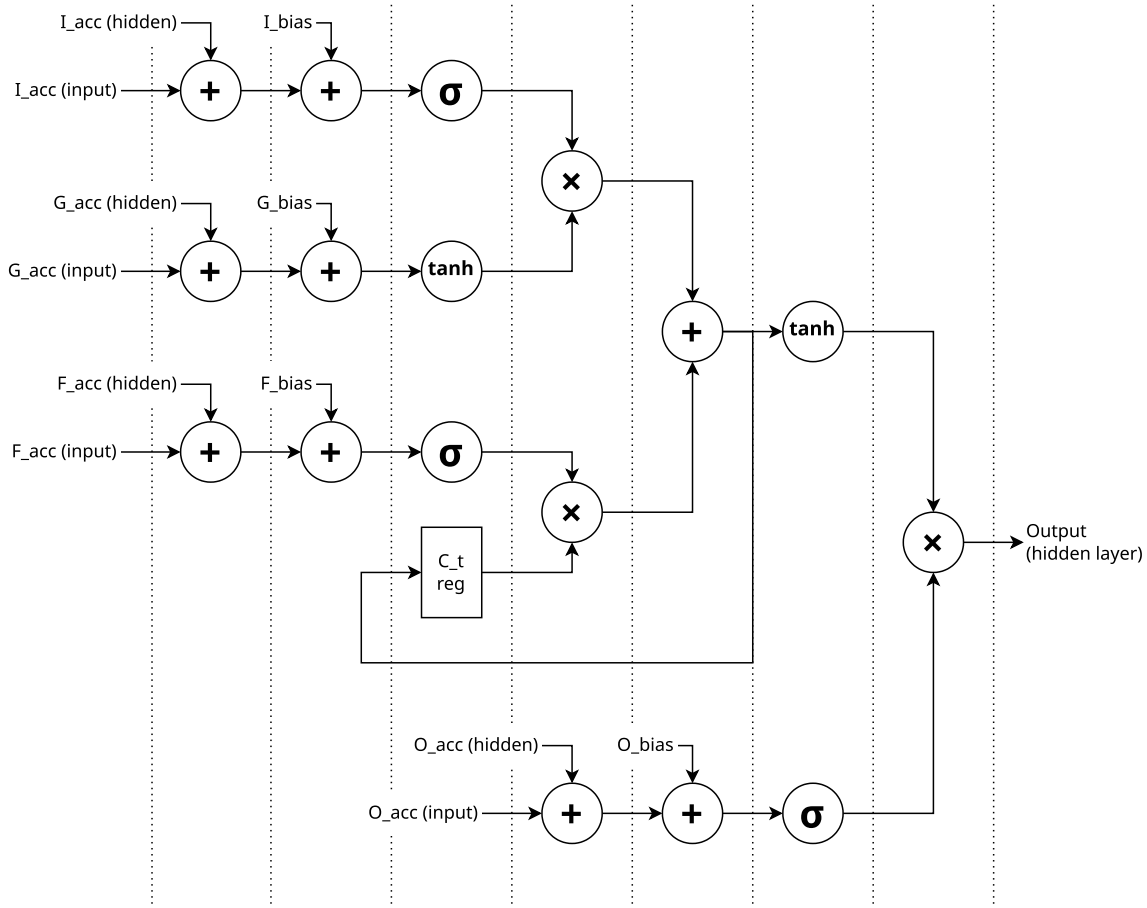


Figure 4.4: Timeline of an optimal sequence to perform the post-processing phase

Since each processing engine contains already 8 multipliers and accumulators, resource-wise it is most optimal to reuse those to perform the multiplications and additions. However, this means that the output of multiplication results has to be diverted away from entering the accumulator and values have to be inserted from outside into the accumulator, as seen on the left in Figure 4.5. However, this modification makes the design much more complex to implement in VHDL and additionally, the accumulator is more difficult to use as an adder due to the interface. One would think that since the input of the adders is already an accumulated value the interfacing is not hard but the last signals reset the accumulators thus the last signals have to be intercepted as well resulting in a lot more circuitry. Moreover, in the normal case of addition if the value is not in the accumulator the addition takes two cycles because the accumulator only accepts one input per cycle. For these reasons, it has been decided to go for the design on the right of Figure 4.5 and go for separate adder-configured IPs but keep the multiplier modification. The adders do not require a lot of hardware to be implemented while the multipliers do especially without DSPs as discussed before. Currently, the design uses 3 adder-configured IPs to allow for the optimal throughput but if this results in too much hardware it can be opted to use fewer adders and perform the addition sequentially instead of in parallel.

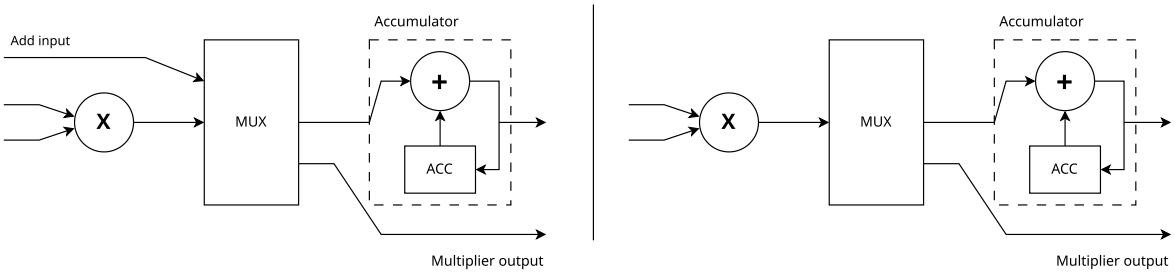


Figure 4.5: MAC with (optional) add input and multiplier output

The activation functions sigmoid and hyperbolic tangent (tanh) are more difficult to compute in hardware. These activation functions are both not linear, being exponential and hyperbolic functions (see Equation 4.36 and 4.37) for sigmoid and tanh respectively.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.36}$$

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{4.37}$$

Due to their non-linearity, these functions are not trivial to implement in hardware, therefore, most often in hardware design, there is a trade-off between hardware complexity and accuracy loss due to approximation [69]. To make this trade-off decision, multiple activation function techniques will be analyzed.

- **Xilinx IPs**: The first implementation is designing a pipeline using the IPs provided by Xilinx. By examining Equation 4.36 and 4.37, a pipeline can be constructed as seen in Figure 4.6 and 4.7 where each dotted box indicates a clock cycle. The additions and multiplications can be performed by the multipliers in the MAC unit and adders existing in the PEs. The reciprocal and exponential would require their own IPs. While the reciprocal uses only 17 LUTs and the exponential requires 167 LUTs for half precision, the exponential IP also has a hard requirement to use 2 DSPs [68]. Since the implementation has allocated all DSPs to the MACs, this makes this implementation of activation not applicable. Moreover, it is not well documented how this exponential is implemented algorithmically or how accurate they are.

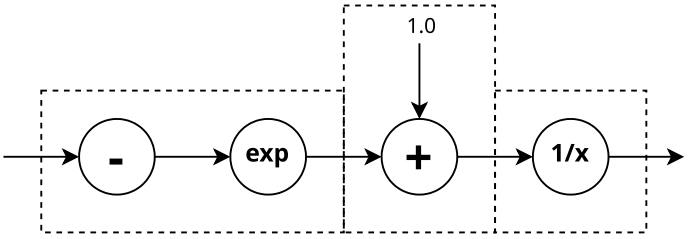


Figure 4.6: Pipelining of the sigmoid function using floating-point IPs

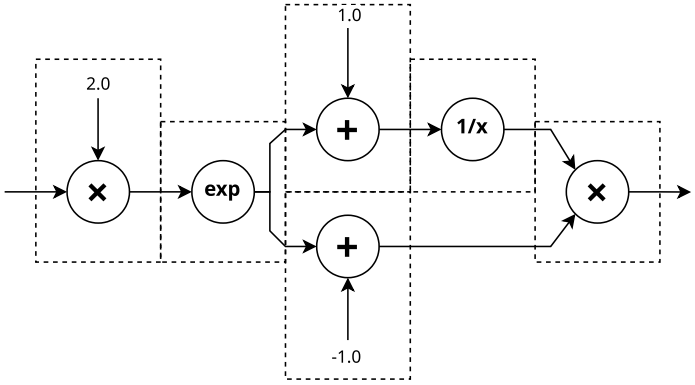


Figure 4.7: Pipelining of the tanh function using floating-point IPs

- **Non-linear piece wise approximation**: A very accurate implementation is using non-linear piece-wise approximation [70]. In this design, the sigmoid curves are approximated by constructing logarithmic piecewise functions. While this is very accurate, it also requires a lot of resources to calculate the logarithmic functions, amounting to around 11 DSPs; and as mentioned before this is too much for this design.

- **CORDIC**: CORDIC, which stands for "Coordinate Rotation Digital Computer," is a digital algorithm used for efficiently calculating various trigonometric and other transcendental functions using only simple arithmetic and bit-shifting operations. The CORDIC algorithm operates by iteratively rotating a vector in a coordinate system by small angles until the desired trigonometric value is achieved [71]. It can approximate the sigmoid and tanh functions with very high accuracy [72] and Xilinx even provides IPs to perform CORDIC operations. However, these IPs consume around 3000 LUTs which is too much to use them in the implementation.

- **Look-up tables**: This approximation works by indexing a look-up table with the input data to look up what the corresponding output is. The accuracy is naturally dependent on the amount of entries in the look-up tables, where more entries result in more generated hardware. The accuracy hardware trade-off is not very competitive compared to other approximations as a lot of LUTs are required to still have fairly low accuracy [73][74].

- **Piece-wise linearization**: The piece-wise linearization method seems to be a popular approach to perform activation function approximation on FPGAs [75][76]. Piece-wise linearization is a mathematical technique used to approximate a complex nonlinear function with a series of linear segments. The idea is to break down the nonlinear function into smaller intervals where it behaves more linearly. The piece-wise linearization is particularly interesting as the existing multipliers and adders can reused to calculate the linear segments in the form of $y = ax + b$, resulting in low additional needed hardware to implement this.

- **Range-addressable look-up tables**: Range-addressable look-up tables (RaLUTs) are very similar to look-up tables approximation. However, in RaLUTS the index address of the LUT is variable in width, allowing for very granular and coarse indexing steps between entries. This variability is useful for asymptotic functions like the sigmoid and tanh. As seen in Figure 4.8, the derivative

value is very high in both functions in the area around $x = 0$, while the derivative quickly goes to zero outside this area. This can be used in a RaLUT by using a lot of entries around zero as values change quickly there, and using fewer and fewer entries when it goes to zero. There is, however, little to no information on how to construct such a RaLUT and it seems to be only mentioned as a possible approximation but is rarely used.

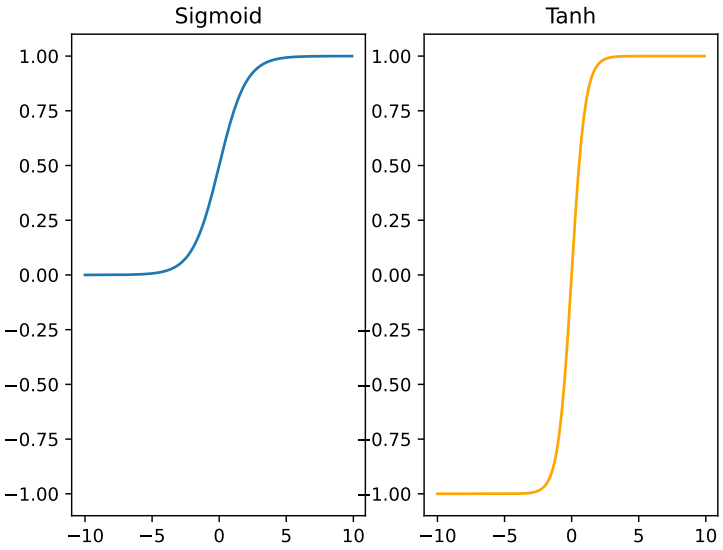


Figure 4.8: Plot of sigmoid function and hyperbolic tangent (tanh)

After considering all these implementation options, the decision has fallen to use piece-wise linearization to approximate the activation functions. The reason for this decision mainly comes from the balance it strikes between accuracy and hardware consumption, while additionally having the ability to reuse the hardware of the MACs to perform the linear calculations. Z. Li et al. [75] published research on piece-wise linearization of the sigmoid activation function. They provide multiple piece-wise linearization functions of which they state that 10 pieces is optimal due to little gains from having more pieces. This piece-wise function is shown in Equation 4.38. However, a pre-made piece-wise function does not exist for tanh meaning that one has to be created from scratch. Since the creation method is well documented by Z. Li et al. the same method will be applied to make the tanh piece-wise function.

$$p_{\sigma\_10}(x) = \begin{cases} 0 & x \leq -8 \\ 0.00252x + 0.01875 & -8 < x \leq -4.5 \\ 0.02367x + 0.11397 & -4.5 < x \leq -3 \\ 0.06975x + 0.25219 & -3 < x \leq -2 \\ 0.14841x + 0.40951 & -2 < x \leq -1 \\ 0.2389x + 0.5 & -1 < x \leq 1 \\ 0.14841x + 0.59049 & 1 < x \leq 2 \\ 0.06975x + 0.74781 & 2 < x \leq 3 \\ 0.02367x + 0.88603 & 3 < x \leq 4.5 \\ 0.00252x + 0.98125 & 4.5 < x \leq 8 \\ 1 & x > 8 \end{cases} \tag{4.38}$$

First, the curvature of the tanh will be determined using Equations 4.39 to 4.41 to see which parts of the tanh are the least linear. Plotting the curvature, see Figure 4.9, it is apparent that the curvature of the tanh is the largest around the area of $x = 1$.

$$\frac{dtanh(x)}{dx} = sech(x)^2 \tag{4.39}$$

$$\frac{d^2tanh(x)}{dx^2} = -2sech(x)^2tanh(x) \tag{4.40}$$

$$curve(tanh) = \frac{\left|\frac{d^2tanh(x)}{dx^2}\right|}{\left(1+\left(\frac{dtanh(x)}{dx}\right)^2\right)^{3/2}} \tag{4.41}$$

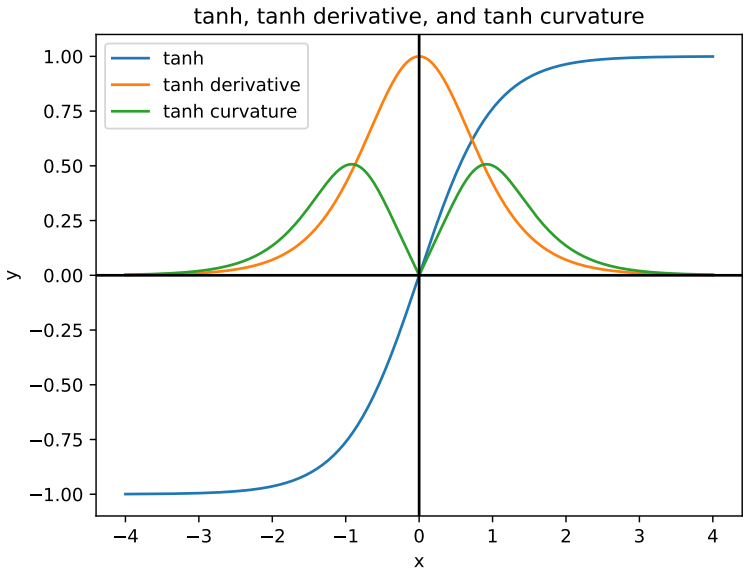


Figure 4.9: Plot of the derivative and curvature of the hyperbolic tangent (tanh)

The method starts by sampling n samples uniformly from the tanh function ($\vec{y}$), see Equation 4.42. As seen in Figure 4.41, the figure has asymptotes on $y = -1$ and $y = 1$ from $x = -4$ and $x = 4$ respectively, therefore the samples are sampled uniformly from $[-4, 4]$ ($\vec{x}$). The sample size is 1000, the same as in the reference paper.

$$\vec{y} = tanh(\vec{x}) \tag{4.42}$$

According to the paper, the piece-wise linear function can be constructed using Equation 4.43, where m equals the number of linearized pieces.

$$p(x) = \begin{cases} -1 & x \leq b_1 \\ \beta_1 + \beta_2(x - b_1) & b_1 < x \leq b_2 \\ \beta_1 + \beta_2(x - b_1) + \beta_3(x - b_2) & b_2 < x \leq b_3 \\ \vdots & \vdots \\ \beta_1 + \beta_2(x - b_1) + \beta_3(x - b_2) + \cdots + \beta_m(x - b_{m-1}) & b_{m-1} < x \leq b_m \\ 1 & x > b_m \end{cases} \tag{4.43}$$

The piece-wise functions from Equation 4.43 are placed in matrix A (Equation 4.45) for each input sample and are activated using the step function $\delta_{x_i > b_j}$ (Equation 4.44).

$$\delta_{x_i > b_j} = \begin{cases} 0 & x_i \leq b_j \\ 1 & x_i > b_j \end{cases} \tag{4.44}$$

$$A = \begin{bmatrix} 1 & x_1 - b_1 & (x_1 - b_2)\delta_{x_1 > b_2} & \cdots & (x_1 - b_{m-1})\delta_{x_1 > b_{m-1}} \\ 1 & x_1 - b_1 & (x_1 - b_2)\delta_{x_1 > b_2} & \cdots & (x_1 - b_{m-1})\delta_{x_1 > b_{m-1}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n - b_1 & (x_n - b_2)\delta_{x_n > b_2} & \cdots & (x_n - b_{m-1})\delta_{x_n > b_{m-1}} \end{bmatrix} \tag{4.45}$$

With the matrix from Equation 4.45 and the tanh samples ($\vec{y}$), the $\vec{\beta}$ vector can be calculated to obtain the final piece-wise function, see Equation 4.46.

$$\vec{\beta} = \left(A^T A\right)^{-1} A^T \vec{y} \tag{4.46}$$

The difficult part is selecting the right values for b the bounds of the pieces. The paper is not really clear how these are bounds i.e. in Equation 4.38 are selected. However, they derive them somehow from the curvature of the function. Trying to do this for tanh resulted in subpar results therefore it was chosen to choose the optimal bound values for tanh by brute forcing using a script for every combination of bounds in the interval $[-4, 4]$ and selecting the one with the smallest error. This is subsequently done from 4 to 14 pieces, as seen in Figure 4.10; the average absolute error is calculated according to Equation 4.47.

$$e_{avg} = \frac{\sum_{i=1}^{n} \left| A\vec{\beta} - \vec{y} \right|_i}{n} \tag{4.47}$$

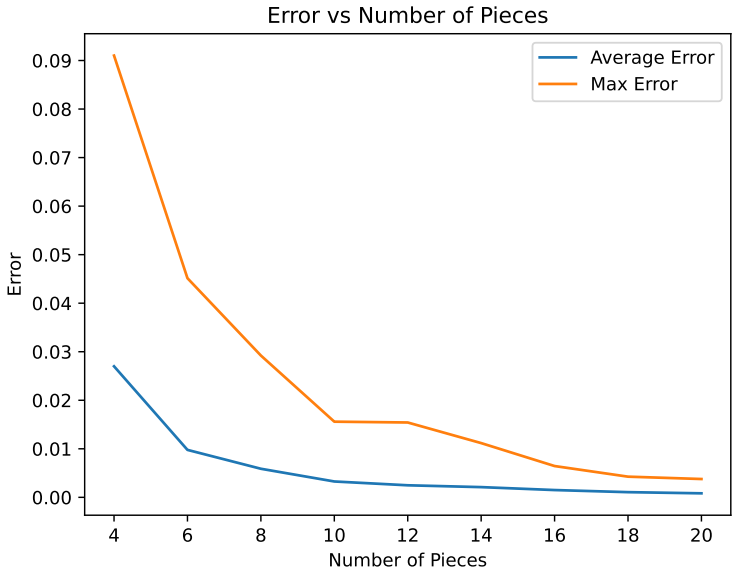


Figure 4.10: The error due to the piece-wise linearization against the number of utilized pieces

From the results from Figure 4.10, it has been decided to use the 16 pieces as the max error plateaus after that value. Figure 4.11 shows the error of the 16 pieces and clearly shows that as expected the error around $|x| = 1$ is the largest as the curvature and non-linearity is the highest there as discussed in Figure 4.9. The final resulting piece-wise function is shown in Equation 4.48.

$$p_{tanh\_16}(x) = \begin{cases} -1 & x \leq -4 \\ 0.00739x - 0.97183 & -4 < x \leq -2.5 \\ 0.05619x - 0.84983 & -2.5 < x \leq -1.75 \\ 0.18788x - 0.61937 & -1.75 < x \leq -1.25 \\ 0.36098x - 0.40300 & -1.25 < x \leq -1.0 \\ 0.50082x - 0.26316 & -1.0 < x \leq -0.75 \\ 0.69131x - 0.12029 & -0.75 < x \leq -0.5 \\ 0.87373x - 0.02908 & -0.5 < x \leq -0.25 \\ 0.99007x & -0.25 < x \leq 0.25 \\ 0.87373x + 0.02908 & 0.25 < x \leq 0.5 \\ 0.69131x + 0.12029 & 0.5 < x \leq 0.75 \\ 0.50082x - 0.26316 & 0.75 < x \leq 1.0 \\ 0.36098x + 0.40300 & 1.0 < x \leq 1.25 \\ 0.18788x + 0.61937 & 1.25 < x \leq 1.75 \\ 0.05619x + 0.84983 & 1.75 < x \leq 2.5 \\ 0.00739x + 0.97183 & 2.5 < x \leq 4 \\ 1 & x > 4 \end{cases} \tag{4.48}$$

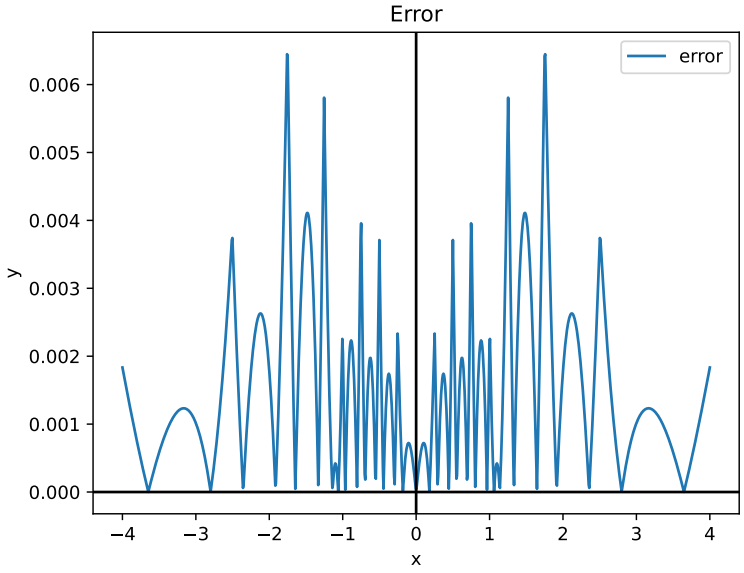


Figure 4.11: The error due to the piece-wise linearization using 16 pieces

With the piece-wise functions in Equation 4.38 and 4.48, the sigmoid and tanh functions can be implemented in hardware. The hardware design consists of three stages, as seen in Figure 4.12. The first stage consists of the sigmoid/tanh decision-maker, this component determines the slope and offset value depending on in which segment the input lays. This decision-making is essentially a look-up table for the slope and offset value with as input the boolean output of comparators which check if the input lay in a selected segment. Due to the segments purposely being powers of 2, these comparisons are very lightweight as only a few bits of the exponent and fraction parts of the floating point have to be analyzed. Next to the slope and offset, the decision-maker unit additionally outputs the input value to calculate the linearization in the subsequent stages with the slope and offset. Moreover, it outputs an optional value output for the asymptotic outputs which require no linearization calculation, if this value is outputted the input, slope, and offset outputs are disabled by being flagged as invalid, and vice-versa for the value output when the slope and offset are being outputted.

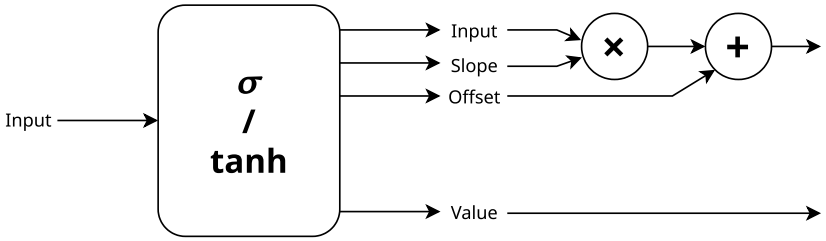


Figure 4.12: Implementation of a sigmoid or a tanh decision-maker

With the activation functions implemented, it is possible to make the final post-processing timeline as seen in Figure 4.13, with the activation functions unrolled as seen in Figure 4.12. Altogether, the post-processing phase has 11 stages of calculations, as seen in Figure 4.13.

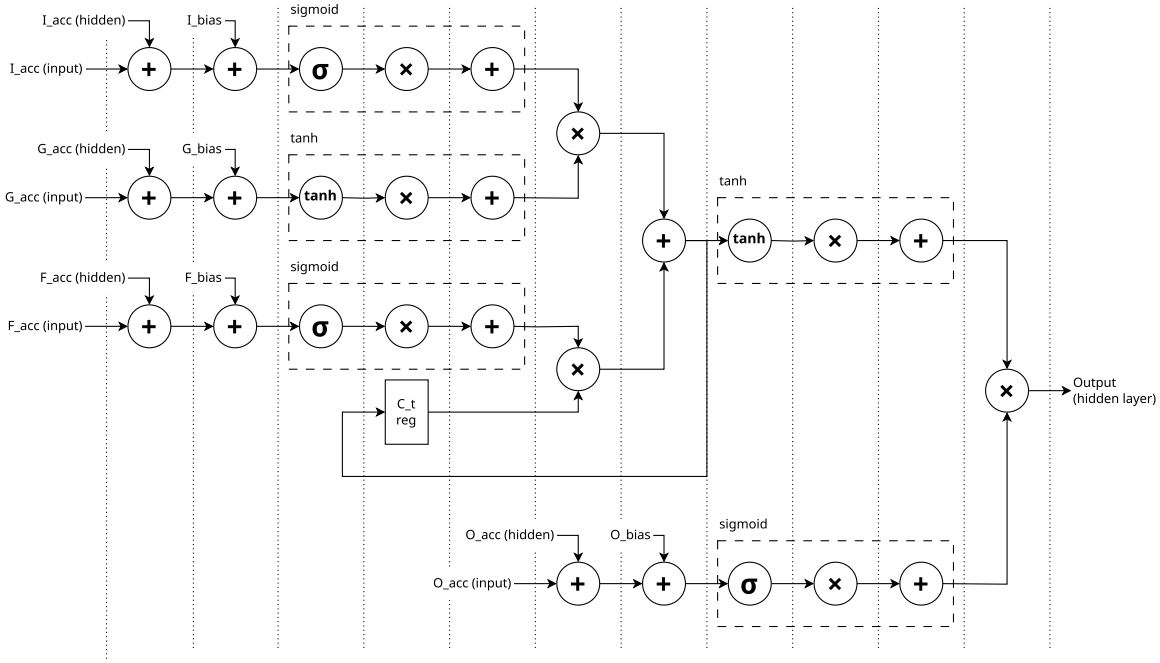


Figure 4.13: Timeline of a sequence to perform the post-processing phase with sigmoid and tanh performed as piece-wise linearization

Initial design

With both hardware designs for the matrix-vector phase and the post-processing phase ready, the PE architecture can be finalized. The primary state machine of the PEs can be seen in Figure 4.14. There are two phases in the matrix-vector phase as the matrix-vector calculation has a latency of 2 cycles, this means that if the input triggers the last signal the MACs output the last output two cycles later. In this finishing state, inputs have to be set to not ready meaning that the PEs are not ready to process new data which is equally valid for the post-state, otherwise the next inference will be started from the input side. In the post-stage 11, the output is ready and the PE does not go to the next state until this output is ready to be received on the receiving side.

Figure 4.15 shows the architecture of the PE. As outlined in the post-processing design, there is a need for 3 multipliers, therefore 3 of the MAC units are in the form in which the multiplication can be diverted away from the accumulator. Additionally, there are post-buffers in the design where the outputs of each post-state can be stored and are used again as inputs in the subsequent post-state. In total, there are 9 of the post-buffers as the sigmoid and tanh produces at maximum 3 outputs each,

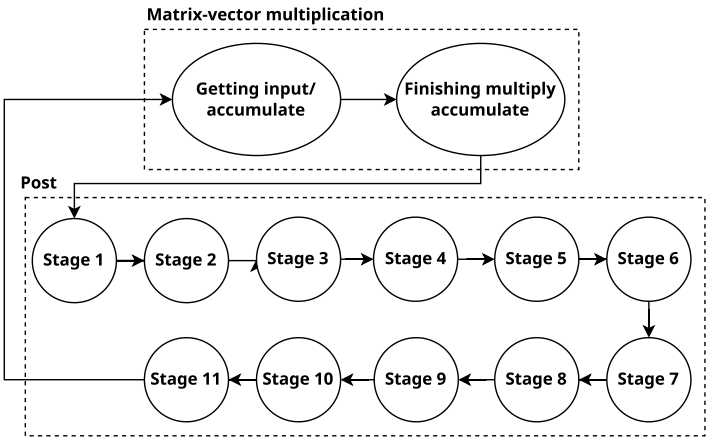


Figure 4.14: Flowchart of states in the PE state machine

while additionally having three decision-makers in parallel. However, these post-buffers do make the post twice as slow as the result has to be writing the result and reading cannot occur in the same clock cycle. Additionally, there is a mention of the c_t and bias inputs shown in Figure 4.15, however, these will be later discussed in the section about memory management.

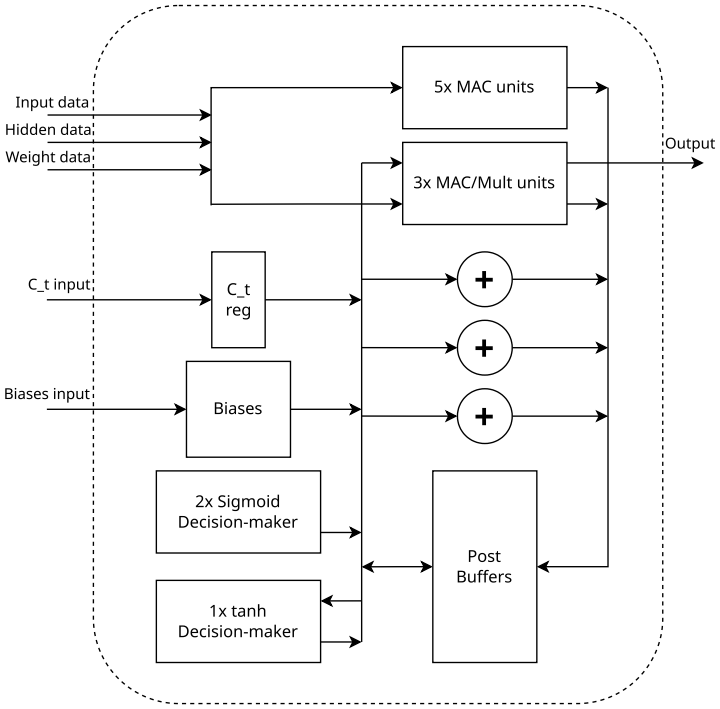


Figure 4.15: Abstract hardware architecture of the processing engines

However, after synthesizing the PEs it became apparent that resource consumption was really excessive, especially in LUTs, as seen in Table 4.5 that shows the result of a partially finished implementation. During the design phases, the LUTs were not really taken into consideration as they were hard to predict and the resource estimation tool in Vivado is not very accurate. After these results, it is clear that implementing the super model is not possible, only if the design of the matrix-vector multiplication is also segmented over the y-axis 5 times; which additionally leads to 5x less performance naturally.

It is clear that the implementation has to be downsized in some way. The focus was therefore shifted to the HAC model. The number of PEs is now 384 instead of 1024, while this is only a reduction of

Table 4.5: Resource utilization of partially finished PE for super model

Num PEs	LUT	Flip-Flops	DSPs
1024	428.63%	211.57%	90.78%

2.3x instead of the required 5x, there are also more DSPs available so LUTs can be replaced by DSPs allowing the MACs to be fully implemented with 3 DSPs. However, there are not enough DSPs to use only full DSP MACs but this results in utilization of 102% (see Table 4.6). This overflow in DSPs can be reduced to 100% by replacing some MACs with 1 DSP MACs (as used in the super model) in a ratio of 372 full DSP MACs and 12 1 DSP MACs. However, by using 1 DSP MACs, the LUT consumption naturally becomes higher again, see Table 4.6

Table 4.6: Resource utilization of PE for HAC model

	Num PEs	LUT	Flip-Flops	DSPs
Full DSP MACs	384	203.98%	111.56%	102.13%
Partial 1 DSP MACs	384	208.14%	111.56%	100%

Iteration

After looking further into the documentation and having gained more experience over the course of the project, a possibility was found to configure the fused multiply-add IP in a non-blocking manner. The non-blocking allows the fused-multiply add to be calculated without any latency, this means that every clock cycle an FMADD operation can be performed and that the output is already present before the next rising edge; allowing the output to be reinserted in the next cycle as the additive. This means that there are no problems with time dependency on the accumulation anymore. Additionally, this simplifies the design in many aspects:

- Due to the non-blocking configuration and zero latency, the FMADD units do not have ready signals anymore which also reduces the amount of flip-flops used as the inputs do not have to be buffered in the case the FMADD is not ready.
- The multiplication output does not have to be diverted away in the post-state as the FMADD can be used as a multiplier by using a 0 as an additive. This means that there is no need for different FMADD designs unlike with the MACs.
- The FMADDs can also be used as an adder by calculating $y = add_1 \times 1 + add_2$ using the FMADD, which means that there is no need for separate adders.
- The activation functions can be calculated within 2 cycles as opposed to 3 with the MACs, as the linearization calculation $y = ax + b$ can be calculated in 1 cycle, which means that the post-processing has 9 stages as opposed to 11, see Figure 4.16.

Moreover, due to the non-blocking and less complex design, the post-processing has been reduced to 1 cycle instead of 2 for each stage, meaning for each inference the post sequence is reduced from 18 to 9 cycles. This reduction is possible because there is no ready check and a value can be immediately inserted into the next arithmetic unit because it does not have to be buffered in the post buffers. The post buffers are still used to store the result of the $o_{input} + o_{hidden} + o_{bias}$ result, see Figure 4.16, and to store the value output of the activation decision-maker if it does not have to go through the linearization. Using the FMADD units the current PE design is as shown in Figure 4.17. In the current configuration, the FMADD IPs use 3 DSPs, as seen before this results in more DSPs than available which means that a few PEs have to be configured using 1 DSP. The total resulting resource consumption can be seen in Table 4.7, where it has been chosen to use the DSPs as much as possible resulting in 370 PEs with 3 DSPs and 14 with 1 DSP. Additionally, the LUT utilization is still high but it is not expected that the memory management will take up too much resources.

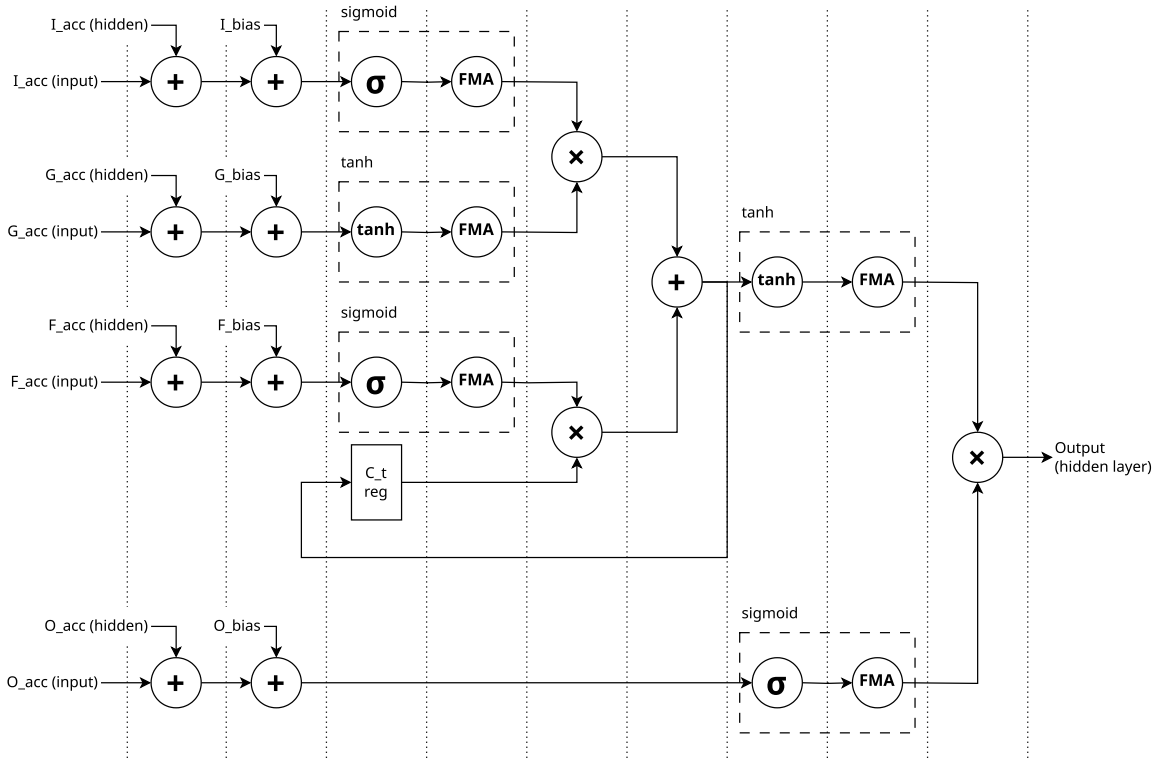


Figure 4.16: Timeline of a sequence to perform the post-processing phase using fused multiply-add

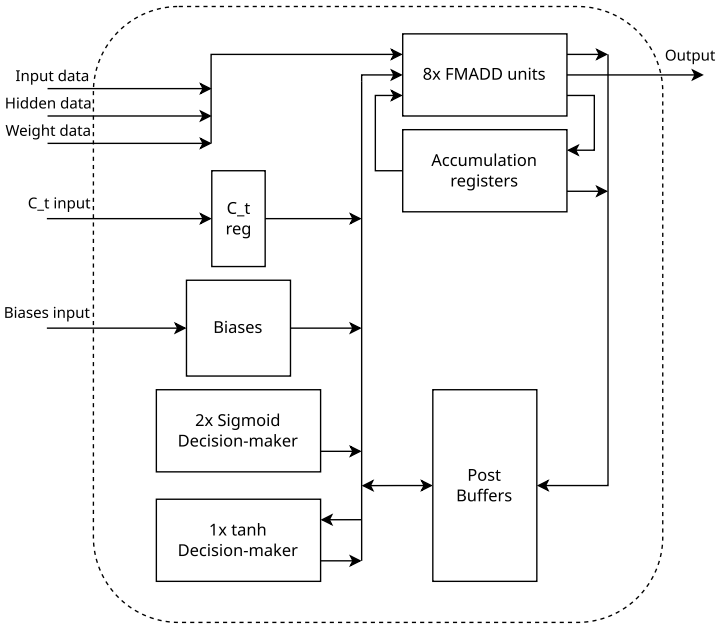


Figure 4.17: Abstract hardware architecture of the processing engines using fused multiply-add

Table 4.7: Resource utilization of the PE after the design iteration

PE type	Num PEs	LUT	Flip-Flops	DSPs
Full DSP	370	90.00%	11.81%	98.40%
LUT/DSP	14	5.63%	0.45%	1.24%
Total	384	95.63%	12.26%	99.65%

4.3.2. Memory management

After finishing the designs of the PEs, the infrastructure for streaming in and out the data has to be designed. Memory management is notoriously hard in hardware description languages due to having to orchestrate many distributed memories without any pre-made foundation for the memory infrastructure [77]. Naturally, this allows for a very granular and performant design but at the cost of a lot of verbosity and programming errors in the design process resulting in a lot of testing iteration making it time-intensive.

First, the external memory of the design is to be determined, HMB or DDR. Then the memory infrastructure is designed and a timeline of the schedule of the dataflow. Finally, there will be a look at how to implement the distributed internal memories.

External memory

Before calculating and determining which external memory to use, first, an assessment of the data transfers for each layer has to be made. The following list shows all the data transfers:

- Reading in the cell state (c_t) and writing it back on when the layer is finished.
- Reading in the biases
- Writing out the hidden state of the previous layer and reading in the hidden state for the next layer
- Reading in the weights

For these transfers, the data size can be determined as seen in Table 4.8; all data has a bit-width of 16 bits. The cell state, hidden state, and biases are all vectors of size 384, however, there are 4 bias vectors. The weights naturally dominate the data transfers being 8 matrices of 384×384. Altogether, this results in a transfer size of *2.37 MB* per layer.

Table 4.8: Transfer sizes of data per layer

Transfer	Transfer times	Transfer size	Total size
Cell state	2	0.77 KB	1.54 KB
Hidden state	2	0.77 KB	1.54 KB
Biases	1	3.07 KB	3.07 KB
Weights	1	2.36 MB	2.36 MB
Total			**2.37 MB**

Since the PE design is now known, it is possible to determine the amount of clock cycles per layer. The matrix-vector calculation takes 384 cycles plus the additional 9 cycles for the post and while using a chunk size of 1000, this results in each layer taking 3.93×10^5 cycles to calculate. Moreover, the HBM has a bandwidth of *460 GB/s* while the DDR has a bandwidth of *38 GB/s*. This means that the transfer time to transfer the *2.37 MB* per layer is $62.2\mu s$ and $8.48\mu s$ for DDR and HBM respectively. This means that to not oversaturate the maximum clock speed is *6.31 GHz* and *46.35 GHz* for DDR and HBM respectively. These clock speeds are unreachable on FPGAs, therefore it is safe to say that the implementation is computationally bound. Additionally, this means that the DDR will suffice and that there is no need for HBM. Moreover, HBM also requires additional hardware resources than DDR which are already lacking in this LSTM implementation.

Memory Architecture Design

In Figure 4.18, the abstract dataflow and required memories can be seen. Moreover, due to the implementation being computationally bound, it is possible to apply a double-buffering method. However, the representation in Figure 4.18 is not possible to realize on the FPGA as the DDR DRAM has only an interface of 512 bits, this interface is provided by the DDR IP which abstracts the DDR interface and instead delivers an AXI4 interface. To utilize this interface in the implementation, it has been chosen to use a bus interface of 512 bits to transfer data to the internal memories, see Figure 4.19.

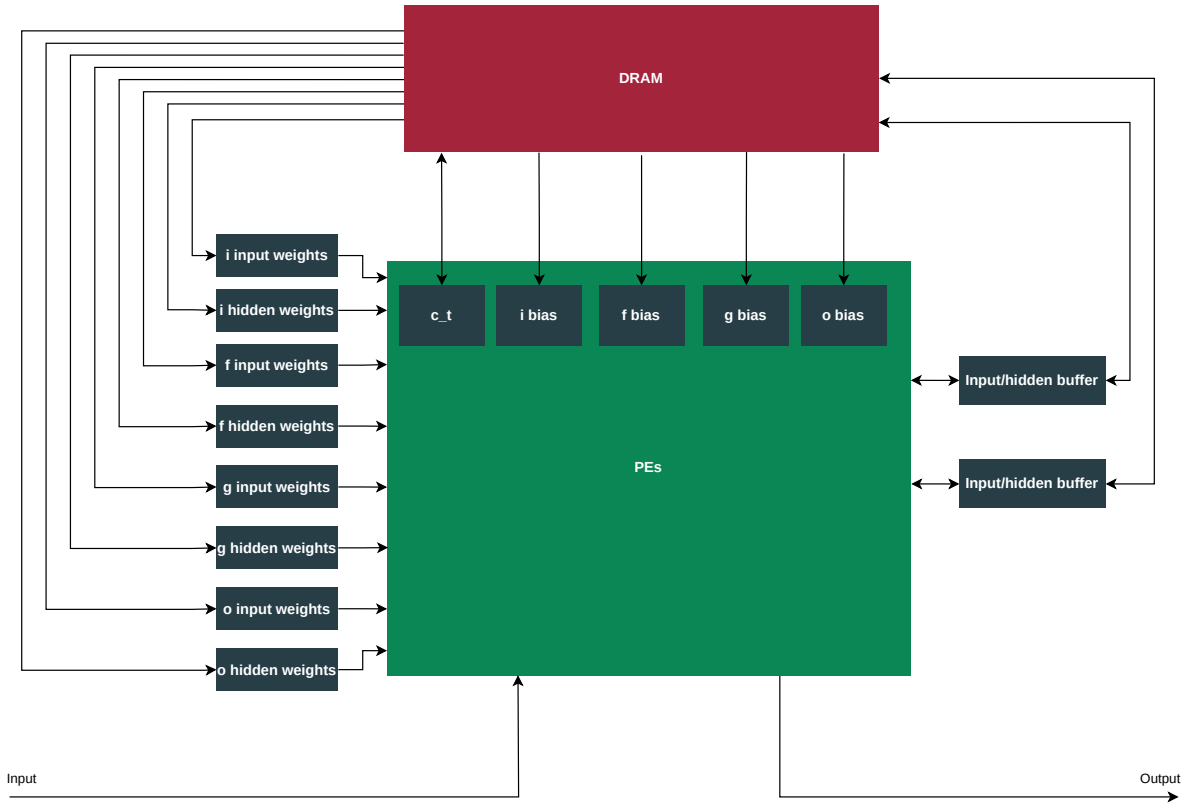


Figure 4.18: Full overview of memories and dataflow in the system architecture

The bus has to be bi-directional as the LSTM state has to be written back to DRAM, this complicates the design a little bit, however, this can be solved by utilizing the tri-state buffers on the FPGA. Each component has a bus interface with these tri-state buffers, these buffers allow the bus interfaces to read and write from the bus or go in a high-impedance state when the current bus interface is idle. This high-impedance mode ensures that an idle interface does not interrupt the communication on the bus between two other interfaces.

By using a bus there is also a need to schedule, see Table 4.9, the read/write process over the bus as, naturally, the DRAM cannot provide all the transfer data in one cycle and it is not allowed that two different interfaces write to the bus at the same time. The first transfers in the schedule are to read in the cell state (c_t) and biases for the current computing layer. Transferring these is possible as during the matrix-vector multiplication this data is not needed yet, only in the post phase, this provides a window of 384 to transfer the cell state and biases which should take only 60 cycles, therefore double buffering is not applied to these memories. After that, the transfer order is not that critical, the hidden state of the previous layer is written out which is buffered in the *Input Hidden/Output Bus Interface* since the last inference of the previous layer. After that, the largest transfers occur of the weights which is followed by reading the hidden state of the next layer. The hidden of the next layer is buffered in the *Next layer hidden buffer* (see Figure 4.19), this buffer is only used for the first inference of a layer as hidden input for the PEs, for the subsequent inferences the output of the PEs is used, as mentioned before. After this, the bus waits until the last inference has occurred and the c_t memory in the PEs is flagged as valid, after which the c_t state is written back to the DRAM. To organize the transfers, the *DRAM bus interface* has an additional output to all the other bus interfaces indicating the bus state. The bus states are the transfers seen in Table 4.9 and an additional `IDLE` state in which every bus interface goes in high-impedance mode, using these bus states the other bus interfaces can be coordinated.

The DRAM bus interface has another task next to coordinating, which is writing and reading from the DDR DRAM. The data that the DRAM has to store for each layer is a cell state, hidden state, 4 bias vectors, and 8 weight matrices, which amount to *2.37 MB* as mentioned before or a total of *11.85 MB*.

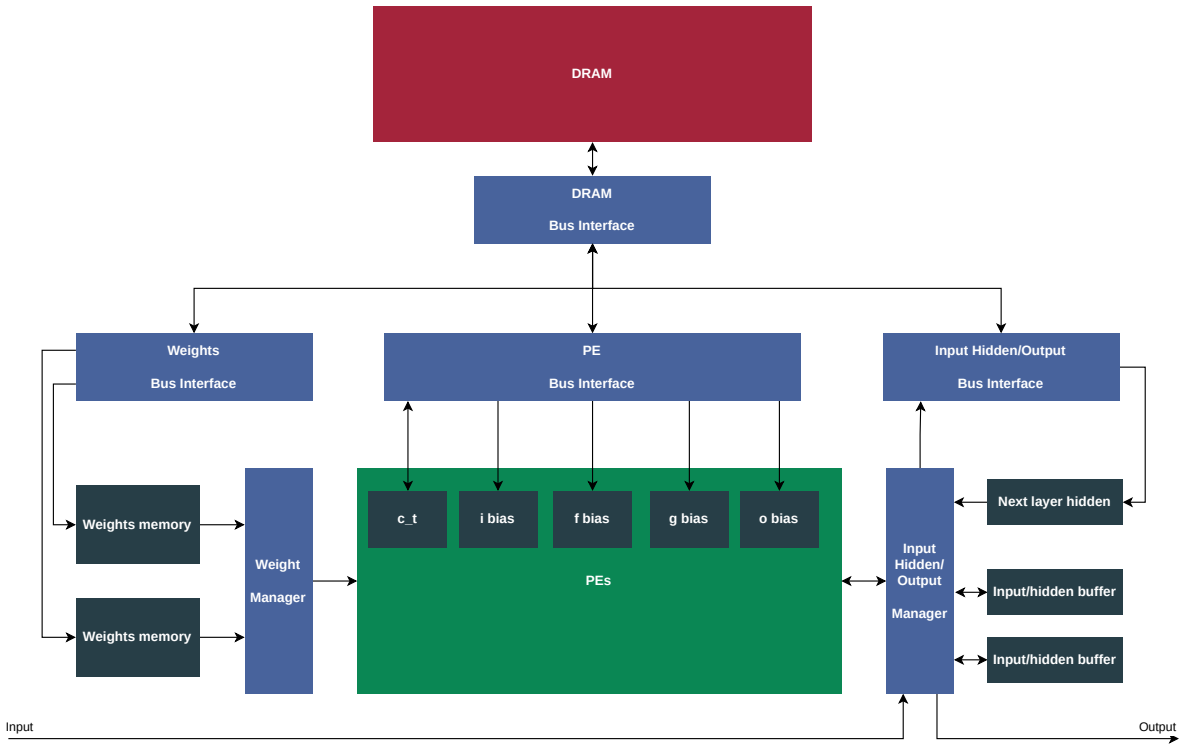


Figure 4.19: Implementation of the memory architecture

Transfer	DRAM operation	Size (bits)	Cycles
c_{layer}	Read	384×16	12
I bias	Read	384×16	12
F bias	Read	384×16	12
G bias	Read	384×16	12
O bias	Read	384×16	12
$h_{layer-1}$	Write	384×16	12
I_{input} weight	Read	$384 \times 384 \times 16$	4608
I_{hidden} weight	Read	$384 \times 384 \times 16$	4608
F_{input} weight	Read	$384 \times 384 \times 16$	4608
F_{hidden} weight	Read	$384 \times 384 \times 16$	4608
G_{input} weight	Read	$384 \times 384 \times 16$	4608
G_{hidden} weight	Read	$384 \times 384 \times 16$	4608
O_{input} weight	Read	$384 \times 384 \times 16$	4608
O_{hidden} weight	Read	$384 \times 384 \times 16$	4608
$h_{layer+1}$	Read	384×16	12
c_{layer}	Write	384×16	12

Table 4.9: Schedule for reading and writing to the DRAM

Since this is a fraction of the *32 GB* that DRAM can store, it is possible to structure the data in DRAM so that it is easy to manage. The resulting DRAM address can be seen in Figure 4.20, where 3 bits are assigned to indicate the layer, 4 bits to indicate the data type (states, biases, weights), and 13 bits to index the data types where 13 bits is based on the 4608 data blocks sized 512 bits for the weights. The remaining bits are unused and set to zero.

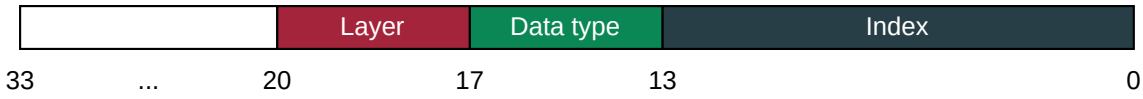


Figure 4.20: DRAM address

The weight manager and ”Input Hidden/Output Manager” in Figure 4.19 manage the data flow from the internal memories to the PEs. The weight manager reads from one of the weight memories to enable the double buffering, in the meanwhile the weights bus interface writes to the other memory. The memory from which they read and write to naturally switches between layer changes. The ”Input Hidden/Output Manager” essentially acts similarly and switches between reading and writing from the input/hidden buffers for each layer. The hidden state is actually buffered inside the manager and when the manager outputs the hidden value to the PEs, it concurrently writes it to the hidden/output buffer memory. After the last inference, there is no next inference therefore the hidden state is written to the bus interface to be written to DRAM while concurrently writing to the hidden/output buffer memory.

Distributing Internal Memory

The final piece of the memory architecture is assigning the different internal memory types to the memories. The Alveo U280 FPGA has two types 960 UltraRAMs (URAMs) and 2016 Block RAMs (BRAMs). The main difference in this use case is the read latency and the size of the memory. The URAMs can read a data entry in one cycle while a BRAM takes two cycles. A URAM primitive can hold 288 kb (kbits) of data while the BRAM can hold 36 kb of data. A 36 kb BRAM can additionally be subdivided into two BRAMs of 18 kb with each 18 kb having its own interface; the URAMs do not have such a feature.

RAM type	**Primitive size (kb)**	**Number available**	**On-Chip Size (MB)**
URAM	288	960	34.56
BRAM	36/18	2016	9.07
Total			43.63

Table 4.10: Alveo U280 SRAMs specification

However, there is a problem due to a trade-off between the data width of the memory against memory utilization. This problem is of course lessened by the changing from the super model to the HAC model, however, it is still apparent in the weight memories as for each weight memory each cycle $384 \times 16 = 6144$ bits have to be read, while a URAM only has a data width of 72 bits. This means that 86 URAMs are required to satisfy the data width of 6144 bits, which takes up 3.1 MB of storage while a single weight matrix only uses 295 KB of storage meaning only 10% of the URAMs are utilized while storing the weights, which is naturally very inefficient. A way to solve this data-width problem is by *”over-clocking”* the URAMs as the data width can be reduced by the amount the clock speed of the URAMs is increased, as the single data request can be split over the extra clock cycle obtain from *”over-clocking”*. However, instead, it has been decided to store the weights in BRAM as the BRAM are smaller memories meaning that less space is wasted by having a larger data width. It is not easy to calculate the amount of BRAMs that are generated due to the BRAMs having a variable data width depending on the data depth, however, the IP used to generate the BRAMs states that 86 36kb BRAMS are used meaning that the weights memory utilization is bumped up to 76%.

Conversely, the inputs/hidden buffers are implemented using URAMs as these do not require a large data width as each cycle only 1 input (16 bits) is read from the memory. While the utilization is not great for both memories being 33%, combined they are just two memories storing 1.5MB which is marginal

compared to the 35 MB available. It does show that utilizing the URAMs is not as straightforward and would have added more design complexity if the super model was to be fitted onto the FPGA.

The SRAMs do not allow different read-write data widths, therefore, to write to the weights memory, the weight data has to be buffered in the weights bus interface to assemble the 6144 bit data from the 512 bit data from the data bus. Moreover, this issue is more problematic after the last inference in a layer when the final result has to be written to the bus interface and hidden/output buffer memory, as this memory has a data width of 16 bits meaning that the computation must stall for 384 cycles. However, this stall amounts only to 0.09% of the complete inference cycles which is marginal.

4.3.3. Control

The system architecture's final element is a controller that orchestrates and synchronizes all the aforementioned components. The controller unit is at its core a counter which counts from 0 to 383. This counter value is outputted to the weight memory and input/hidden memory which subsequently outputs the corresponding value from their memory. Moreover, in every two layers, the values in each input vector have to be read in reverse from the input/hidden memory, this is done by subtracting the counter value from 383 and combining it with the chunk value to get the read address. This naturally results in overhead as 384 is not a power of 2, so the values for every chunk from 384 to 512 are unutilized. However, due to the poor utilization of the URAMs (33%), this overhead is negligible as the resulting address is 14-bit while the URAM block allows for 19-bit addressing. When the counter reaches 383, concurrently, a last signal is raised which triggers the PEs to go to the post-processing phase. At the same time during the counting, the DRAM interface is triggered to start running through the schedule to fetch and write the states, biases, and weights. The controller also listens to the bus state to check if the cell state and biases of the current layer are read in already, otherwise, it blocks the PEs from starting the post until those values are finished reading in. However, this stalling is very unlikely because as mentioned before reading in those values takes approximately 60 cycles while the matrix-vector calculation takes 384 cycles, but it is added as a precaution.

Going back to the last signals emitted from the control unit, a multi-dimensional last signal is used which is inspired by the Tydi interface specification [78]. While a multi-dimensional last signal is not in the AXI4 specification, it does help to easily communicate the ending of a single inference and additionally signal the last inference of the entire layer. This second last signal is controlled by a different counter which counts the chunk progress from 0 to 999 representing the chunk size of 1000. After 1000 is reached the control unit resets its counters a restarts the counters and the entire process starts for the next layer. The information of the current layer is additionally outputted to i.e. the *"input hidden/output manager* which uses this information to take the input from the input instead of the memories in the first layer or output it to the output instead of writing it to memory in the last layer, or, it uses it also to know when to read the input vectors in reverse.

Moreover, the controller manages the initialization of the system by loading the weights as due to the double buffering mechanism used in the design, it is necessary that the weights of the first layer are already pre-loaded into the SRAMs before inference.

4.4. Full system design

With all the components outlined and implemented, Figure 4.21 shows the final architecture of the FGPA implementation of Bonito. The architecture is implemented by using a layer-by-layer approach, meaning that one of the five Bonito layers is computed at a time. More layers to perform pipelining cannot be implemented due to hardware limitations as by using of 16-bit floating point, which is not natively supported on the used FPGA, the hardware can only perform 8×384 multiply-accumulate operations which is equivalent to a single layer of the Bonito high-accuracy (HAC) model. However, some optimizations have been performed such as double-buffering. Since there is enough space in on-chip memory for two layers, by using double-buffering the next layer can be loaded from external memory (DRAM) into another on-chip memory while the current layer is being computed. This double-buffering allows for immediate switching and continuation of computing the next layer when a layer is finished. Moreover, since the FPGA in this implementation is compute-bound, this allows the implementation to be continuously busy with computing layers and not be stalled with memory transfers. Next to the weights, which are the largest memory transfers, for each layer the biases and the long and short-term memory state vectors have to be loaded from DRAM and additionally the memory state vectors of the previous layer have to be written back to the DRAM. As seen in Figure 4.21, all these layer data have to be written and read from different components, therefore, a data bus has been designed for the DRAM for more organized DRAM access.

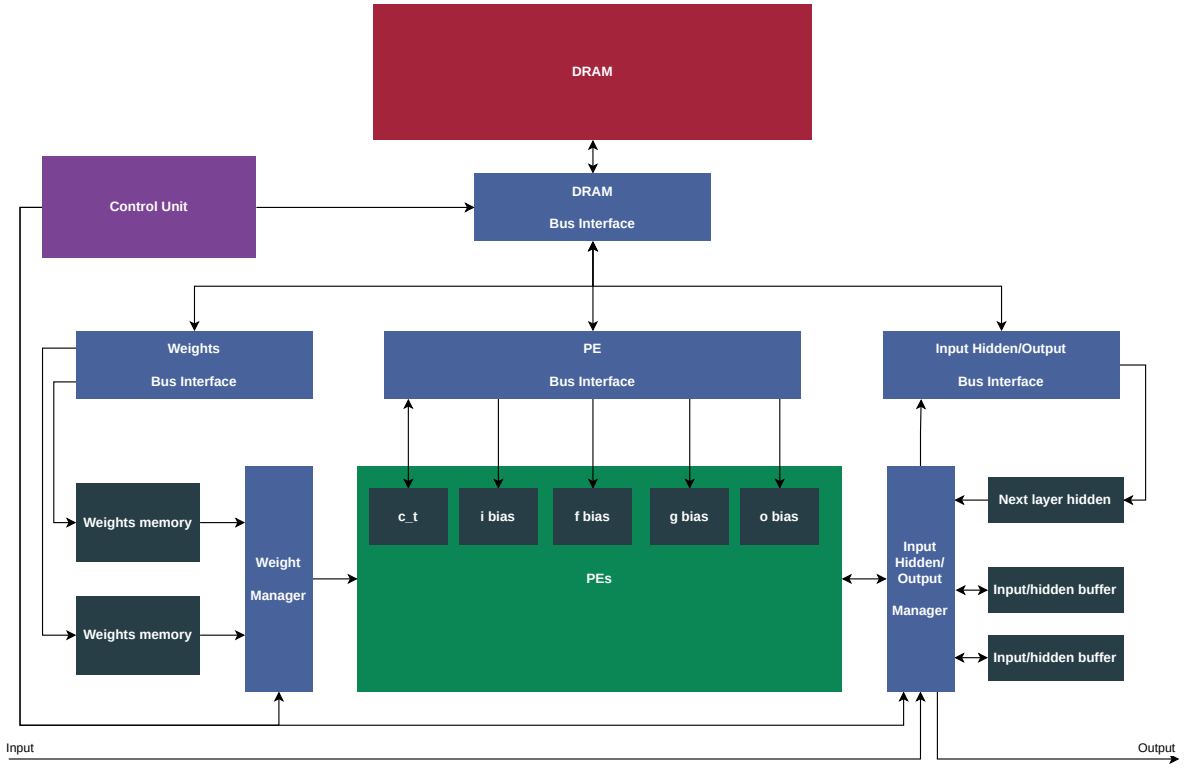


Figure 4.21: Final architecture of the FPGA implementation

5 Evaluation

This chapter focuses evaluating on the implemented design from the previous chapter. The implementation will be evaluated on the following points:

- **Resource utilization**: Validates if the implemented design fits on the FPGA.
- **Performance**: Compares the FPGA performance to the Bonito LSTM layers on the GPU.
- **Validity**: Statistically compare the output of the LSTM layers to a known functioning baseline implementation on a CPU or GPU.

5.1. Resource utilization

With the implementation finished, the design can be synthesized and implemented on the FPGA. Table 5.1 shows the synthesis results of the implementation, there are two results:

- **Full DSP**: This is the utilization by using full DSPs (3 DSPs) for the fused multiply adds (FMADD). However, as seen from Table 5.1 and as discussed before, this results in too much DSP usage.
- **Max available DSP:** This implementation replaces 2 of the DSPs with LUTS in the FMADD units leaving 1 DSP, to achieve a DSP utilization under 100%. However, doing so naturally increases LUT utilization which in turn exceeds 100%.

Implementation	LUT	FF	URAM	BRAM	DSP
Full DSP	98.40%	13.17%	26.04%	68.28%	102.13%
Max available DSP	100.63%	13.17%	26.04%	68.28%	99.65%

Table 5.1: Resource utilization of the implementation on the U280

As mentioned before, the DSPs use around 95% of the LUT consumption, this means that the rest of the components utilize the remaining 5%. After going through the utilization reports, it becomes apparent that 3.8% of this 5% is spent on the weights manager, which is the component that reads from one of the two double-buffered weight memories. However, scaled up to 8 weight memories, every cycle 8 of the 16 memories have to be read. To alternate between reading from 8 memories and in another layer from the other 8 memories, a very large MUX is required of $2 \times 8 \times 6144 = 98304$ bits to switch between the two. This large MUX is implemented using LUTs thus resulting in the 3.77% resource utilization. While it is possible to reduce the data width by *"overclocking"* the memories and buffering the results in flip flops, this will still result in around 98% which is additionally very hard to impossible to route and place on the FPGA for the implementor.

After these results, there was looked at different/newer FPGAs that with perhaps smaller transistor technology, which might not be available to this project but exist on the market. After searching, the U250 came into view, while first overlooked as it is not available at the Q&CE cluster at TU Delft,

and thought of having less due to being labeled lower than the U280, this FPGA has more primitive resources (LUTs, etc.) available than the U280. The difference between the U280 and the U250, however, is that the U250 has no HBM like the U280. The HBM is most likely replaced by the primitive resources on the FPGA, since this project does not use HBM the U250 is a good target to synthesize to. The synthesis results of the implementation can be seen in Table 5.2. Using the Alveo U250 allows all the FMADDs to be implemented with full DSPs and results in good utilization of the other primitive resources. The drawback is that this FPGA is available only at the *Heterogeneous Accelerated Compute Clusters* at ETH Zürich for which there was little time left to apply for using it and setting up and testing the implementation. Moreover, as seen later in the performance results, the performance will not be competitive on an FPGA, therefore, due to time constraints, there is more value in demonstrating the validity of the implementation than showing that it can run on the FPGA.

Implementation	LUT	FF	URAM	BRAM	DSP
Full DSP	74.24%	9.93%	19.53%	51.21%	75.00%

Table 5.2: Resource utilization of the implementation on the U250

5.2. Performance

The performance of the implementation will be compared to the equivalent HAC model Bonito GPU implementation. The performance of the GPU implementation will be examined by analyzing the trace of an inference and measuring the time spent on the LSTM layers. For this analysis, the NVIDIA RTX 2080 Ti is used which is the best GPU available at the *Q&CE cluster* at TU Delft. If the amount of *FLOP* (floating-point operations) is combined with the LSTM time, the *FLOPS* (*FLOPS*) can be determined which is a general computational throughput metric. The number of *FLOP* for a matrix multiplication of $(n \times p) \cdot (p \times m)$ is $nm(2p-1)$, this is with considering a fused multiply-add as 2 *FLOP*. Where $n = p = 384$, the layer size, and m is the batch size which is $m = 96 \times 1000 = 96000$ for the GPU as the GPU batches the chunks in batches of 96. Scaling this up to 8 matrix multiplications and 5 layers $\textit{FLOP} = 8 \times 5 \times nm(2p-1) = 566$ *GFLOP*. The *FLOPS* on the GPU is shown in Table 5.3.

Device	TFLOP	Compute time (ms)	TFLOPS
GPU	1.13	40.4	28.00

Table 5.3: Computational throughput of the HAC Bonito model on a NVIDIA RTX 2080 Ti

Conversely, on the FPGA the *FLOPS* can be calculated, however, since the the design is not implemented it is not possible to obtain a clock speed. Therefore, as an estimation, a clock speed of 200MHz has been chosen which is typical for an FPGA implementation. Moreover, a *FLOP/cycle* has to be established for the FPGA implementation to combine with the frequency to obtain *FLOPS*, since the FPGA performs $384 \times 8 = 3072$ FMADDs per cycle this is the *FLOP/cycle*. Using this frequency and *FLOP/cycle*, the *FLOPS* can be determined which is 614.4 GFLOPS. This *FLOPS* can be used to get an estimated required frequency to match the GPU implementation which is 4.67 GHz which is completely impossible to achieve as theoretically the FMADDs max out on 850 MHz, which means that computationally it is impossible to obtain the same performance as a GPU.

Device	TFLOP	Estimated compute time (ms)	TFLOPS	Required frequency (GHz)
FPGA	1.13	940	1.23	4.67

Table 5.4: Computational throughput of the HAC Bonito model on an FPGA, with the required frequency to match the GPU

5.3. Validation

An important aspect of the implementation is validating the design. Continuous validation is especially important in hardware design as due to the massive amount of signals, interconnectedness, and parallelism, it is very easy to make a mistake that will get harder to detect and fix over time. This is why every little component described in the previous chapter has gone through testing to make sure it

works. Until this point, however, the components have only been tested on their own for validity. Testing these components is done by placing the component in a VHDL testbench file. In this testbench file, time-sequential VHDL code is written to emulate the components that interact with it. This testbench can in turn be simulated using a simulator that produces signal waves like an oscilloscope, see Figure 5.1. While this simulation output is great for validating for the hardware designer, the output in Figure 5.1 shows only the waves for ~10 clock cycles for ~10 data signals which is only ~1% of the data flow. Making it impossible to show the validity of the system in this report this way.

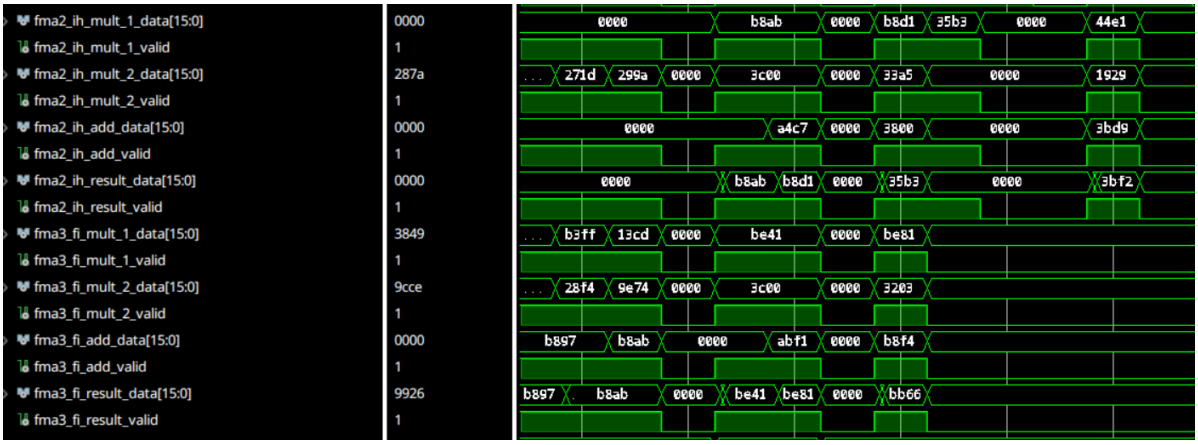


Figure 5.1: Simulation example of time-sequential VHDL code

Therefore, another way has to be decided to show validity in the report. The best way to show the validity of the system is to measure the accuracy of the output of the implementation and compare it to the baseline Bonito GPU implementation and subsequently use the benchmarks by i.e. M. Pagès-Gallego et al. [32] to validate the implementation result in read accuracy compared to other basecaller. However, this is also complicated to achieve because the FPGA implementation only has the implementation of the LSTM layers, while the CNN and linear layers are absent as discussed before. As only a part of the basecalling implementation is implemented, the data would have to be first fed into the GPU implementation then taken out of the GPU implementation, to be then fed into the FPGA implementation, and feed the FPGA back into the GPU implementation to finish it off. As seen during the fixed-point analysis, it proves very difficult to get the data out of the GPU implementation, moreover, the implementations run on different computers adding more difficulty. Therefore, this is also not a viable option.

The decision has fallen on validating the implementation by comparing the implemented design to a CPU-based design and comparing the accuracy between the two. The CPU implementation is a Python script that is modeled to be equivalent to the Bonito LSTM layer implementation but without multithreading, etc. which makes it difficult to work with the LSTM in isolation on the Bonito GPU implementation. The FPGA implementation is run in the Vivado simulator, as running on the FPGA is not possible as mentioned before. While using the Vivado simulator works, it is, however, very slow and requires a lot of compute resources from the host to simulate the implementation. The memory requirement is high due to the implementation having to be compiled into code that can be run in the simulator, while the simulator itself is slow due to the simulator having to keep track of thousands of signals at each time increment. At first, it was tried to simulate the design on a desktop PC with an *AMD Ryzen 7 3700X* with 32 GB DDR4 RAM, however, this was not enough to simulate the design as it ran out of RAM. Fortunately, it is possible to run the Vivado simulator on the Q&CE servers, which provides 254 GB RAM with an *AMD EPYC 7542* CPU. While it is not optimal to perform Vivado simulation on a server CPU compared to a desktop CPU due to it being a very single-threaded process, the amount of RAM on the server allows the simulation to run which is naturally more important. The Python implementation is run on the aforementioned desktop PC.

To validate the implementation, two tests are run: a single-layer inference and a full model inference. The single-layer test is used to evaluate the error induced by one layer alone, while the full-model test evaluates how this error will cascade over the 5 LSTMs. To perform the tests, a chunk sample is extracted from the Bonito model using the variable watch debug tool, additionally, the weights and

biases of pre-trained models are provided in a Pickle file by Oxford Nanopore. This Pickle file can easily be read by Python meaning that the Python implementation has all the data it needs. Getting the input, weight, and bias data into the VHDL simulation is a bit more tricky as VHDL is primarily designed as hardware description language, therefore, performing I/O operations with the simulating PC to load in the input, weight, and bias data is a bit convoluted. The current process to provide the data to the simulation is by converting the data, which is a 16-bit float, to 16-bit hexadecimal characters to make a 4-character hex string. These hex strings are then placed in text files, which are in turn read by the VHDL simulation. These hex strings can be interpreted as an array of bits (std_logic_vector) by VHDL and subsequently cast to a half-precision type. The same process happens in reverse when writing out the results and reading them back into Python to perform statistical analysis.

It was initially planned to infer the entire chunk of 1000 vectors, however, this seemed to be impossible given time constraints due to how slow the simulation is. After testing only one inference, the simulation took around 25 minutes to complete plus 40 minutes of compile time. Therefore, it was opted to go for only 10 inferences per layer. These 10 inferences took 6 hours and 5 minutes in total to complete for one layer, which indicates that the simulation speed slows down over time. For this reason and, additionally, that the Q&CE website states that an X11-Forwarding SSH session will be terminated after 10 hours, it has been chosen to segment the full-model inference over 5 simulations, one for each layer, taking around 30 hours to complete.

The results of single-layer inference can be seen in Figure 5.2 (left). What is interesting to note is that the error of the single-layer seems to correlate with the error of the activation functions as seen in Figure 5.2. Additionally, what is interesting to note when looking into the outliers (above the 99.9% line) is that they had larger outputs on the interval $abs([1.0, 4.0])$ out of one or more of the accumulators. These larger accumulations had already errors around $[10^{-3}, 10^{-2}]$ which suggests accuracy loss due to accumulation. When in the design the accumulator was still split from the multiplier the accumulator accumulated into a larger bit-width which results in a smaller error when accumulating, once the output is requested the accumulation is reduced back to 16-bit. However, with the fused multiply-add this is not possible as the input and output all have to be 16-bit, therefore, offering a lower resolution in accumulation. It seems that both factors influence the error.

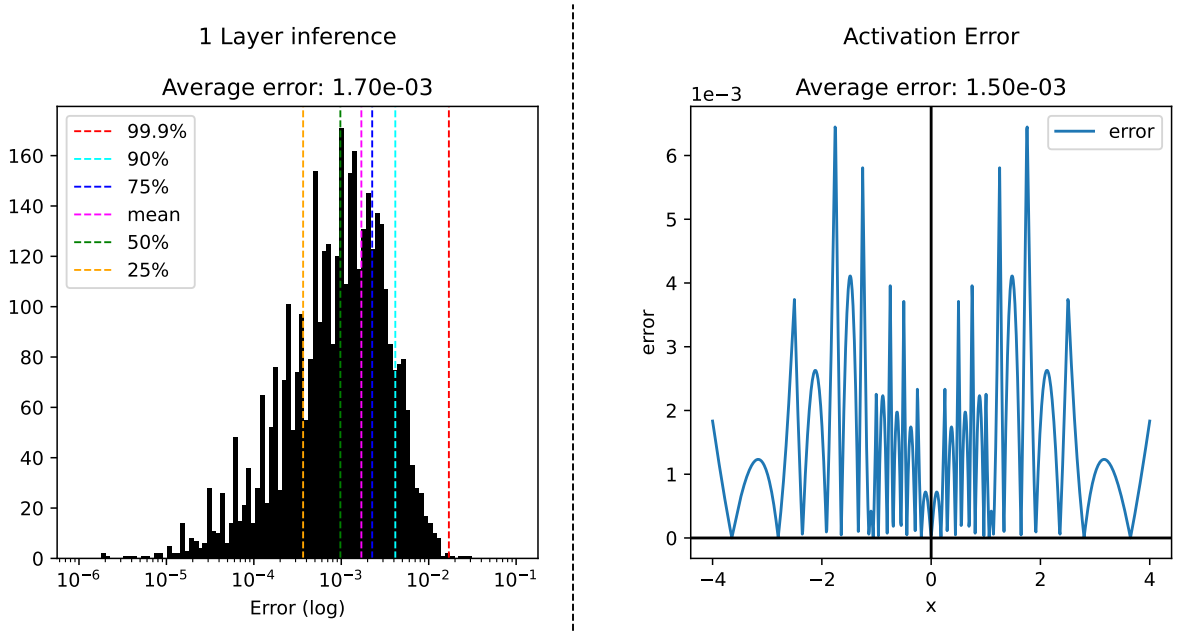


Figure 5.2: Error of 1 layer inference (left) compared to the error of the activation functions (right)

The results of the multi-layer inferences can be seen in Figure 5.3. The figure shows the results of subsequent layer inferences, meaning that the output of the *2 Layer inference* is the input for the *3 Layer inference* and so forth. Surprisingly, the error is not increased accumulative or multiplicative, which one would first expect due to the output with the error being fed into a new layer as input. However, here the

error increases and then decreases in subsequent layers. A possible explanation for this phenomenon is that errors are canceling out over time as all the erroneous samples from the output of the previous layer are recombined in the next layer with the matrix multiplication. This might explain the large error peak in layer 2, which slowly regresses back to the mean activation function error due to more samples being recombined with each subsequent layer inference.

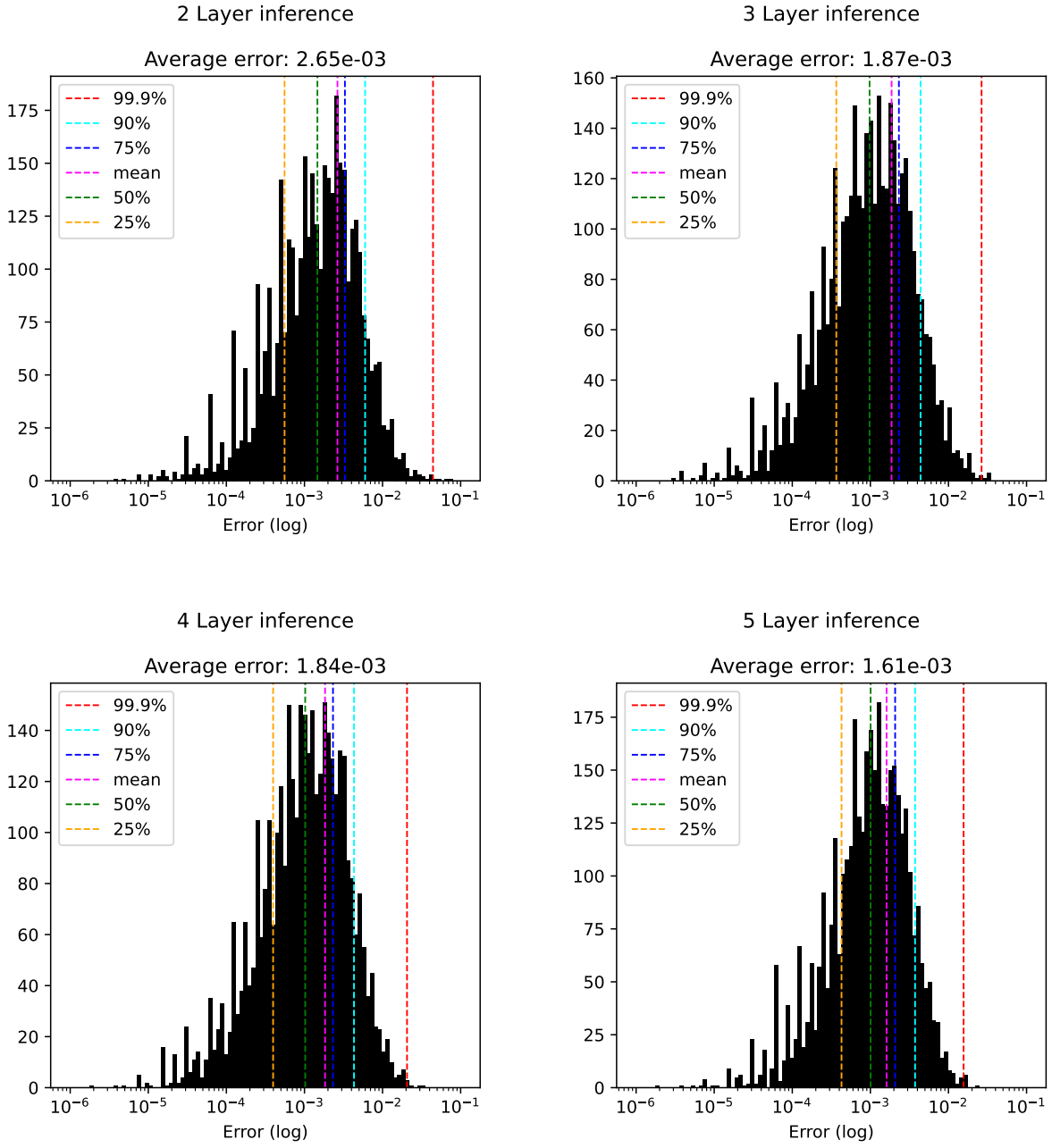


Figure 5.3: Error of multiple layer inferences

6

Discussion and performance projections

In this chapter, the focus is shifted to exploring areas to investigate in the future using the implementation. Potential research directions will be discussed that are beyond the scope of the present work and could offer incremental improvements on the current implementation based on the lessons learned during the thesis project.

6.1. ASIC implementation

As the evaluation has shown implementing LSTMs of this size is not competitive on an FPGA compared to a GPU. However, the implemented design can function as a proof-of-concept to be implemented on an ASIC (application-specific integrated circuit). The clock speed for matching the implementation to the GPU is 4.6 GHz which is too high for the FPGA. While 4.6GHz is also very high in most ASIC designs, the clock speed of an ASIC can most often be 10x higher than an FPGA, therefore, implementing the design on an ASIC will improve the performance over an FPGA.

6.1.1. FPGA hard-copy

Implementing this design on an ASIC is not a fast process and would potentially require additional effort. A relatively easy option is to order an ASIC hard copy of the FPGA implementation. This is a service that takes the FPGA and the implementation of the design and implements it on an ASIC without the reconfigurability of the FPGA, meaning that the ASIC logic cannot change like an FPGA. This conversion service offers quite some benefits with a (hard copy) ASIC over an FPGA:

- **Reduced power consumption**: As mentioned before the logic in an FPGA is implemented with LUTs (lookup tables). As the suggests, this is essentially a read-only memory (ROM) containing the truth tables of the desired logic. While great for reconfigurability, this is intrinsically not as efficient transistor-wise compared to implementing the logic with logic gates at the transistor level because the look-up table requires an address decoder, an output mux, etc. Due to having fewer transistors the power consumption is reduced by a factor of 3 to 6 because there are fewer transistors switching [79][80].
- **Non-volatility**: The ASIC implementation is non-volatile, the reconfigurability is stripped away, the ASIC is also more stable and secure as the design is *"hard coded"* on the die. Conversely, the FPGA has to flash its interconnects (to route signals correctly) and LUTs on power loss by loading the implementation (bitstream) from memory on booting, which takes time. However, more importantly, this configuring can lead to all kinds of security and error-inducing issues [79][81].
- **Unit cost**: Somewhat contradicting to expectations *onsemi*, a company that offers such service, claims a unit price for hard copy ASICs of 25% to 75% compared to the price of the FPGA [79], which is surprising due to fact that it is a custom silicon fabrication. Moreover, FPGAs often have very high pin count which is most often not fully utilized by reducing the pin count for an ASIC to full utilization the packages can be reduced in size making fabrication cheaper [80].

However, these services do not offer any real speed-ups compared to their FPGA implementations which is a little unexpected, as this might be because the ASIC keeps the grid logic floorplanning layout of the FPGA, see Figure 6.1, which is not optimized for specific applications. Moreover, these service providers warn against IP locking which means that your implementation is not directly mappable to FPGA because of proprietary IP utilization in the design [79]. These proprietary IPs have often encrypted source codes which in the best-case scenario can be obtained monetarily or worst-case not at all, meaning that the IP has to be replaced with something with obtainable source code. This thesis implementation is, however, IP-locked due to utilizing the floating-point IPs by Xilinx meaning that these would have to be replaced with a custom implementation.

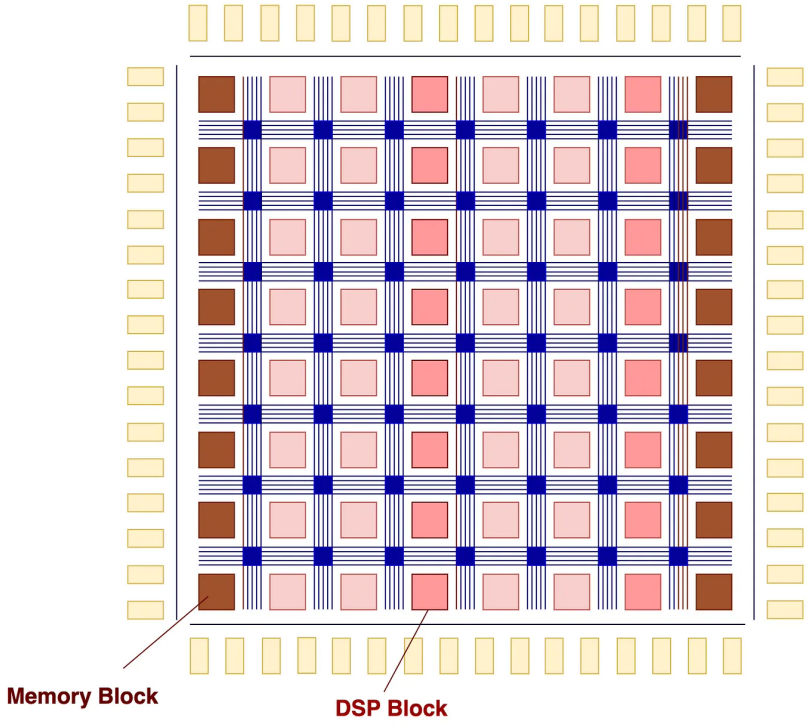


Figure 6.1: Schematic illustration of an FPGA die [82]

6.1.2. Custom ASIC

While mapping the FPGA implementation directly to an ASIC has quick time to market, minus having to create custom floating-point units, it still does not address the speed performance issue compared to the GPU. To address this, there has to be looked more at custom optimizations like routing and die floorplanning to increase clock speed and throughput per cycle to increase performance. However, by looking at the FPGA primitive resource distribution, see Figure 6.2, it is apparent that a relatively small portion of the die is allocated to the DSPs, while Figure 6.2 would probably not be the exact distribution of the Alveo FPGA used in this project, it shows a very typical FPGA resource distribution. In an implementation, like this thesis project, where the throughput is constrained by DSPs, one would conversely want as many DSPs as possible which makes this distribution quite ineffective. However, one could validly state that the LUT utilization in this thesis project implementation is also high (75%), as seen before in the evaluation section. This high LUT utilization is because the DSPs in the Alveo FPGAs cannot perform floating-point operations natively, therefore, the LUTs have to accompany the DSPs to perform floating-point operations which is not hardware efficient. Moreover, the DSPs are also capable of performing features that are not utilized in this implementation thus resulting in unutilized transistors. By designing an ASIC the arithmetic units can be completely designed to be specialized in performing fused multiply-add operations in floating-point, which would use significantly fewer transistors.

Combining these insights, the conclusion can be drawn that an FPGA is not laid out well enough

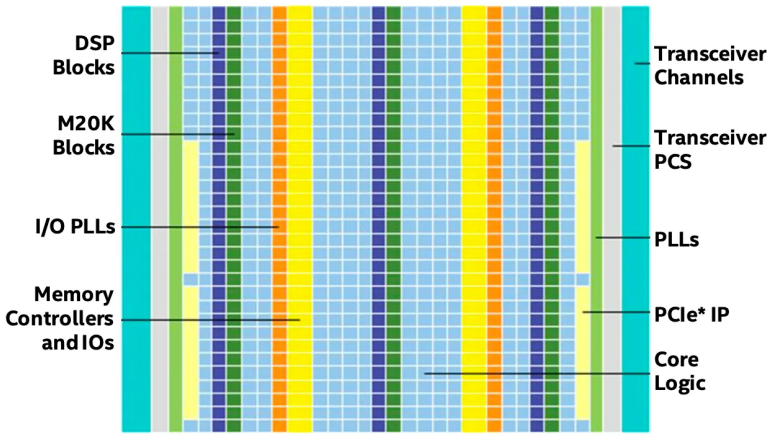


Figure 6.2: Typical primitive resource distribution of an FPGA die [82]

to optimally perform the LSTM calculations, leaving quite some potential optimizations. However, the additional transistors that become available due to the optimizations can not be directly assigned to the implementation and automatically make the design more performant. Therefore, there will now be a look at how to utilize these transistors by exploring architectural changes to accommodate them. Before performing further analysis on the ASIC design to obtain some preliminary result estimation, it must be stated that assumptions about the transistor performance increases have to be made. This is because there is no formula to estimate the area utilization (transistor usage) in converting an FPGA implementation to an ASIC implementation. Additionally, the optimization is on a spectrum where on the one side with low effort there is the hard copy, while optimizing further it is naturally pushed further away from the hard copy but by requiring more time and using more design budget. The hard copy can give a baseline to estimate transistor utilization improvements. The hard copy service providers state that the power consumption is improved 3 to 6 times compared to the FPGA without stating any speed differences while stating that this reduction is coming from replacing LUTs with logic gates. Suppose it is assumed that the power consumption scales linearly with the transistor count, it can be assumed that just by doing a hard copy there are potentially 3 to 6 times more transistors available. With already 5 times more transistors one can potentially pipeline the entire 5 layers of the model, see Figure 6.3. By fully pipelining the design, 5 layers can be calculated at the same time. Moreover, the fully pipelined implementation also eliminates the need for any additional external DRAM as everything can be stored on-chip. Fully pipelining the design also simplifies the entire architecture as input/output data does not have to be buffered only the hidden state and the cell state. Moreover, the states and weights do not have to be switched out every layer switch, and the weights data does not have to be double buffered. Naturally, storing all five layer weights on-chip increases the SRAM utilization but this only occupies 11.85 MB with each weight memory taking up 2.37 MB. Moreover, due to the design being an ASIC the memory utilization can be better optimized for the data width of the application, leading to better memory utilization than in the FPGA implementation. Moreover, SRAMs can take up a lot of chip area so optimizing the SRAM to be as close to 12 MB as possible can leave more area for floating point units. Finally, the on-chip memory can also be implemented as read-only memories (ROMs) which is even more efficient than SRAMs as the weight memories do not have to be written to. However, using ROMs will sacrifice changing the weights, for instance when Oxford Nanopore offers newly trained weights.

Additionally, by looking at the resource distribution as seen in Figure 6.2, say with some optimization and making dedicated floating point units, with some optimism, the transistor count increases to 10 fold instead of 5. Looking back to the matrix-vector multiplication optimization, see Equation 6.1, in addition to pipelining, two parallel accumulations per layer are possible, as seen in Equation 6.2. The parallel accumulations can be added together in the post, leading only to two extra cycles in the post-processing, one extra for the input and one for the hidden input, while being twice as fast in accumulation.

$$\left[\begin{array}{c|c|c|c} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{array}\right] \times \left[\begin{array}{c} w \\ \hline x \\ \hline y \\ \hline z \end{array}\right] = \left[\begin{array}{c|c|c|c|c|c|c|c} a \times w & + & b \times x & + & c \times y & + & d \times z \\ e \times w & + & f \times x & + & g \times y & + & h \times z \\ i \times w & + & j \times x & + & k \times y & + & l \times z \end{array}\right] \tag{6.1}$$

$$\left[\begin{array}{cc|cc} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{array}\right] \times \left[\begin{array}{c} w \\ \hline x \\ \hline y \\ \hline z \end{array}\right] = \left[\begin{array}{c|c|c} a \times w & + & c \times y \\ e \times w & + & g \times y \\ i \times w & + & k \times y \end{array}\right] + \left[\begin{array}{c|c|c} b \times x & + & d \times z \\ f \times x & + & h \times z \\ j \times x & + & l \times z \end{array}\right] \tag{6.2}$$

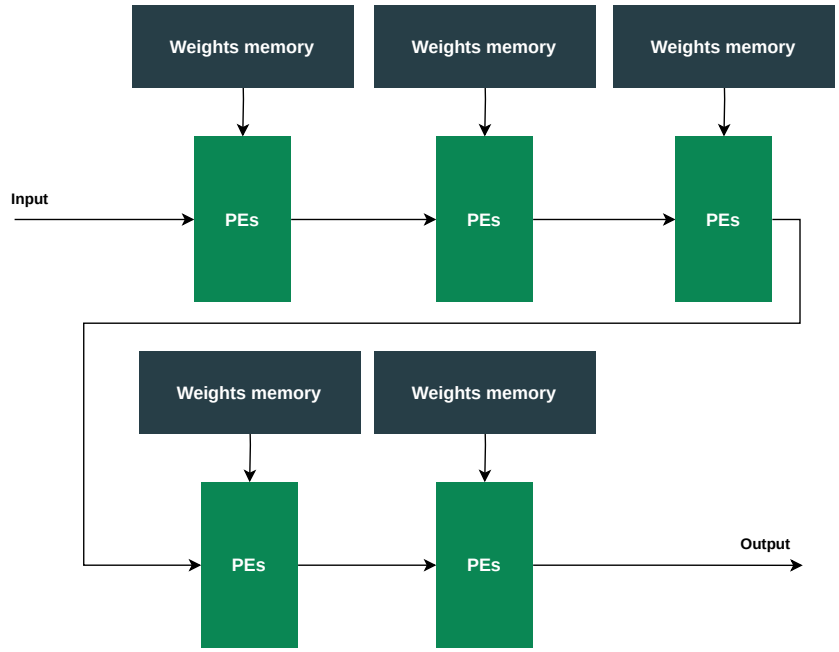


Figure 6.3: A fully pipelined Bonito implementation

Moreover, due to having more efficient (shorter) signal paths due to optimizing the transistor usage, additionally, in an ASIC the clock frequency can be increased significantly compared to an FPGA. This is because each clock cycle the signals have to go through fewer transistors and each transistor adds a delay to the signal. While an FPGA has clock speeds around 100 to 200 MHz, it is common for ASICs to have clock speeds 10x higher in the of 1 to 2 GHz [83]. Combining that with the fact of pipelining and parallel accumulations a speed up of 100x can be achieved over the FPGA implementation. Looking back at the performance evaluation, the computational throughput of the FGPA with 1.13 TFLOPS/s, theoretically, the ASIC can achieve around 113 TFLOPS/s which vastly outperforms the benchmark GPU (RTX 2080Ti) of 28 TFLOPS/s. However, when looking at NVIDIA A100 which are the GPUs paired by Oxford Nanopore with their sequencers, that GPU can still theoretically perform 156 TFLOPS/s [84] showing how performant high-end GPUs are.

While the ASIC implementation shows potential, this is, as mentioned before, a preliminary look at such an implementation. Moreover, ASIC implementations are not without monetary risks. Manufacturing an ASIC is already a large investment resulting in costs of 10 to 100 million dollars [85]. Such investments are naturally risky and require good market evaluation to validate if that investment can be returned. Moreover, the design process of an ASIC is significantly longer due to validation because if there is a bug in your hardware there is no way of fixing it once manufactured, losing potentially a lot of money. Conversely, a bug in an FPGA implementation can be fixed with an update by reconfiguring the hardware [83]. However, by analyzing the price-per-unit the ASIC shows potential over an A100 GPU. By calculating using similar silicon wafer properties as an A100 GPU, a preliminary analysis of the price-per-unit can be performed.

Integrated circuits (IC), like an ASIC, are made from a wafer which is a circular slice of semiconductor material (typically crystalline silicon) and by using photolithography [86]. From such a silicon wafer, multiple rectangular IC dies are made, see Figure 6.4. By utilizing the data from TSMC (one of the largest silicon manufacturers), at the time of writing these silicon wafers have a diameter of 300mm (~12 inches) [87] with a reticle limit of 858mm² [88] which indicate the maximum area of an IC die. An NVIDIA A100 GPU has a die size of 826mm² [60], therefore, in this analysis, the same value of 826mm² using square dies is chosen since the comparison is against the A100 GPU as it is used by Oxford Nanopore.

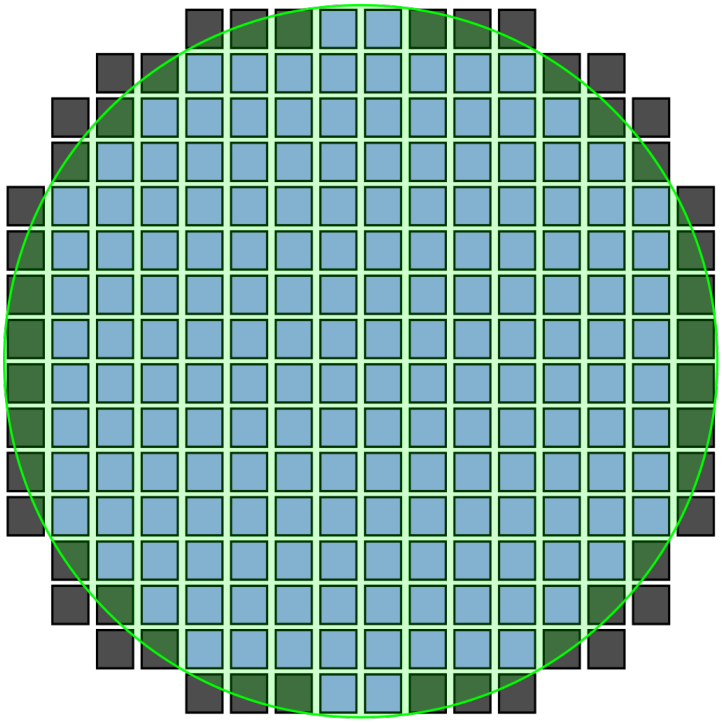


Figure 6.4: Abstract representation of a silicon wafer (circle) and silicon dies (squares) [89]

However, as seen in Figure 6.4, obviously it is not possible to fully utilize the circular area of the wafer for the dies since the dies are rectangular. The dies per wafer (DPW) can subsequently be calculated by utilizing Equation 6.3 [90], where d is the wafer diameter and S is the area of the dies. Using Equation 6.3, this is approximated to result in 66 dies using a 300mm wafer and 826mm² dies. A 300mm silicon wafer from TSMC using 3nm technology costs around $20,000 [91] which allows for calculating the price per die. However, before calculating the price per die, it is important to note that the die yield is a considerable factor, as with TSMC's 3nm manufacturing process the yield is reported to be about 55% [92]. Accounting for all these parameters, the price per die would be around $550. To get to the final ASIC the die has to be packaged, tested, etc., therefore, a margin of double the die cost is taken as an estimation to account for that, resulting in a price-per-unit of $1,100. Comparing this price-per-unit to an NVIDIA A100 of around $45,000 shows that in terms of cost, the ASIC implementation shows a lot of potential. However, as mentioned before, sales volume is very important as silicon manufacturers naturally do not make wafers in small volumes, nevertheless, given the price difference and interest in genomics, it is not unusual to assume a large volume of i.e. 100,000 units.

$$DPW = \frac{\pi d^2}{4S} - 0.58\frac{\pi d}{\sqrt{S}} \tag{6.3}$$

6.2. Fixed-point analysis

While the possibility of fixed-point arithmetic instead of floating-point has already been explored at the beginning of the design phase of the project, it is still a possible improvement over floating-point in this

implementation. Due to time constraints, the validation of a possible fixed-point implementation could not be finished, in the current state, it still shows potential as it does not seem that this implementation needs the large dynamic range of floating-points. In the discussion of choosing a data format of Section 4.3.1, it was already shown that the weight quantization has a low error, in the order of 10^{-5}. Combining that quantization error, with the observed error as seen in Section 5.3 evaluating the accuracy of the floating-point LSTM implementation which is 2 orders of magnitude larger, we estimate that the impact of using fixed-point is low to marginal due to the effect of the activation functions. Because the fixed-point arithmetic units are more efficient to implement than floating-point units [93], it should be possible to increase the computational throughput this way further in an ASIC than using floating-point. Moreover, even for the FPGA implementation, the DSPs on the FPGA can perform multiple integer operations simultaneously per DPS and only require 1 DSP fused multiply-add over 3 for the floating-point, meaning that the computational throughput could be much higher. Moreover, 1 DSP in fixed-point arithmetic can perform 3 fixed-point computations at the same time, further increasing computational throughput. In total, this would allow for 9x computational throughput improvement over a floating-point implementation, thus theoretically also a 9x improvement over an NVIDIA A100 GPU when this fixed-point implementation would be implemented on an ASIC. However, the first step is to validate the Bonito model with fixed-point quantization which is, as discussed before in Section 4.3.1, not trivial to do.

6.3. Activation functions

As seen with the evaluations, the error of the layer outputs seems to correlate with the error of the activation functions. However, due to the limited amount of resources on the FPGA, any other implementation than lookup tables (LUT) and piece-wise linearization (PWL) was not possible, as the PWL method could reuse hardware and the LUT methods just the FPGA LUTs. However, as seen in Figure 6.5, there is no more potential in PWL as having more pieces does not lower the error anymore. However, in the case of an ASIC with more resources, it is possible to delve into more accurate activation function implementations, like CORDIC, etc. as discussed in Section 4.3.1 on implementing activation functions. With more accurate activation functions, it will also be interesting to see if the error is actually correlated to the activation function or if some other factors are at play.

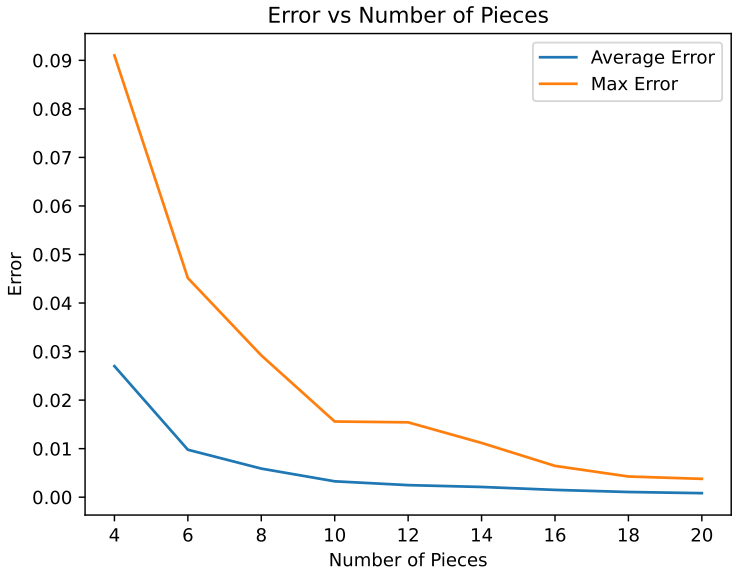


Figure 6.5: The error due to the piece-wise linearization against the number of utilized pieces

6.4. FPGA PCIe interface

In order to perform I/O externally on AMD Alveo Data Center Cards, the use of the PCIe interface is required. However, due to time constraints, this PCIe interface is not implemented. Moreover, the FPGA implementation does not fit on the available FPGAs at the Q&CE clusters at TU Delft, therefore, the implementation was not run on any physical device, it was only simulated. Due to it not being run physically, the PCIe interface addition was dropped for other efforts on the thesis as it is only an interface to communicate outside the FPGA and is not a core functionality of the FPGA implementation. If someone wants to further develop the FPGA implementation, i.e. for the U250, the PCIe interface has to be implemented, however, since the performance on the FPGA is not as performant as the GPU implementation it is not really recommended to further develop it unless better optimizations are possible on the FPGA implementation. The current implementation has an AXI4 streaming interface for the external I/O, so adding the PCIe interface probably only requires using the PCIe IP from Xilinx and maybe some FIFO IPs for buffering.

7

Conclusions and recommendations

7.1. Conclusions

This thesis aims to find an implementation of an accelerated hardware solution for the compute-intensive process of basecalling. The thesis lays the foundation of an accelerator for basecalling nanopore sequenced long reads. As identified during analyzing those Oxford Nanopore long-read pipelines, the basecalling showed to be the main computational bottleneck in both time and accuracy. While great improvements in basecalling accuracy have been made over time by developing and training newer models, there seemed to be little focus on the computational throughput aspects. After selecting a model and analyzing that model, it became apparent that the LSTM (Long short-term memory) layers of the machine learning models were taking up around 90% of computation time, making the thesis shift more to accelerating those LSTM layers over accelerating the whole basecalling network. The support of LSTM layers on FPGA through libraries, etc. is very little or not in usable states, therefore, a from-scratch implementation in VHDL was chosen which enables more control in optimization but also makes developing it more difficult.

While outlining and researching LSTM implementations on FPGA, it was difficult to find LSTM implementation of the size and amount of layers used in the Bonito model. Most of the LSTM implementations in literature were only for small layers and few implementations were available, indicating on-chip memory as a limitation. Therefore, validating the performance and capacity of on-chip and off-chip memory was crucial before starting such an implementation. However, after implementing the arithmetic units, contrary to expectations, the memory was not the limiting factor but the amount of arithmetic units that could fit on the FPGA was. The limitation of arithmetic units is mainly because the arithmetic accelerators on the FPGA cannot natively perform floating-point operations, meaning that multiple of these accelerators and reconfigurable logic have to be combined to facilitate a floating-point unit. This severely limits the possibility of performing more optimizations on the FPGA regarding the arithmetic units on the FPGA which made it impossible to create an FPGA implementation that is competitive to even the benchmark GPU (NVIDIA RTX 2080Ti), while that GPU is not even the best GPU on the market. This means that running the LSTM on FPGAs is not feasible as it does not offer any real benefit over the GPU implementation. The GPU implementation has better integration with the host system (i.e. PyTorch), better performance, and comparable energy consumption.

However, while the FPGA implementation itself is not competitive, it can serve as a starting point for developing an application-specific integrated circuit (ASIC) for Bonito LSTM inferences or even the whole Bonito model. The main problems with the FPGA are that it has no floating point units and the FPGA primitive resources are not well laid out to facilitate maximum computational throughput. In an ASIC, these elements can be optimized allowing for higher computational throughput making it, with a preliminary theoretical analysis, comparable to a high-end GPU like an NVIDIA A100. Moreover, utilizing fixed-point arithmetic over floating-point makes the ASIC up to 9 times faster than an A100 GPU but due to issues with validation, it is not certain yet if fixed-point quantization will lead to too much accuracy loss. However, developing an ASIC from this design is not a trivial effort requiring additional design and validation effort. Moreover, manufacturing an ASIC has a high monetary investment with more risks than an FPGA implementation due to hardware bugs and potential loss of investment. Nevertheless,

analyzing the potential price-per-unit of such an ASIC implementation, the estimated price-per-unit is $1,100 which is significantly less than an NVIDIA A100 GPU costing around $40,000.

7.2. Recommendation

Based on the findings of this thesis the following are recommendations for future work:

- Further exploring the possibilities of implementing the Bonito model on an ASIC as discussed in Section 6.1.
- Fully validate fixed-point quantization possibilities as discussed in Section 4.3.1 and 6.2.
- Further look into the activation functions as the activation functions in the current implementation functions seem to be the main contributor of accuracy loss, see Section 4.3.1 and 6.3.

Bibliography

[1] Miguel García-Sancho and James Lowe. *A History of Genomics across Species, Communities and Projects*. Springer International Publishing, 2023. DOI: 10.1007/978-3-031-06130-1. URL: https://doi.org/10.1007/978-3-031-06130-1.

[2] Sam Kovaka et al. “Approaching complete genomes, transcriptomes and epi-omes with accurate long-read sequencing”. In: *Nature Methods* 20.1 (Jan. 2023), pp. 12–16. DOI: 10.1038/s41592-022-01716-8. URL: https://doi.org/10.1038/s41592-022-01716-8.

[3] Shanika L. Amarasinghe et al. “Opportunities and challenges in long-read sequencing data analysis”. In: *Genome Biology* 21.1 (Feb. 2020). DOI: 10.1186/s13059-020-1935-5. URL: https://doi.org/10.1186/s13059-020-1935-5.

[4] Sergey Nurk et al. “The complete sequence of a human genome”. In: *Science* 376.6588 (Apr. 2022), pp. 44–53. DOI: 10.1126/science.abj6987. URL: https://doi.org/10.1126/science.abj6987.

[5] Vivien Marx. “Method of the year: long-read sequencing”. In: *Nature Methods* 20.1 (Jan. 2023), pp. 6–11. DOI: 10.1038/s41592-022-01730-w. URL: https://doi.org/10.1038%5C%2Fs41592-022-01730-w.

[6] Shanika L. Amarasinghe et al. “Opportunities and challenges in long-read sequencing data analysis”. In: *Genome Biology* 21.1 (Feb. 2020). DOI: 10.1186/s13059-020-1935-5. URL: https://doi.org/10.1186%5C%2Fs13059-020-1935-5.

[7] Peter E. Warburton and Robert P. Sebra. “Long-Read DNA Sequencing: Recent Advances and Remaining Challenges”. In: *Annual Review of Genomics and Human Genetics* 24.1 (Aug. 2023), pp. 109–132. DOI: 10.1146/annurev-genom-101722-103045. URL: https://doi.org/10.1146%5C%2Fannurev-genom-101722-103045.

[8] Jared T Simpson et al. “Detecting DNA Methylation using the Oxford Nanopore Technologies MinION sequencer”. In: (Apr. 2016). DOI: 10.1101/047142. URL: https://doi.org/10.1101/047142.

[9] S.A. Miller, D.D. Dykes, and H.F. Polesky. “A simple salting out procedure for extracting DNA from human nucleated cells”. In: *Nucleic Acids Research* 16.3 (1988), pp. 1215–1215. DOI: 10.1093/nar/16.3.1215. URL: https://doi.org/10.1093/nar/16.3.1215.

[10] Ruta Sahasrabudhe. *High molecular weight DNA isolation (HMW-DNA)*. URL: https://dnatech.genomecenter.ucdavis.edu/high-molecular-weight-dna-isolation-hmw-dna/.

[11] Oxford Nanopore Technologies. *Library preparation to sequence any fragment length*. May 2023. URL: https://nanoporetech.com/library-preparation.

[12] Morgan MacKenzie and Christos Argyropoulos. “An Introduction to Nanopore Sequencing: Past, Present, and Future Considerations”. In: *Micromachines* 14.2 (Feb. 2023), p. 459. DOI: 10.3390/mi14020459. URL: https://doi.org/10.3390/mi14020459.

[13] Oxford Nanopore Technologies. *How basecalling works*. Apr. 2023. URL: https://nanoporetech.com/how-it-works/basecalling.

[14] Yoritaka Harazono. *Principle of Nanopore DNA Sequencing*. 2020. DOI: 10.7875/togopic.2020.01. URL: https://dbarchive.biosciencedbc.jp/data/togo-pic/image/202001_nanopore_sequencing.svg.

[15] Jared T Simpson et al. “Detecting DNA Methylation using the Oxford Nanopore Technologies MinION sequencer”. In: (Apr. 2016). DOI: 10.1101/047142. URL: https://doi.org/10.1101/047142.

[16] Oxford Nanopore Technologies. *Promethion*. Apr. 2023. URL: https://nanoporetech.com/products/promethion.

[17] Michael Schatz. URL: https://twitter.com/mike_schatz/status/1630649605632172034?s=20.

[18] NVIDIA. *Nvidia A100 GPUs: power the modern data center*. URL: https://www.nvidia.com/en-us/data-center/a100/.

[19] Peter Krusche et al. “Best practices for benchmarking germline small-variant calls in human genomes”. In: *Nature Biotechnology* 37.5 (Mar. 2019), pp. 555–560. DOI: 10.1038/s41587-019-0054-x. URL: https://doi.org/10.1038/s41587-019-0054-x.

[20] Mian Umair Ahsan et al. “NanoCaller for accurate detection of SNPs and indels in difficult-to-map regions from long-read sequencing by haplotype-aware deep neural networks”. In: *Genome Biology* 22.1 (Sept. 2021). DOI: 10.1186/s13059-021-02472-2. URL: https://doi.org/10.1186/s13059-021-02472-2.

[21] Ying Chen et al. “Efficient assembly of nanopore reads via highly accurate and intact error correction”. In: *Nature Communications* 12.1 (Jan. 2021). DOI: 10.1038/s41467-020-20236-7. URL: https://doi.org/10.1038/s41467-020-20236-7.

[22] IBM. *What are recurrent neural networks?* URL: https://www.ibm.com/topics/recurrent-neural-networks.

[23] Awni Hannun. “Sequence Modeling with CTC”. In: *Distill* (2017). https://distill.pub/2017/ctc. DOI: 10.23915/distill.00008.

[24] Xuan Lv et al. “An End-to-end Oxford Nanopore Basecaller Using Convolution-augmented Transformer”. In: *2020 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, Dec. 2020. DOI: 10.1109/bibm49941.2020.9313290. URL: https://doi.org/10.1109/bibm49941.2020.9313290.

[25] Ryan R. Wick, Louise M. Judd, and Kathryn E. Holt. “Performance of neural network basecalling tools for Oxford Nanopore sequencing”. In: (Feb. 2019). DOI: 10.1101/543439. URL: https://doi.org/10.1101/543439.

[26] Haotian Teng et al. “Chiron: translating nanopore raw signal directly into nucleotide sequence using deep learning”. In: *GigaScience* 7.5 (Apr. 2018). DOI: 10.1093/gigascience/giy037. URL: https://doi.org/10.1093/gigascience/giy037.

[27] Harald Scheidl. *An intuitive explanation of connectionist temporal classification*. July 2021. URL: https://towardsdatascience.com/intuitively-understanding-connectionist-temporal-classification-3797e43a86c.

[28] Oxford Nanopore Technology. *Flappie*. Aug. 2020. URL: https://github.com/nanoporetech/flappie.

[29] Oxford Nanopore Technology. *Scrappie*. Aug. 2019. URL: https://github.com/nanoporetech/flappie.

[30] Oxford Nanopore Technology. *Bonito*. Aug. 2023. URL: https://github.com/nanoporetech/bonito.

[31] Oxford Nanopore Technology. *Dorado*. Aug. 2023. URL: https://github.com/nanoporetech/dorado.

[32] Marc Pagès-Gallego and Jeroen de Ridder. “Comprehensive benchmark and architectural analysis of deep learning models for nanopore sequencing basecalling”. In: *Genome Biology* 24.1 (Apr. 2023). DOI: 10.1186/s13059-023-02903-2. URL: https://doi.org/10.1186/s13059-023-02903-2.

[33] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. “Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data”. In: *Proceedings of the Eighteenth International Conference on Machine Learning*. ICML ’01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 282–289. ISBN: 1-55860-778-1. URL: http://dl.acm.org/citation.cfm?id=645530.655813.

[34] Zhimeng Xu et al. “Fast-Bonito: A Faster Basecaller for Nanopore Sequencing”. In: (Oct. 2020). DOI: 10.1101/2020.10.08.318535. URL: https://doi.org/10.1101/2020.10.08.318535.

[35] Jingwen Zeng et al. “Causalcall: Nanopore Basecalling Using a Temporal Convolutional Network”. In: *Frontiers in Genetics* 10 (Jan. 2020). DOI: 10.3389/fgene.2019.01332. URL: https://doi.org/10.3389/fgene.2019.01332.

[36] Neven Miculinić, Marko Ratković, and Mile Šikić. *MinCall - MinION end2end convolutional deep learning basecaller*. 2019. DOI: 10.48550/ARXIV.1904.10337. URL: https://arxiv.org/abs/1904.10337.

[37] Yao-zhong Zhang et al. “Nanopore basecalling from a perspective of instance segmentation”. In: *BMC Bioinformatics* 21.S3 (Apr. 2020). DOI: 10.1186/s12859-020-3459-0. URL: https://doi.org/10.1186/s12859-020-3459-0.

[38] Rick Merritt. *What is a transformer model?* Sept. 2022. URL: https://blogs.nvidia.com/blog/2022/03/25/what-is-a-transformer-model/.

[39] Charu C. Aggarwal. “Recurrent Neural Networks”. In: *Neural networks and deep learning: A textbook*. Springer, 2018, pp. 271–310.

[40] Ashish Vaswani et al. *Attention Is All You Need*. 2017. DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762.

[41] 2021. URL: https://rentruewang.github.io/learning-machine/layers/transformer/transformer-vs-rnn.html.

[42] Neng Huang et al. “SACall: A Neural Network Basecaller for Oxford Nanopore Sequencing Data Based on Self-Attention Mechanism”. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 19.1 (Jan. 2022), pp. 614–623. DOI: 10.1109/tcbb.2020.3039244. URL: https://doi.org/10.1109/tcbb.2020.3039244.

[43] Xuan Lv et al. “An End-to-end Oxford Nanopore Basecaller Using Convolution-augmented Transformer”. In: (Nov. 2020). DOI: 10.1101/2020.11.09.374165. URL: https://doi.org/10.1101/2020.11.09.374165.

[44] Zhanghao Wu et al. *Lite Transformer with Long-Short Range Attention*. 2020. DOI: 10.48550/ARXIV.2004.11886. URL: https://arxiv.org/abs/2004.11886.

[45] Sean R Eddy. “What is a hidden Markov model?” In: *Nature Biotechnology* 22.10 (Oct. 2004), pp. 1315–1316. DOI: 10.1038/nbt1004-1315. URL: https://doi.org/10.1038%5C%2Fnbt1004-1315.

[46] Matei David et al. “Nanocall: an open source basecaller for Oxford Nanopore sequencing data”. In: *Bioinformatics* 33.1 (Sept. 2016). Ed. by Inanc Birol, pp. 49–55. DOI: 10.1093/bioinformatics/btw569. URL: https://doi.org/10.1093%5C%2Fbioinformatics%5C%2Fbtw569.

[47] Johan Peltenburg et al. “Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow”. In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Sept. 2019. DOI: 10.1109/fpl.2019.00051. URL: https://doi.org/10.1109%5C%2Ffpl.2019.00051.

[48] Oxford Nanopore Technology. *POD5 File Format*. Aug. 2023. URL: https://github.com/nanoporetech/pod5-file-format.

[49] Ali Al-Safwan, Chao Song, and Umair bin Waheed. *Is it time to swish? Comparing activation functions in solving the Helmholtz equation using physics-informed neural networks*. 2021. DOI: 10.48550/ARXIV.2110.07721. URL: https://arxiv.org/abs/2110.07721.

[50] davidcpage. *Question about the crf model*. Jan. 2021. URL: https://github.com/nanoporetech/bonito/issues/101.

[51] Michaela Blott et al. *FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks*. 2018. DOI: 10.48550/ARXIV.1809.04570. URL: https://arxiv.org/abs/1809.04570.

[52] Xilinx. *AMD Vitis AI: Adaptable Real-Time AI Inference Acceleration*. Aug. 2023. URL: https://github.com/Xilinx/Vitis-AI.

[53] AMD. *VCK5000 Versal Development Card*. URL: https://www.xilinx.com/products/boards-and-kits/vck5000.html#specs.

[54] Hao Wang et al. “Implementation of Bidirectional LSTM Accelerator Based on FPGA”. In: *2022 IEEE 22nd International Conference on Communication Technology (ICCT)*. IEEE, Nov. 2022. DOI: 10.1109/icct56141.2022.10072756. URL: https://doi.org/10.1109%5C%2Ficct56141.2022.10072756.

[55] Weifeng Zhang et al. “Design and Implementation of LSTM Accelerator Based on FPGA”. In: *2020 IEEE 20th International Conference on Communication Technology (ICCT)*. IEEE, Oct. 2020. DOI: 10.1109/icct50939.2020.9295665. URL: https://doi.org/10.1109%5C%2Ficct50939.2020.9295665.

[56] Aurélien Géron. “Recurrent Neural Networks”. In: *Neural networks and deep learning*. O’Reilly Media, Inc., Mar. 2018.

[57] Guillaume Chevalier. *LSTM Cell*. 2018. URL: https://upload.wikimedia.org/wikipedia/commons/1/17/The_LSTM_Cell.svg.

[58] AMD. *Alveo U250 Data Center Accelerator Card*. URL: https://www.xilinx.com/products/boards-and-kits/alveo/u250.html#specifications.

[59] AMD. *AMD Ryzen™ Threadripper™ PRO 5995WX Processor*. URL: https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-pro-5995wx.

[60] Ronny Krashinsky et al. *NVIDIA Ampere Architecture In-Depth*. URL: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/.

[61] AMD. *Alveo U280 Data Center Accelerator Card*. URL: https://www.xilinx.com/products/boards-and-kits/alveo/u280.html#specifications.

[62] Hao Wang et al. “Implementation of Bidirectional LSTM Accelerator Based on FPGA”. In: *2022 IEEE 22nd International Conference on Communication Technology (ICCT)*. IEEE, Nov. 2022. DOI: 10.1109/icct56141.2022.10072756. URL: https://doi.org/10.1109/icct56141.2022.10072756.

[63] AMD. *AMBA AXI4 Interface Protocol*. URL: https://www.xilinx.com/products/intellectual-property/axi.html.

[64] *IEEE Standard for Floating-Point Arithmetic*. DOI: 10.1109/ieeestd.2019.8766229. URL: https://doi.org/10.1109/ieeestd.2019.8766229.

[65] Wenzhe Zhao et al. “Optimizing FPGA-Based DNN Accelerator With Shared Exponential Floating-Point Format”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2023), pp. 1–14. DOI: 10.1109/tcsi.2023.3300657. URL: https://doi.org/10.1109/tcsi.2023.3300657.

[66] Xilinx. *UltraScale Architecture DSP Slice*. URL: https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp.

[67] PyTorch Contributors. *Quantization*. URL: https://pytorch.org/docs/stable/quantization.html.

[68] Xilinx. *Performance and Resource Utilization for Floating-point v7.1*. URL: https://www.xilinx.com/htmldocs/ip_docs/pru_files/floating-point.html.

[69] Safa Bouguezzi, Hassene Faiedh, and Chokri Souani. “Hardware Implementation of Tanh Exponential Activation Function using FPGA”. In: *2021 18th International Multi-Conference on Systems, Signals & Devices (SSD)*. IEEE, Mar. 2021. DOI: 10.1109/ssd52085.2021.9429506. URL: https://doi.org/10.1109/ssd52085.2021.9429506.

[70] Peter W. Zaki et al. “A Novel Sigmoid Function Approximation Suitable for Neural Networks on FPGA”. In: *2019 15th International Computer Engineering Conference (ICENCO)*. IEEE, Dec. 2019. DOI: 10.1109/icenco48310.2019.9027479. URL: https://doi.org/10.1109/icenco48310.2019.9027479.

[71] Jack E. Volder. “The CORDIC Trigonometric Computing Technique”. In: *IRE Transactions on Electronic Computers* EC-8.3 (Sept. 1959), pp. 330–334. DOI: 10.1109/tec.1959.5222693. URL: https://doi.org/10.1109%5C%2Ftec.1959.5222693.

[72] Hui Chen et al. “A CORDIC-Based Architecture with Adjustable Precision and Flexible Scalability to Implement Sigmoid and Tanh Functions”. In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, Oct. 2020. DOI: 10.1109/iscas45731.2020.9180864. URL: https://doi.org/10.1109%5C%2Fiscas45731.2020.9180864.

[73] Manal T. Ali and Bassam H. Abd. “An Efficient area Neural Network Implementation using tan-sigmoid Look up Table Method Based on FPGA”. In: *2022 3rd International Conference for Emerging Technology (INCET)*. IEEE, May 2022. DOI: 10.1109/incet54531.2022.9825348. URL: https://doi.org/10.1109/incet54531.2022.9825348.

[74] Revathi Pogiri, Samit Ari, and K K Mahapatra. “Design and FPGA Implementation of the LUT based Sigmoid Function for DNN Applications”. In: *2022 IEEE International Symposium on Smart Electronic Systems (iSES)*. IEEE, Dec. 2022. DOI: 10.1109/ises54909.2022.00090. URL: https://doi.org/10.1109/ises54909.2022.00090.

[75] Zerun Li et al. “FPGA Implementation for the Sigmoid with Piecewise Linear Fitting Method Based on Curvature Analysis”. In: *Electronics* 11.9 (Apr. 2022), p. 1365. DOI: 10.3390/electronics11091365. URL: https://doi.org/10.3390/electronics11091365.

[76] Ivan Tsmots, Oleksa Skorokhoda, and Vasyl Rabyk. “Hardware Implementation of Sigmoid Activation Functions using FPGA”. In: *2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM)*. IEEE, Feb. 2019. DOI: 10.1109/cadsm.2019.8779253. URL: https://doi.org/10.1109/cadsm.2019.8779253.

[77] Adrian Sampson. *From Hardware Description Languages to Accelerator Design Languages*. URL: https://www.sigarch.org/hdl-to-adl/.

[78] Johan Peltenburg et al. “Tydi: An Open Specification for Complex Data Structures Over Hardware Streams”. In: *IEEE Micro* 40.4 (July 2020), pp. 120–130. DOI: 10.1109/mm.2020.2996373. URL: https://doi.org/10.1109%5C%2Fmm.2020.2996373.

[79] Tekmos. *FPGA Conversions*. URL: https://www.tekmos.com/products/asics/fpga-conversions.

[80] Onsemi. *FPGA-to-ASIC Conversion*. URL: https://www.onsemi.com/products/custom-assp/soc-sip-custom-products/fpga-to-asic-conversion#benefits.

[81] Simon Watson. *FPGAs: Security Through Obscurity?* Dec. 2021. URL: https://research.nccgroup.com/2021/12/14/fpgas-security-through-obscurity/.

[82] Jonathan Hui. *AI chips: FPGA*. Dec. 2020. URL: https://jonathan-hui.medium.com/ai-chips-fpga-875093ab0a7d.

[83] Gisselquist Technology LLC. *FPGAs vs ASICs*. Oct. 2017. URL: https://zipcpu.com/blog/2017/10/13/fpga-v-asic.html.

[84] NVIDIA. *GPU Performance Background User’s Guide*. Feb. 2023. URL: https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html.

[85] AnySilicon. *White Paper: Is an ASIC Right for Your Next IoT Product?* URL: https://anysilicon.com/asic-right-next-iot-product/.

[86] Hitachi High-Tech. *Semiconductor manufacturing process*. URL: https://www.hitachi-hightech.com/global/en/knowledge/semiconductor/room/manufacturing/process.html.

[87] TSMC Ltd. *Fab Capacity*. URL: https://www.tsmc.com/english/dedicatedFoundry/manufacturing/fab_capacity.

[88] Anton Shilov. *Pumped Up Procs: TSMC Planning Chips 3x Bigger Than Today*. May 2023. URL: https://www.tomshardware.com/news/tsmc-6-reticle-sized-processors.

[89] Moxfyre. *Wafermap showing fully and partially patterned dies*. URL: https://commons.wikimedia.org/wiki/File:Wafermap_showing_fully_and_partially_patterned_dies.svg.

[90] D.K. de Vries. “Investigation of Gross Die Per Wafer Formulas”. In: *IEEE Transactions on Semiconductor Manufacturing* 18.1 (Feb. 2005), pp. 136–139. DOI: 10.1109/tsm.2004.836656. URL: https://doi.org/10.1109/tsm.2004.836656.

[91] Anton Shilov. *TSMC Will Reportedly Charge $20,000 Per 3nm Wafer*. Nov. 2022. URL: https://www.tomshardware.com/news/tsmc-will-charge-20000-per-3nm-wafer.

[92] Josh Norem. *Analyst: TSMC Hitting 55% Yields on 3nm Node for Apple’s A17 Bionic, M3 SoCs*. July 2023. URL: https://www.extremetech.com/computing/analyst-tsmc-hitting-55-yields-on-3nm-node-for-apples-a17-bionic-m3-socs.

[93] Analog Devices Inc. URL: https://www.analog.com/en/technical-articles/fixedpoint-vs-floatingpoint-dsp.html.