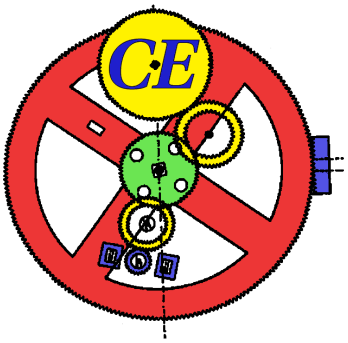


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2020

MSc THESIS

Accelerating Basecalling with Dataflow Computing

Lukas Vermond

Abstract

In an effort to make DNA sequencing more accessible and affordable, Oxford Nanopore Technologies developed the MinION: A portable cellphone sized DNA sequencing device. Translating the information from this device to a nucleotide sequence is called basecalling, and is done with the aid of artificial neural networks. In this thesis, we accelerate a neural network using the concept of Dataflow programming. The result is a complete basecalling application that relies on an FPGA based platform to run the compute intensive parts. We achieve up to 1.51x speedup over a high-end server with two Intel Xeon Processors, with a rate of 57,040 bases per second. In addition, our implementation uses up to 90.27% less energy compared to the original implementation.

Q&CE-CE-MS-2020-10

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Accelerating Basecalling with Dataflow Computing

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Lukas Vermond
born in Utrecht, the Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Accelerating Basecalling with Dataflow Computing

by Lukas Vermond

Abstract

In an effort to make DNA sequencing more accessible and affordable, Oxford Nanopore Technologies developed the MinION: A portable cellphone sized DNA sequencing device. Translating the information from this device to a nucleotide sequence is called basecalling, and is done with the aid of artificial neural networks. In this thesis, we accelerate a neural network using the concept of Dataflow programming. The result is a complete basecalling application that relies on an FPGA based platform to run the compute intensive parts. We achieve up to 1.51x speedup over a high-end server with two Intel Xeon Processors, with a rate of 57,040 bases per second. In addition, our implementation uses up to 90.27% less energy compared to the original implementation.

Laboratory : Computer Engineering
Codenumber : Q&CE-CE-MS-2020-10

Committee Members :

Advisor:	Prof. Dr. Ir. Georgi N. Gaydadjiev, CE
Chairperson:	Prof. Dr. Ir. Georgi N. Gaydadjiev, CE
Member:	Prof. Dr. Ir. Wouter A. Serdijn, Bioelectronics
Member:	Dr. Ir. Marjan Popov, ESE-IEPG

Contents

List of Figures

List of Tables

Acknowledgements

I Would like to thank my supervisor, Georgi Gaydadjiev, for the opportunity to work on this research project, and for his invaluable feedback and suggestions.

In addition, I would like to thank my colleagues at Maxeler Technologies, especially Nils Voß and Joost Hoozemans for their technical help and support.

Last but not least I would like to thank my family and friends for supporting me and taking my mind off of things.

Lukas Vermond
Delft, The Netherlands
September 1, 2020

1 Introduction

Oxford Nanopore Technologies (ONP) developed the first and only nanopore based DNA/RNA sequencing devices, one of which is called the MinION: A portable cellphone sized device that can be connected to a regular computer or laptop. The portability and ease of use of the device aims to enable anyone from sequencing DNA anywhere [5]. Uses of it include: Large scale human genomics, cancer research, research in microbiology and environmental research.

In the context of ONP sequencing, basecalling is the process of translating the raw electrical signal from e.g., the MinION to a nucleotide sequence. Several different basecalling applications exist, all of which are implemented using artificial neural networks [3]. One such application is *Scrappie* [6]. Scrappie is an open-source basecaller by Oxford Nanopore. In this thesis work we port Scrappie to Maxeler Dataflow Engines (DFEs).

1.1 Motivation

Basecalling is typically performed on a CPU, either on a laptop or workstation (slow), or in a data center (expensive, requires internet connection). Another option is acceleration using GPUs. However, this comes at the cost of increased energy usage. An alternative is to use FPGAs to accelerate basecalling, achieving performance on par with GPUs, while still using significantly less energy.

Dataflow computing, developed by Maxeler Technologies, provides means to program FPGA based platforms in an easy and scalable way, by using the abstraction of Dataflow Engines (DFEs).

We can translate this into the following three research questions:

1. Which existing basecalling algorithm is best suited for FPGA acceleration?
2. Can FPGAs provide an interesting alternative for accelerating basecalling/neural-networks in terms of throughput and power consumption as compared to CPUs and GPUs?
3. Is there a way to achieve maximum DRAM read and write bandwidths in the context of an FPGA basecalling application?

1.2 Thesis Organization

In chapter 2 the necessary background on DNA sequencing and artificial neural networks is presented to understand the operation of Scrappie. Subsequently, existing ONP basecallers are discussed, and the concept of dataflow programming using FPGAs is explained. Lastly, a structured design and implementation methodology used to implement such systems is explained.

Chapter 3 introduces the artificial neural network used to implement basecalling in Scrappie. The application is profiled to identify all computational and communication bottlenecks. A software/hardware partitioning of the network is subsequently studied based on the above analysis.

The DFE performance model and hardware design are presented in chapters 4 and 5 respectively. In the performance model, area utilization, I/O usage, and throughput are carefully characterized.

The performance of the implemented final system and its predicted performance are evaluated in chapter 6. In addition, the final system performance is compared to state of the art server CPUs.

2 Background

2.1 DNA Sequencing

To understand what DNA sequencing is, we must first understand the basic structure of DNA. Deoxyribonucleic acid (DNA) consists of two chains of nucleotides, wrapped around one another in a double-helix shape. Each nucleotide contains one of four possible nitrogen bases: Adenine (A), Thymine (T), Cytosine (C), or Guanine (G). The order of these different bases in a chain essentially encode the "instructions" on how an organism will develop [7].

Each base of the one chain is connected to another base in the other chain according to the following rules: A connects to T, C connects to G. This means that both chains contain the same information. Figure 2.1 shows the double-helix structure of a DNA double-helix, and the complementary connected bases.

Figure 2.1: DNA Double-helix structure

In general, DNA sequencing is the process of turning the chains of the DNA double-helix structure into a (digital) sequence of bases, i.e.: a sequence of A, T, C, and G. These sequences can then be used in biological research for a number of tasks, such as person identification or medical diagnosis.

In the following sub-sections, we will briefly go over the history of different DNA sequencing techniques, based on the overview made by Slatko et al [8].

2.1.1 First Generation: Sanger Sequencing

Sanger sequencing was developed in 1977, and is named after its inventor: Frederick Sanger. Although slow compared to later generation sequencing techniques, improvements and commercialization of the technique make it relevant even today.

The Sanger sequencing method works by first splitting the double-helix structure of a sequence in two. One of these strands, the *template sequence*, is replicated by an enzyme called DNA polymerase which adds the complementary sequence to the template. It does so by adding bases one by one. Each time a base is added, there is a chance that this will be a specially engineered base, which will stop the replicating process. These special bases are also luminescently marked. Each base type (A, T, C or G) has it's own unique color.

This will result in many copies of the template, each of different length, and each ended by a luminescent terminating base. The strands are sorted by length, and irradiated by a laser. The luminescent terminating bases are detected by a sensor, which identifies the specific bases. When the above is performed with enough clones of the template sequence, all bases can be determined.

2.1.2 Second & Third Generation Sequencing

The need for faster and cheaper sequencing techniques gave rise to the second and third generation of sequencing techniques. Most of these techniques are variations and/or improvements on Sanger sequencing, mostly focused on sacrificing accuracy for speed. Variations include shortening templates, allowing for more parallel sequencing runs. Lower accuracy can be compensated for by basecalling the same sequences multiple times, and taking some sort of concensus between all results.

2.1.3 Fourth Generation: Nanopore Based Sequencing

Nanopore based sequencing was proposed in the nineties, but only recently popularized by Oxford Nanopore (ONP) [5]. ONP Sequencing devices can be as small as a mobile phone, and are lower cost than aforementioned techniques.

An ONP sequencing device consists of one or more flow-cells. Each flow cell contains up to 2,048 *nanopores*. A solution containing pre-processed DNA samples is inserted into the flow-cell. DNA Strands are allowed to diffuse towards the nanopores. Once at the nanopores, one side of the DNA strand is pulled through the nanopore by an electric field. As the strand moves through the pore, the current across it changes. The resulting signal can be correlated to a sequence of bases.

There exist a multitude of tools to convert a current-over-time signal to a sequence of bases. All of these are implemented in the form of artificial neural networks. The concept of artificial neural network, and several of the available tools will be introduced in section 2.2 and 2.3 respectively.

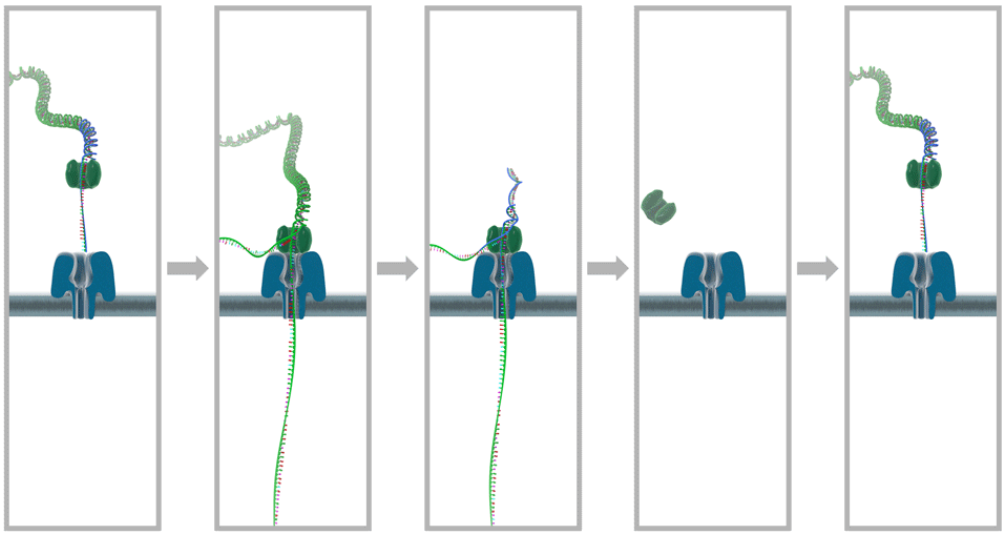


Figure 2.2: Oxford Nanopore sequencing process [1]

Figure 2.2 visualizes the process: So called *motor proteins* are attached to the DNA samples before they are inserted into a flow-cell. These are the dark green shapes in figure 2.2. These motor proteins flow down to the pores, where they are attached to a nanopore. The DNA sequence then moves through the pore, changing the current. When the entire sequence has passed through the nanopore, the process can start again from the beginning.

2.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational structures inspired by biological neural networks. They are often used in applications where classical algorithms are hard to design, or are intractable. Examples include image recognition, natural language processing, and search engine optimizations.

In the first section below we will look at the simplest neural network type, and explain the principles of inference and training. In subsequent sections additional layer types and the related background will be gradually introduced.

2.2.1 Feed-forward Neural Network

One of the simplest example of an artificial neural network is the feed-forward neural network. As the name suggests, in such a network signals flow in one direction through the network layers, i.e.: there are no internal cycles.

An example of a feed-forward neural network with five layers is shown in figure 2.3. Here we see five fully connected layers. These can be clasified as three types: The input layer,

three hidden layers, and the output layer. Each layer contains several nodes, called neurons. The input neurons only hold an input value, programmatically set to some scalar value. For example, in an image-processing network these values might represent pixel data. Each input neuron broadcasts its value to every neuron in the next layer (hidden layer 1). In other words: Neurons in the input layer and neurons in the first hidden layer are connected all-to-all. Neurons in the hidden layer perform a function on their inputs, and present the output, again, to all neurons in the next layer. What kind of function they perform will be explained in more detail in the next sub-section.

The last hidden layer gives outputs to the neurons in the outputs layer. Neurons in the output layer do not perform any function, but simply contain the outputs of the previous layer. These can be read out to obtain the result from the network. In the previously mentioned example of image-processing, y_1 could indicate the likelihood of the image containing a cat, whereas y_2 could indicate the likelihood of the image containing a dog.

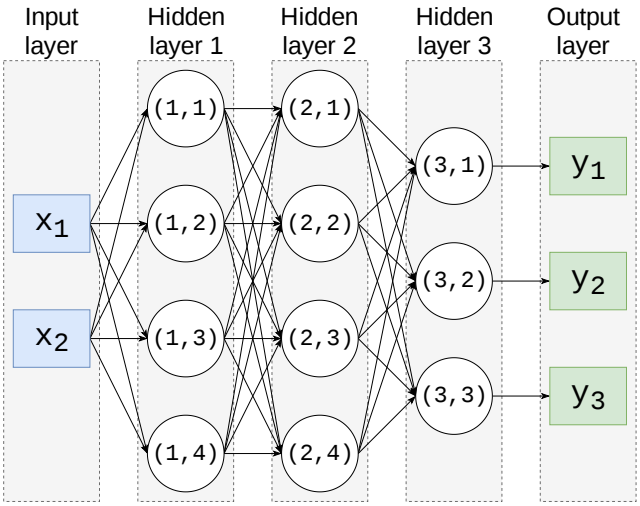


Figure 2.3: Example feed-forward network

Hidden Layers

Every neuron in each of the hidden layers takes the weighted sum plus some bias of its inputs, to produce $z_i^{(L)}$ (equation 2.1), where L is the layer number, i indicates the $i'th$ neuron in that layer, and N_L is the amount of neurons in layer L. $z_i^{(L)}$ then passes through an activation function to produce the neuron's output $a_i^{(L)}$ (equation 2.2), which pushes its argument to either zero or one, analogous to the neuron firing or not firing. A popular activation function is the sigmoid function $\sigma(x) = 1/(1 + e^{-x})$, illustrated in figure 2.4.

$$z_i^{(L)} = \sum_k^{N_{L-1}} w_k^{(L)} a_k^{(L-1)} + b_k^{(L)}, \quad i \in [1, N_L] \tag{2.1}$$

$$a_i^{(L)} = \sigma(z_i^{(L)}) \tag{2.2}$$

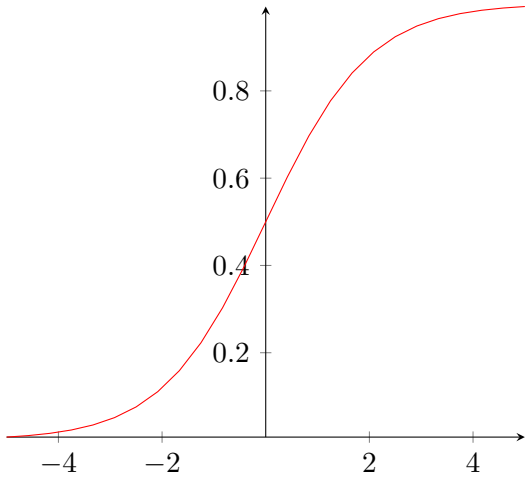


Figure 2.4: Plot of sigmoid function: $\sigma(x) = 1/(1 + e^{-x})$

These equations may be rewritten in a vectorized form, allowing us to think on the layer level, rather than individual neuron level:

$$\vec{z}^{(L)} = W^{(L)} \vec{a}^{(L-1)} + \vec{b}^{(L)} \tag{2.3}$$

$$\vec{a}^{(L)} = \vec{\sigma}(\vec{z}^{(L)}) \tag{2.4}$$

The weight matrices $W^{(L)}$ and bias vectors $\vec{b}^{(L)}$ are initially unknown, and are obtained by training the network.

2.2.2 Training

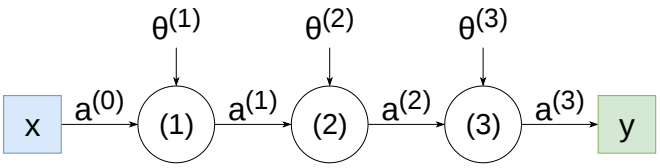


Figure 2.5: Feedforward network vectorized

The network introduced in the previous section is visualized in its vectorized form in figure 2.5. The parameters (weights and biases) for each hidden layer L are represented

with $\theta^{(L)}$. Training the network is done by initially setting all parameters (represented by θ) to initial values. The network is then presented with training inputs $\vec{x}$, with which it will produce outputs $\vec{y}$. These outputs are compared to reference outputs $\hat{\vec{y}}$, and the parameters are updated such that the actual outputs become more similar to the reference outputs. To make the outputs from the network converge to the reference outputs, a cost function is defined, e.g., the mean squared error as shown in equation 2.5. Training the network then becomes equivalent to solving the unconstrained optimization problem of minimizing the cost function for all inputs in the training set.

$$C(\theta) = \frac{1}{N_x} \sum_{\vec{x}} \|\hat{\vec{y}} - \vec{y}\|^2 \tag{2.5}$$

Gradient Descent

Gradient descent is a popular optimization method for minimizing cost functions in neural networks. It tries to find the smallest possible value of $C(\theta)$ for some θ, thereby finding the best parameters for the network. It does this by starting at some point θ_0, and taking a step in the direction such that the cost function decreases fastest. This direction is given by the negative gradient (derivative) of the cost function. Our new point can then be calculated using equation 2.6, where γ is some constant step-size.

$$\theta_{k+1} = \theta_k - \gamma \nabla C(\theta_k) \tag{2.6}$$

This process is repeated until a minimum of the cost function is reached. The final vector θ_k contains the learned weights and biases of the network. This process is illustrated in figure 2.6.

2.2.3 Convolutional Layers

Convolutional layers are slightly different from the feed forward layer we have seen in the last section, and are inspired by the visual cortex in human brains.

1D Convolution

In convolution layers, a window $\vec{k}_i$ slides over an input sequence x, as shown in figure 2.7. As the window slides of the input sequence, the values of the window at each position i are combined into a single value y_i. This is shown in equation 2.7, where $\vec{c}$ is a weight vector, and b is a bias scalar. Both are obtained by training.

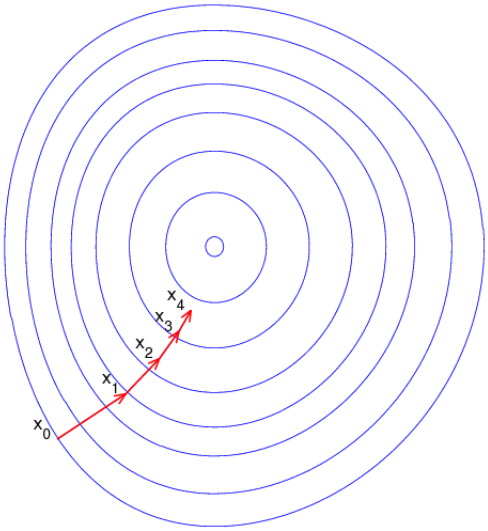


Figure 2.6: Gradient Descent [2]

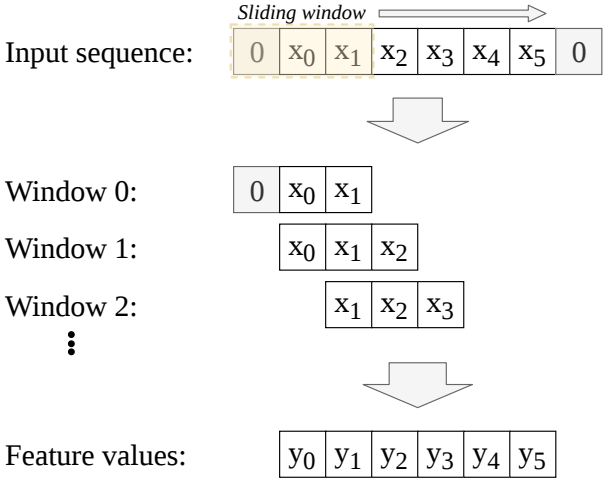


Figure 2.7: 1D Convolution

$$\vec{y_i} = \vec{c}^T \vec{k_i} + b \tag{2.7}$$

$$\vec{k_i} = \begin{bmatrix} x_{i-(winlen-1)/2} \\ \vdots \\ x_i \\ \vdots \\ x_{i+(winlen-1)/2} \end{bmatrix}$$

It is possible to extract multiple features per window. The resulting operation can be

written as a matrix-vector multiplication, followed by addition as shown in equation 2.8. In this case, matrix C consists of parameters that should be learned by the network.

$$\begin{bmatrix} y_{i,0} \\ y_{i,1} \\ \vdots \\ y_{i,m-1} \end{bmatrix} = \begin{bmatrix} \vec{c}_0^T \\ \vec{c}_1^T \\ \vdots \\ \vec{c}_{m-1}^T \end{bmatrix} \vec{k}_i + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{bmatrix} \quad \Leftrightarrow \quad \vec{y}_i = C\vec{k}_i + \vec{b} \tag{2.8}$$

2.2.4 Gated Recurrent Units

Asside from feed-forward and convolutional layers seen in the previous sections, other types of layers are used, such as Gated Recurrent Unit (GRU) layers [9]. These are a type of recurrent neural network which implement a concept of memory.

A GRU layer has inputs $\vec{x}(t)$, and an output $\vec{h}(t)$. Note that inputs and outputs are now a function of time step t, something we did not consider for feedforward layers. A concept of time is important if we want to implement memory in these layers. To be able to function as a type of memory, the output from last time-step $h(t-1)$ feeds back into the layer.

Figure 2.8 shows the design of a GRU layer. The blue lines indicate new information, which is combined with memory, effectively contained in the output of the previous time-step $h(t-1)$. Aside from the input, output, and feedback signals, there are a number of other important signals, namely the *update gate* and *reset gate*, which control how much of the old memory to keep, and how much of new information to use to construct a new memory $h(t)$.

Update Gate

The update gate (equation 2.9) is a function of the input, and the previous "memory" of the output that is mapped to a value between 0 and 1 by the sigmoid function.

$$\vec{z}(t) = \vec{\sigma}(W_z\vec{x}(t) + U_z\vec{h}(t-1)) \tag{2.9}$$

It is used as a factor that controls how much of past information $h(t-1)$ should be remembered, i.e.: How much should be passed on to the current output $h(t)$.

Reset Gate

The reset gate is very similar, and controls how much of the past information to forget. It is defined as:

$$\vec{r}(t) = \vec{\sigma}(W_r\vec{x}(t) + U_r\vec{h}(t-1)) \tag{2.10}$$

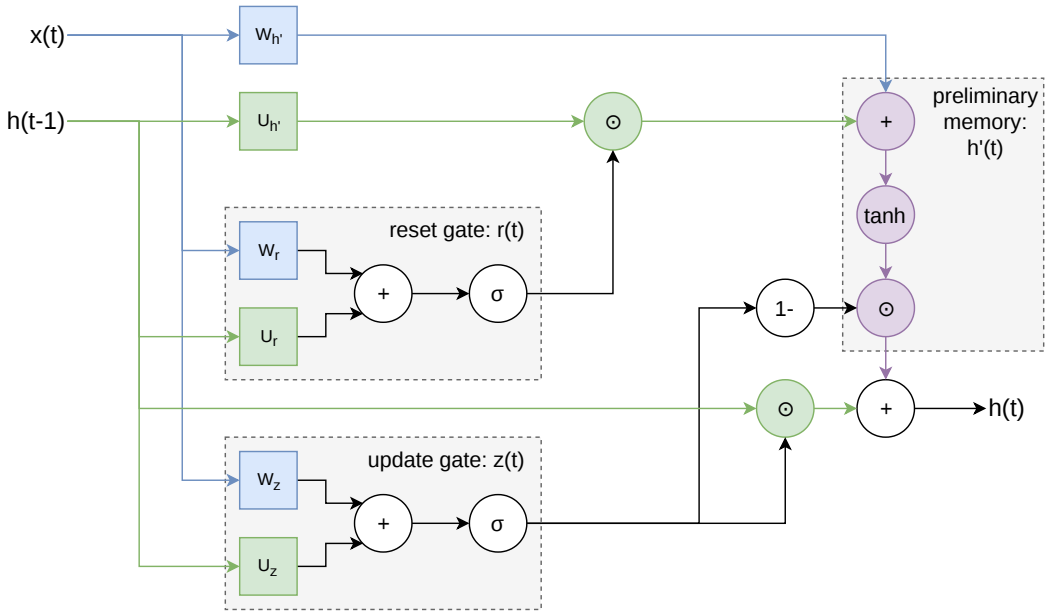


Figure 2.8: GRU Cell

The main difference is in the coefficients (W_r and U_r), and in the way it is used.

Final Memory

A preliminary memory is constructed from the input and the previous memory:

$$\vec{h}'(t) = \tanh\left(W_{h'}\vec{x}(t) + [\vec{r}(t) \odot U_{h'}\vec{h}(t-1)]\right) \tag{2.11}$$

where $\odot$ is the element-wise product of two vectors. Here we can see that $r(t)$ controls how much to keep of $h(t-1)$. We also use a different activation function, namely the hyperbolic tangent. This serves a similar purpose to the sigmoid function, only now the argument is scaled between -1 and 1 rather than 0 and 1.

The final memory is formed as follows:

$$\vec{h}(t) = [\vec{z}(t) \odot \vec{h}(t-1)] + [(1 - \vec{z}(t)) \odot \vec{h}'(t)] \tag{2.12}$$

Since $z(t)$ is between 0 and 1, we get a weighed average between the previous memory $h(t-1)$ and the preliminary memory $h'(t)$, controlled by the update gate $z(t)$. In other words: The update gate controls how much to keep of the previous memory, and how much of the preliminary memory, which in turn is a combination of new information and old information.

In this type of layer all W and U matrices are determined by training the network. In addition, the initially unknown memory $\vec{h}(-1)$ is set to zero.

2.2.5 Conditional Random Fields

The outputs $a_{y_t}(x_t)$ of a neural network can be turned into a probability distribution using the *Softmax function* (equation 2.13). The softmax function computes the probability that output y_t has label L given input x_t.

$$p(y_t|\vec{x_t}) = \frac{exp(\, a_{y_t}(\vec{x}_t)\,)}{\sum_{i=1}^{N_{lab}} exp(\, a_i(\vec{x}_t)\,)} \tag{2.13}$$

The probability of a sequence of N_{seq} outputs $\vec{y}$ is then simply the product of the probabilities for each individual output y_t as shown in equation 2.14. This product of exponentials may be written as an exponent of a sum, divided by a normalization constant.

$$\begin{aligned} p(\vec{y}\,|X) &= \prod_{t=1}^{N_{seq}} p(y_t|\vec{x}_t) = \prod_{t=1}^{N_{seq}} [exp(a_{y_t}(\vec{x}_t)) \,/\, z(\vec{x}_t)] \\ &= exp\left(\sum_{t=1}^{N_{seq}} a_{y_t}(\vec{x}_t)\right) \;/\; \prod_{t=1}^{N_{seq}} z(\vec{x}_t) \end{aligned} \tag{2.14}$$

To model the probability of some label in position t being succeeded by another label in position $t+1$, we introduce a new term $V_{y_t,y_{t+1}}$ in equation 2.15. Matrix V contains in the y_t'th row, y_{t+1}'th column, the likelihood that label y_t is followed by label y_{t+1}.

$$p(\vec{y}\,|X) = exp\left(\sum_{t=1}^{N_{seq}} a_{y_t}(\vec{x}_t) + \sum_{t=1}^{N_{seq}-1} V_{y_t,y_{t+1}}\right) \;/\; Z(X) \tag{2.15}$$

The value in this is best explained using an example: Suppose our network takes images of written characters as input, and outputs probabilities of the image being any character. Without matrix V the network may conclude a sequence of, for example, two consecutive I's may be highly likely given two images that resemble an I. But in reality we know that words in the English language rarely contain two consecutive I's. Through training, the network would learn to put a low probability in matrix V at the entry that represent the letter I followed by another I.

The Normalizing Constant

The normalizing constant is computed by $Z(X)$, and is defined in equation 2.16. Computing this function in the naive way is intractable, since it involves computing N_{seq} nested sums of size N_{lab}.

$$Z(X) = \sum_{y'_1} \sum_{y'_2} ... \sum_{y'_{N_{seq}}} exp\left(\sum_{t=1}^{N_{seq}} a_{y'_t}(\vec{x}_t) + \sum_{t=1}^{N_{seq}-1} V_{y'_t, y'_{t+1}} \right) \tag{2.16}$$

A dynamic programming solution exists to compute the normalizing constant, given in listing 2.1.

Listing 2.1: CRF Partition Function

```
1  function Z(X)
2      curr ← [ Σ_{i=1}^{N_lab} exp(a_i(x⃗_1) + V_ij) | j ∈ [1, N_lab] ]
3
4      for each t ∈ [2, N_seq] do
5          prev ← curr
6          curr ← [ Σ_{i=1}^{N_lab} exp(a_t(x⃗_1) + V_ij) × prev_i | j ∈ [1, N_lab] ]
7      end
8
9      return Σ_{i=1}^{N_lab} exp(a_i(x⃗_1)) × curr_i
10 end
```

Viterbi Algorithm

The most likely sequence of outputs $\vec{y}^*$ can be calculated, as shown in equation 2.17, by the Viterbi algorithm (listing 2.2).

$$\vec{y}^* = \arg\max_{\vec{y}} p(\vec{y}|X) = Viterbi(\hat{X}) \tag{2.17}$$

$$\hat{X} = \frac{1}{Z(X)} X \tag{2.18}$$

For each observation t the probability of y having any label is computed, based on the most likely previous label k. These probabilities are stored in matrix $Prob$. The corresponding predecessor index k is stored in the trace-back matrix $Trace$. Note that matrix V encodes transition likelihood, like explained before. Matrix $\hat{X}$ contains the likelihood of getting label i given input x_t at row i, column t. When $Prob$ and $Trace$ are completely filled, the most likely path of labels is selected by walking the trace-back matrix backwards. This is visualized in figure 2.9, where the coloured lines indicate the paths to the final states ($t = N_{seq}$).

Listing 2.2: Viterbi Algorithm

```
1  function Viterbi(X̂)
2      for each i ∈ [1, N_lab] do
3          Prob_{i,1} ← InitProb_i
4      end
```

```
5
6       for each t ∈ [2, N_seq] do
7           for i ∈ [1, N_lab] do
8               Prob_{i,t} ← max_k{Prob_{k,t−1} × V_{k,i} × X̂_{i,t}}
9               Trace_{i,t} ← arg max_k{Prob_{k,t−1} × V_{k,i} × X̂_{i,t}}
10          end
11      end
12
13      y_{N_seq} ← arg max_k{Prob_{k,N_seq}}
14
15      for each t ∈ [N_seq, 2] do
16          y_{t−1} ← Trace_{y_t,t}
17      end
18
19      return ȳ
20  end
```

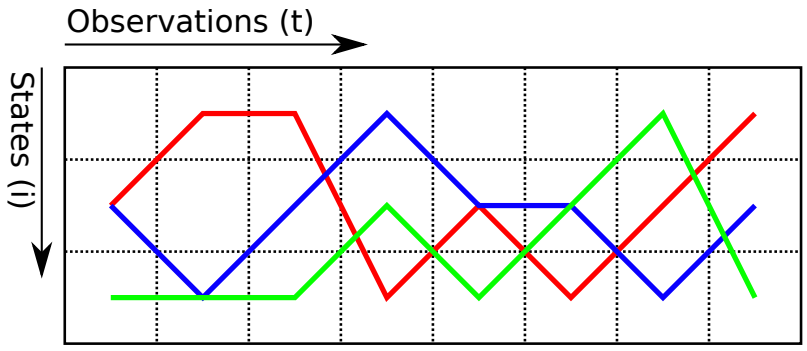


Figure 2.9: Viterbi Traceback Visualised

2.2.6 Residual Blocks

Adding more layers to a neural network typically increases its accuracy. Beyond a certain point, however, accuracy drops. This is due to the fact that derivatives of the network become smaller and smaller as the number of layers increases. This is known as the vanishing gradient problem.

To understand the vanishing gradient problem, let us inspect the cost function of a neural network with multiple hidden layer:

$$\frac{dC(\theta)}{d\theta} = \frac{1}{N_x}\sum_{\vec{x}}\frac{d}{d\theta}C_{\vec{x}} \tag{2.19}$$

$$\frac{dC_{\vec{x}}}{d\theta} = \frac{\partial C_{\vec{x}}}{\partial a^{(N_L-1)}}\left(\frac{\partial a^{(N_L-1)})}{\partial a^{(N_L-2)}}\frac{\partial a^{(N_L-2)}}{\partial a^{(N_L-3)}}\cdots\frac{\partial a^{(1)}}{\partial a^{(0)}}\frac{\partial a^{(0)}}{\partial \theta}\right) \tag{2.20}$$

$$\frac{\partial a^{(L)}}{\partial a^{(L-1)}} = \sigma'(z^{(L)})\frac{\partial z^{(L)}}{\partial a^{(L-1)}} \tag{2.21}$$

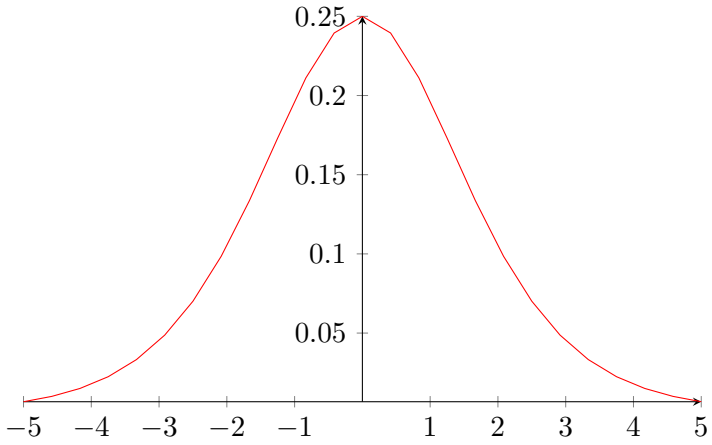
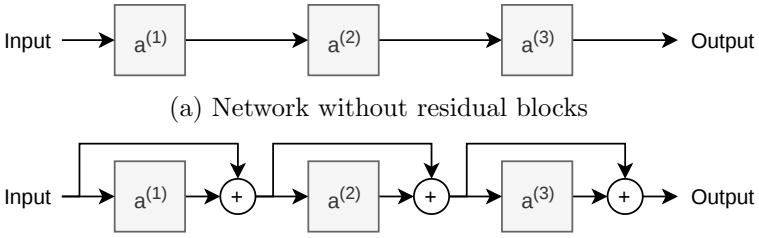


Figure 2.10: Plot of the derivative of the sigmoid function



(a) Network without residual blocks

(b) Network with residual blocks

Figure 2.11: Example networks with and without residual blocks

By repeatedly applying the chain rule of differentiation, we get the partial derivative of each layer, shown in equation 2.21, where σ' is the derivative of the sigmoid function, which is plotted in figure 2.10. Since each layer's partial derivative contains a σ', the gradient of the cost function decays exponentially to zero. As the gradient goes closer to zero, the steps taken in gradient descent becomes smaller and smaller, slowing down training to the point of standstill.

One way to solve this problem is to have residual blocks in the network. In figure 2.11a we see a neural network without residual blocks. In figure 2.11b we see the same network but with residual blocks. By adding the input of a layer to its output, we effectively allow signal to skip layers. The cost function of the residual net can then be written as:

$$\frac{dC_{\vec{x}}}{d\theta} = \frac{\partial C_{\vec{x}}}{\partial f^{(N_L-1)}} \left(1 + \frac{\partial a^{(N_L-1)})}{\partial f^{(N_L-2)}}\right) ... \left(1 + \frac{\partial a^{(1)}}{\partial f^{(0)}}\right) \left(1 + \frac{\partial a^{(0)}}{\partial \theta}\right) \tag{2.22}$$

$$f^{(L)} = f^{(L-1)} + a^{(L)} \tag{2.23}$$

Each factor is now at least 1, so the gradient will not go to zero, unless $\partial C_{\vec{x}}/\partial f^{(N_L-1)}$ goes to zero.

2.3 Oxford Nanopore Technologies Compatible Basecallers

There are several Oxford Nanopore Technologies (ONT) basecallers, some of which are closed-source e.g., *Albacore*, *Guppy*, *BasecRAWller*, and *Metrichor*. These are not suitable candidates for acceleration, since we need access to the application source-code to accelerate it. In addition, there are four open source ONT basecallers:

1. Nanonet;
2. DeepNano;
3. Chiron;
4. Scrappie.

Our comparison is based on a review of these basecallers published by R. Wick et al [3].

Nanonet is ONTs first neural network based basecaller, and is no longer under active development. DeepNano [10] was developed at Comenius University, and also does not seem to be maintained anymore. Chiron [11] was developed at the University of Queensland, and is based on Google's TensorFlow: an open-source software library for machine learning. Scrappie is ONTs research basecaller. It is completely written in C and only relies on the BLAS library for its neural-network implementation.

2.3.1 Basecalling Accuracy

All open-source basecallers mentioned earlier are compared in terms of accuracy. Scrappie implements several networks, named *raw*, *raw R94*, *RGR R94*, *RGRGR R94*, and *RNNRF R94*.

Figure 2.12 shows *read identities* for different basecallers. This is a measure of how well reads align to a given reference sequence. All measurements were made with the same input data. DeepNano, Nanonet and Scrappie's raw, and raw R94 networks perform relatively poorly.

An arguably more important measure is how well a consensus between all reads of a particular basecaller align to the reference. This is done by aligning all reads to the reference. Then, where reads overlap, the best ones are selected based on their read identity and length. The result is called an assembly, and the associated performance metric the assembly identity. Figure 2.13 shows assembly identities. Chiron v0.3 is ahead of most. However, note that the scale of the vertical axis goes from 98.5% to 100%.

2.3.2 Source-code Analysis

Since Nanonet and DeepNano are no longer being developed, and have relatively poor accuracy, they will no longer be considered as candidates for acceleration. That leaves us with several Scrappie networks, and Chiron.

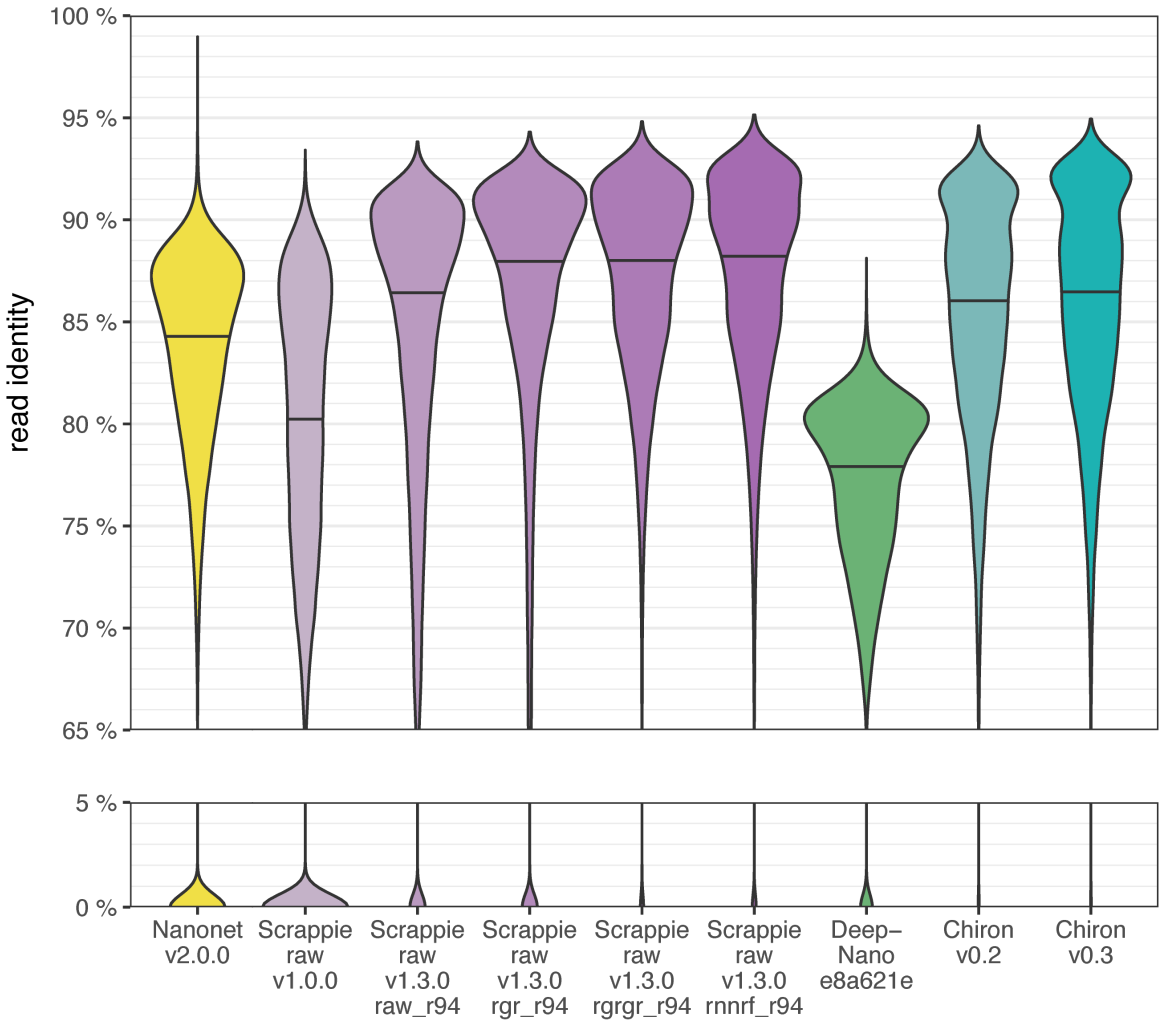


Figure 2.12: Read identities [3]

Chiron is written in Python, and relies heavily on TensorFlow. Because TensorFlow is so heavily used, accelerating Chiron would mainly involve the acceleration of TensorFlow functions. While TensorFlow is open-source, it is a very big and rather complicated library. Analysing its many internal functions and abstraction layers, to then come up with a hardware design performing the same function will most likely cost a lot of time.

Scrappie is written in C, and only relies on BLAS functions and some SSE intrinsics to perform simple vectorized-arithmetic, matrix-vector multiplication, and matrix-matrix multiplication. This is much easier to analyse, and subsequently design hardware for.

In Scrappie 1.4.0 (Jul. 12, 2019), support for the RGR R94 model was dropped. Figure 2.13 shows that the raw R94 model performs slightly worse than the other networks. Mostly due to yielding reads that fail to align (0% read identity). This leaves us with the RGRGR R94 and RNNRF R94 models. The main difference between the RGRGR R94 and the RNNRF R94 models is in the way they decode results at the end. The

RGRGR R94 models decode results with an algorithm that makes a lot of forewards and backwards passes over the data. This doesn't map well to a streaming architecture. Each pass is similar but unique operations are performed on the data. Sharing resources between passes is predicted to be difficult. Therefore mapping this algorithm to hardware is expected to be inefficient.

2.3.3 Conclusion

Scrappie's RNNRF R94 model has good accuracy and is well documented, and has readable source-code. This makes it the best candidate for acceleration in hardware.

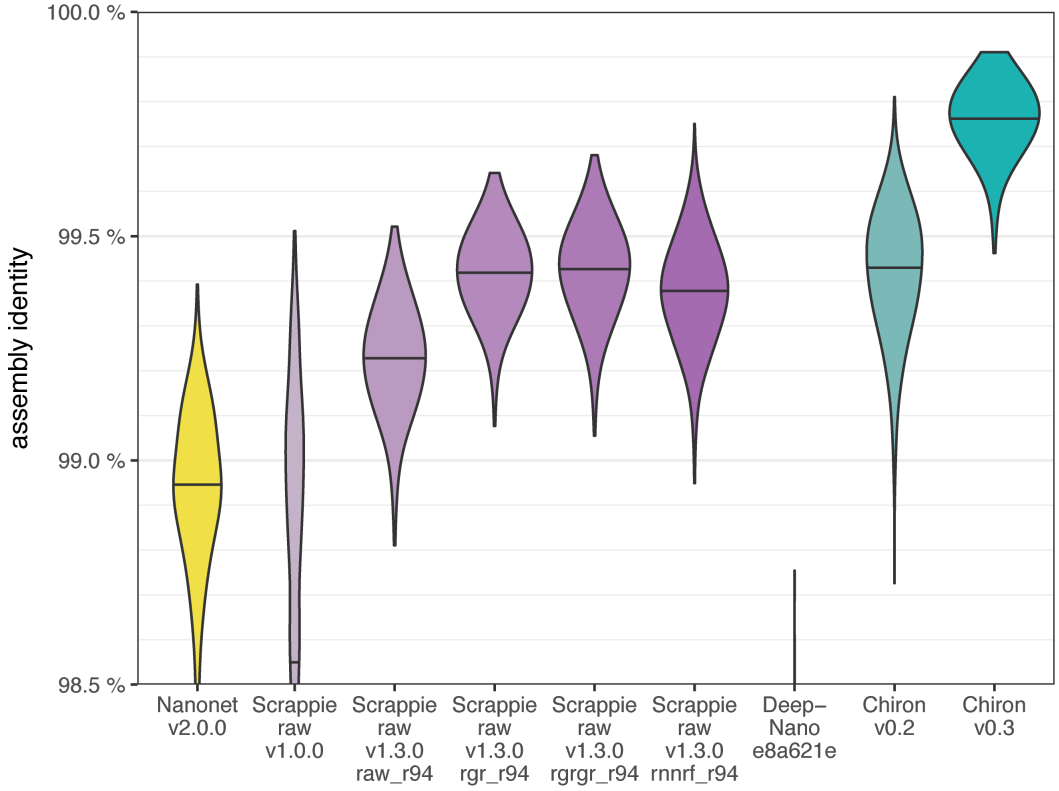


Figure 2.13: Assembly identities [3]

2.4 Related Work

The related work presented below consists of two parts: Neural networks accelerated using FPGAs, and basecallers accelerated using GPUs. One additional basecalling also deserves mention: EPI2ME [12]. EPI2ME is a cloud-based DNA analysis platform by Metrichor, a company owned by Oxford Nanopore Technologies. They offer basecalling, as well as secondary analysis services in the cloud, rather than a local machine. Nothing else is known about the performance of EPI2ME.

2.4.1 FPGA Accelerated Neural Networks

As will become clear in chapter 3, a large portion (93.9%) of time in the RNNRF R94 model is spent in GRU layers. Li et al [13] show how a GRU based recurrent neural network can be accelerated on an FPGA using Vivado HLS [14]. In general, the network works by loading weights into an on-chip buffer before processing input data. Matrix-vector multiplications are tiled and mapped to smaller matrix-vector multipliers on the chip. n by n multipliers are implemented with n parallel multipliers followed by an adder tree. They claim a speedup of about 3.12 on a *Xilinx UltraScale+ XCVU9P* over an *Intel Xeon i5 CPU 650*.

Another FPGA accelerated network developed by Zhang et al [15] is trained for video content recognition. This network is similar to the RNNRF R94 model in the sense that it also has a convolutional layer followed by recurrent layers. However, this design operates on two dimensional input data rather than one dimensional data. Their major challenges are in caching data on-chip to not go to off-chip memory to often. Development has been done using HLS. Their implementation on a *Xilinx Virtex-7 VC709* achieves a speedup of 4.75X over an *Intel Xeon E5-2630* CPU, and a 3.1X over an *NVIDIA K80* GPU.

2.4.2 GPU Accelerated Neural Networks

One of the basecallers mentioned earlier, Chiron [11], is implemented using TensorFlow. TensorFlow is a framework for machine learning by Google [16]. Neural Networks implemented using TensorFlow can run on CPUs as well on GPUs. Chiron achieves a speed of 2,657 bases per second on a *NVIDIA GTX 1080 Ti*.

Guppy is the most recent, fastest, and closed-source basecaller by Oxford Nanopore. Not much is known about its implementation, except that they use alternating reverse and forward GRU layers [3]. Guppy can run on GPUs to produce 1.5 million bases per second. This is likely due to the fact that the network they implement is much smaller then the RNNRF R9 network, or Chiron's network.

After this thesis work was started, an successor of Scrappie has been made open-source by ONP, named Flappie. According to Wick et al [3], Flappie can produce 14,000 bases per second on the CPU. Flappie seems to use similar neural-networks included in Scrappie. The biggest differences seem to be the use of *Long short-term memory* (LSTM) layers instead of GRU layers, and updated decoding methods [17].

2.5 Maxeler Multiscale Dataflow Systems

Using MaxCompiler, developed by Maxeler Technologies, algorithms can be programmed in a high-level language, called MaxJ, and mapped to hardware structures such as FPGAs. In the following sections, the idea behind the language, its compiler, and target hardware platforms are explained.

2.5.1 Dataflow Engines

Figure 2.14 shows the general layout of a Dataflow Engine (DFE). A DFE consists of an FPGA which is connected to dedicated DRAM, and to the CPU through PCI express. The host CPU contains a software layer called MaxelerOS, which allows the DFE to communicate with software programs, typically written in C/C++.

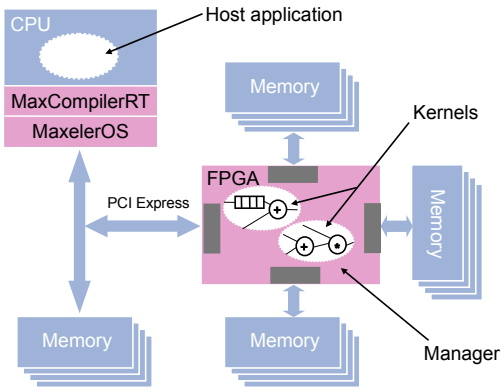


Figure 2.14: Maxeler DFE architecture

2.5.2 Dataflow Programming

Dataflow computing builds upon the idea of systolic arrays [18, 19]. Dataflow Graphs (DFGs) are described in MaxJ: a high-level java based programming language. MaxCompiler schedules DFGs on fixed, interconnected hardware structures on a DFE. These structures are contained in *Kernels*. Kernels are internally synchronous, but communicate with other hardware blocks such as other kernels and memory controllers asynchronously. All the control and stalling logic for this asynchronous communication is generated by MaxCompiler.

Motivational Example: Moving Average Filter

How all of this work together is perhaps best illustrated by example. Consider a moving average filter defined as:

$$y_t = \frac{1}{n} \sum_{i=1}^{n} x_{t-i} \tag{2.24}$$

Suppose that we would like to implement this filter with an input x_t, through which an input value streams into the filter every time-step t. An n-point average is produced every tick at output y_t.

The kernel code for our n-point average filter is shown in listing 2.3 for the case of $n = 4$. We start by defining a class that inherits from `Kernel`. In the constructor we first

define inputs and outputs, which are represented by `DFEVar`'s. These can be thought of as streams from which we can read data (in case of `x`), or to which we can write data (in case of `y`). Subsequently we sum values at different times in `DFEVar sum. stream.offset( x, -i, 0.0)` gets a value at an offset in stream x, namely at point $-i$. i.e.: from i ticks ago, which effectively equals x_{t-i}. The last parameter (0.0) to `stream.offset` is the value returned when $t - i$ becomes negative. Lastly we stream out $\frac{sum}{n}$ to y, indicated by the arrow operator (`<==`).

Figure 2.15 shows the Data Flow Graph (DFG) that is generated from listing 2.3. The stream offsets are represented by diamond shapes containing the amount of ticks to offset the stream by. When this graph is compiled to a hardware implementation, we can imagine these shapes to be replaced by registers to delay the input stream x.

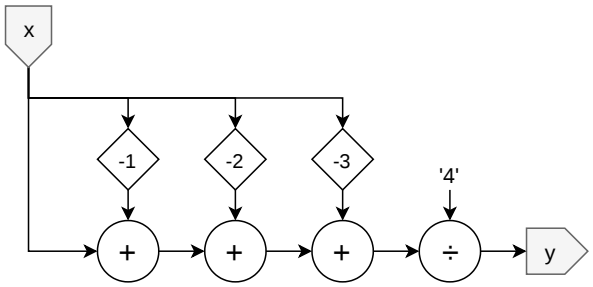


Figure 2.15: Average Filter DFG

Listing 2.3: Average filter kernel code

```
1  public class AvgFilterKernel extends Kernel {
2      public AvgFilterKernel(final KernelParameters parameters) {
3          super(parameters);
4          final int n = 4;
5          DFEVar x = io.input("x", dfeFloat(8, 24));
6          DFEVar y = io.output("y", dfeFloat(8, 24));
7
8          DFEVar sum = x;
9          for(int i = 1; i < n; ++i) {
10             sum += stream.offset(x, -i, 0.0);
11         }
12
13         y <== sum / n;
14     }
15 }
```

Our kernel is instantiated and connected to PCIe input, and PCIe output in another class, which is known as a manager. The manager code is presented in listing 2.4. In the manager constructor the kernel is first instantiated, and subsequently connected to streams from/to the CPU through PCIe.

Listing 2.4: Average filter manager code

```
1  public class AvgFilterManager extends MAX5CManager {
2      public AvgFilterManager(EngineParameters params) {
```

```
3         super(params);
4
5         KernelBlock avgKernel = addKernel(new AvgFilterKernel(
6             makeKernelParameters("AvgFilterKernel")));
7         avgKernel.getInput("x") <== addStreamFromCPU("x");
8         addStreamToCPU("y") <== avgKernel.getOutput("y");
9     }
10 }
```

From the MaxJ code, a DFE configuration is generated, along with some c-code to interface with the average filter from the CPU side through Maxeler's proprietary driver software, called MaxelerOS. Listing 2.5 shows some example c-code that can be used to stream input values into our average filter, and obtain the results with a single call to the generated function `AvgFilter`. `AvgFilter` takes as arguments:

1. The amount of ticks to run for,
2. input data for input x,
3. the size of the input data for input x,
4. a pointer to memory for output data from y, and
5. the expected size of data obtained from y.

Calling this function runs the DFE on the input data, and returns when it's finished producing all output data. There is also an asynchronous interface, but that is out of the scope of this example.

Listing 2.5: Average filter CPU side code

```
1  #include "AvgFilter.h"
2
3  int main(void) {
4      const size_t n = 16;
5      float x[n];
6      float y[n];
7      size_t i;
8
9      for(i = 0; i < n; ++i) {
10         x[i] = i;
11     }
12
13     AvgFilter(n, x, n*sizeof(float), y, n*sizeof(float));
14
15     for(i = 0; i < n; ++i) {
16         printf("y[%lu] = %f\n", i, y[i]);
17     }
18
19     return 0;
20 }
```

Finite State Machines

Finite State Machines (FSMs) can be implemented in MaxJ to control compute paths in kernels. This is done by deriving a class from `KernelStateMachine`, and implementing the methods `updateState` and `setOutput`. An example FSM implemented in MaxJ is shown in listing 2.6. This example FSM has an input `i_reset`, an output `o_count`, and internal state variables `s_state` and `s_counter`.

Listing 2.6: Example FSM code

```
1  class UpDownCounterFSM extends KernelStateMachine {
2      ...
3      @Override void updateState() {
4          IF(i_reset) {
5              s_counter.next <== 0;
6              s_state.next <== State.COUNTING_UP;
7          } ELSE {
8              SWITCH(s_state) {
9                  CASE(State.COUNTING_UP) {
10                     IF(s_counter === m_counterMax) {
11                         s_state.next <== State.COUNTING_DOWN;
12                         s_counter.next <== s_counter - 1;
13                     } ELSE {
14                         s_counter.next <== s_counter + 1;
15                     }
16                 }
17                 CASE(State.COUNTING_DOWN) {
18                     IF(s_counter === m_counterMin) {
19                         s_state.next <== State.COUNTING_UP;
20                         s_counter.next <== s_counter + 1;
21                     } ELSE {
22                         s_counter.next <== s_counter - 1;
23                     }
24                 }
25             }
26         }
27     }
28
29     @Override void setOutput() {
30         o_count <== s_counter;
31     }
32 }
```

2.5.3 Memory Controller

In addition to streaming data from/to PCIe connections, kernels may also stream data to/from on-card DRAM. In MaxJ, DRAM is accessed by instantiating a Memory Controller Pro (MCP) block, which is assigned any amount of available DIMMs. Using the MCP, read and write ports can be instantiated. Both the read and write ports take a stream of commands, which describe what range of memory addresses to read or write.

Figure 2.16 shows an overview of the MCP and its connected parts. Each of the port blocks can be either a read or write port. How the MCP and read-/write ports work, will be explained in the following sub-sections.

The MCP arbitrates the read and write ports, and talks to vendor specific ip-cores to talk to the DIMMs. In case of the MAX5C, these are the Memory Controller Interface (MCI) blocks, which are wrappers around Xilinx ip-cores that talk to the DIMMs.

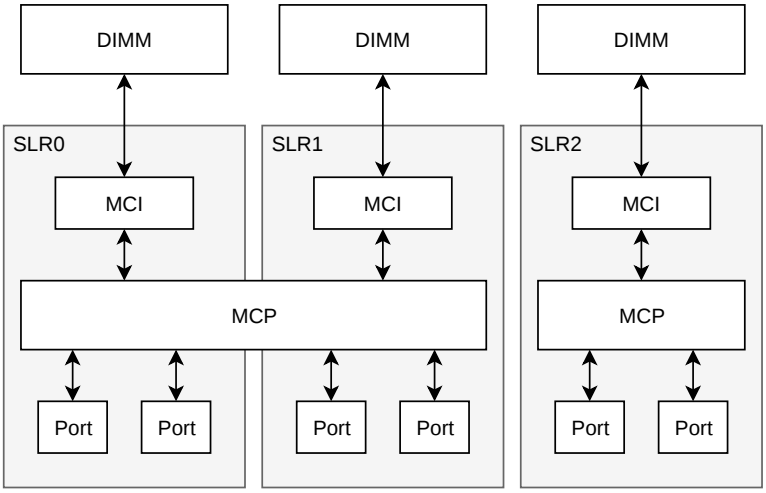


Figure 2.16: Memory Controller Pro (MCP)

Read/Write Ports

Read ports have the following streams connected to them:

- Commands (in);
- Data (out);
- Echoes (out, optional).

Operation is straightforward: The read port receives commands, reads the data, which is then streamed out.

Write ports have the following streams connected to them:

- Commands (in);
- Data (in);
- Echoes (out, optional).

Data is streamed in and written to the address range described by the command stream.

Optionally the read and write port can produce echoes. An echo is produced by the port when a command that has the `SendEcho` bit set is fully processed, i.e.: When it's data is

PCIe (full-duplex)	DRAM
3 GB/s	45 GB/s

Table 2.1: MAX5C Data rates

BRAM	URAM	DRAM
9.3 MB	32 MB	45 GB

Table 2.2: MAX5C Storage capacity

LUTs	FFs	BRAMs	URAMs	DSPs
1,182,240	2,364,480	4,320	960	6,840

Table 2.3: MAX5C Resources [4]

fully read from, or written to DRAM. Echoes can be configured to either be copies of the originating command, or just a single bit.

Command format

Commands streamed to the read/write ports have the following format:

- `SendEcho` (1 bit);
- `Address` (31 bits);
- `Size` (8 bits);
- `Inc` (8 bit);
- `Stream` (15 bits);
- `Tag` (1 bit).

Each command will make a port read/write `Size` amount of bursts, starting at address `Address` with an increment of `Inc` between each burst address. Note that `Address`, `Size`, and `Inc` are all expressed in bursts of 512 bits per DIMM.

If `SendEcho` is set, a copy of the command will be sent over an echo stream as a means of acknowledging that the command has been processed. Optionally, a single high bit can be streamed instead of a copy of the command. If `Tag` has been set, an interrupt will be sent to the CPU, to let the host program know that the command has been processed. The difference between `SendEcho` and `Tag` is that the former is used by kernels, whereas the latter is used by the host CPU program. The `Stream` field can be used to multiplex the command-stream between multiple ports, but this is not used in this work.

2.5.4 MAX5C Dataflow Engine

MaxCompiler can target several different FPGA based platforms. The target platform used in this work the MAX5C. The MAX5C is a card developed by Maxeler that is connected through PCIe to the motherboard. At the heart of the MAX5C is the Virtex Ultrascale+ VU9P FPGA from Xilinx [4]. The VU9P consists of three dies, called Super Logic Regions (SLRs), in a single package. Additionally, the MAX5C card incorporates a large amount of on-card DRAM (called Large Memory or LMEM in the Maxeler tools). Resources, bandwidths, and storage capacities are shown in tables 2.1, 2.2 and 2.3.

Configurable Logic Blocks

The VU9P contains 147,780 Configurable Logic Blocks (CLBs). Each CLB contains 8 Look-up tables (LUTs), and 16 flip-flops. The LUTs can be configured to have 1 output and 6 inputs, or 2 outputs with 5 common inputs. CLBs also contain multiplexers and carry generation logic to implement e.g., adders. CLBs are mostly used for logic, addition, and divider circuits.

Digital Signal Processing Blocks

Digital Signal Processing (DSP) blocks contain 27 by 18 bit multipliers, followed by a 48 bit accumulator. One of the multiplier inputs also contains a 27 bit pre-adder. DSPs are mostly used to implement multipliers.

On-chip Memory

The VU9P has two types of on-chip memory: Block RAMs (BRAMs), and UltraRAMs (URAMs). These are refered to in the Maxeler tools as Fast Memory (FMEM). BRAMs can be configured as $32k \times 1$, $16k \times 2$, $8k \times 4$, $4k \times 9$, $2k \times 18$, $1k \times 36$, or 512×72. BRAMs can be used as a FIFO or a RAM with up to two write ports, and two read ports. In addition, a BRAM block can be configured as two completely independent 18 kb memory blocks, called BRAM18's. MaxCompiler counts memory as BRAM18 blocks in its resource utilization reports.

UltraRAMs are a much bigger on-chip memory, with a capacity of $4K \times 72$ bits. In addition, URAMs have a single read port and a single write port and cannot be used to cross clock domains.

Dynamic Random Access Memory

The MAX5C contains three DIMMs of on-card DRAM, directly connected to the VU9P. Each DIMM has a capacity of 16 GB, for a total of 48 GB.

2.6 Design Methodology

Over the years, a methodology for the design of data-flow applications using DFEs has been developed [20]. The primary goal of this methodology is to do perform design space exploration early at a high abstraction level to prevent spending a lot of time and resources to develop an architecture that will not fit, not meet timing, or will not provide satisfactory speedup. In this section we briefly discuss the most important steps and techniques this method offers.

Figure 2.17 shows the five major parts and how they interact with each other.

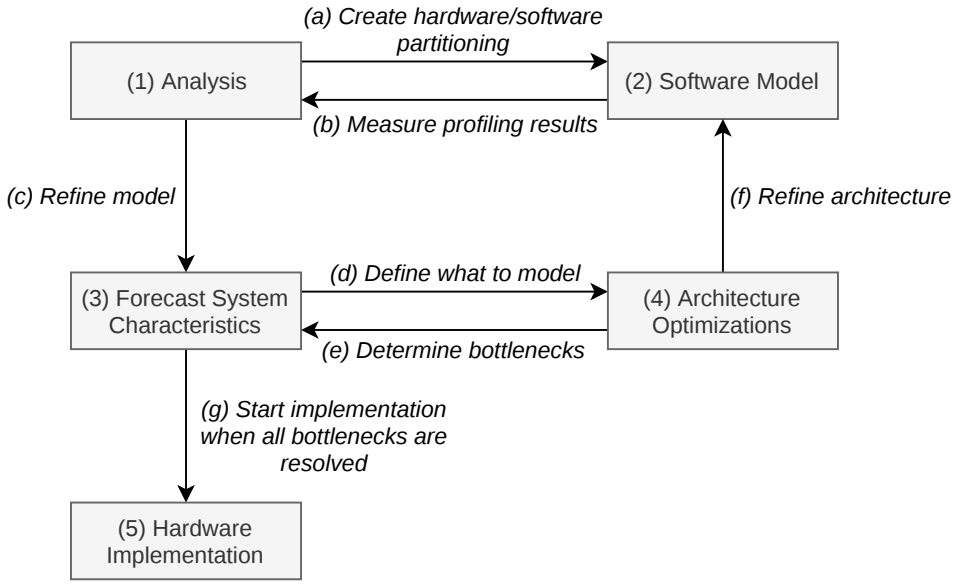


Figure 2.17: Design methodology breakdown and its five parts and seven steps

1. Analysis of target application;
2. Software modelling of the intended hardware design;
3. Predicting performance and resource utilization;
4. Development of hardware architecture;
5. Hardware Implementation.

The starting point in this process is step 1: Analysing the original program/algorithm, in order to develop a software model of your architecture. However, the software model is also needed to analyse the target program/algorithm. Further analysis of the software model is done to refine the architecture, which in turn causes the need to update the software model. The performance of the actual hardware implementation is calculated in a performance model. In the performance model the speed and area usage of the final implementation are approximated, before starting work on the actual implementation. These steps can be iterated upon until a design is obtained that meets its requirements. One thing that is not captured in figure 2.17, is the constant verification between software model, hardware implementation, and performance model.

2.6.1 Application Analysis

The target program is benchmarked to identify compute-intensive tasks that should be accelerated. This allows us to partition the program into a software and hardware part. When partitioning the program it is also important to predict or measure the amount of data that will be streamed between the DFE and CPU, and between the reconfig-

urable chip and the off-chip memory on the DFE. This is to ensure that application-level performance is not bottle-necked by PCIe/LMEM read/write speed.

Information should be collected about the DFE part in terms of data movement, number of compute-operations, and in what order they are to be executed. In addition, data movement between the DFE and off-chip memory should be carefully analyzed.

2.6.2 Software Model

The software model is essentially a re-implementation of the target application, where the DFE part is written not for speed, but to represent the intended hardware implementation. The software model can be used to verify correctness of the design, to test and to debug the DFE implementation in later stages.

2.6.3 Performance Model

With an initial design, a performance model can be constructed. In the performance model, runtime and area usage are estimated. Because FPGAs consist of fully predictable building blocks, it is possible to model performance accurately without the need for implementing and benchmarking your design. This allows for rapid design space exploration. Once a satisfactory design is obtained, the software model should be updated.

2.6.4 Hardware Implementation

Once the performance model indicates satisfactory performance, the hardware implementation can be started. While developing the hardware implementation, correctness should continuously verified with the software model. In addition, resource utilization and performance characteristics can be checked against the performance model.

3 System Overview

An overview of Scrappie's RNNRF R94 network is given in figure 3.1. Data streams through the network, one step at a time. Each edge in the diagram is annotated with the width of the data flowing from one layer to the next. The raw sequence is read from a file generated by an ONP sequencing device, and are streamed into the network. Sequencing samples are streamed into a convolutional layer, which creates 112 features per input element.

The stream of feature-vectors produced by the convolutional layer is passed to a sequence of five GRU layers. The five GRU layers alternate between reading data streams forwards and backwards, with the first one reading backwards. This means that all data from the convolution layer, and each GRU layer must be processed entirely before the next layer can begin, since it needs the last vector from the previous layer first.

After the final GRU layer, a feed-forward layer with 112 input neurons and 25 output neurons creates a stream of size 25 vectors. These output vectors are interpreted as 5 by 5 matrices for use in the CRF decoding step, which follows next.

A modified CRF decoder is used to produce labels, which are represented by a stream of integers. There are 5 possible labels: `A`, `T`, `C`, `G`, and `b`. `A`, `T`, `C` and `G` represent the four nucleotides of DNA. The `b` label is the blank label. It is taken to mean 'No change'. In the Viterbi algorithm explained in section 2.2.5, a series of observations (matrix X), and a transition probability matrix V are used to find the most likely sequence of labels for the input data. In the RNNRF R94 network a slight variation is used. Instead of a constant transition matrix V, there is one transition matrix V_t per time-step. These matrices come from the previous layer. The X matrix is left out completely. Since matrix X is left out, a normalization step for it is not needed, but the V_t matrices do need to be normalized, so a normalization layer is still present.

The modified Viterbi algorithm layer produces a sequence that still contains blank labels. To get the final result we simply remove all the blank labels in the sequence of labels.

In subsequent sections, we will characterize the performance of the CPU implementation of this network in terms of speed and data movement.

3.1 Software & Hardware Partitioning

The original software implementation of Scrappie has been benchmarked in order to obtain the time spent computing the results of each neural-network layer, and subsequent

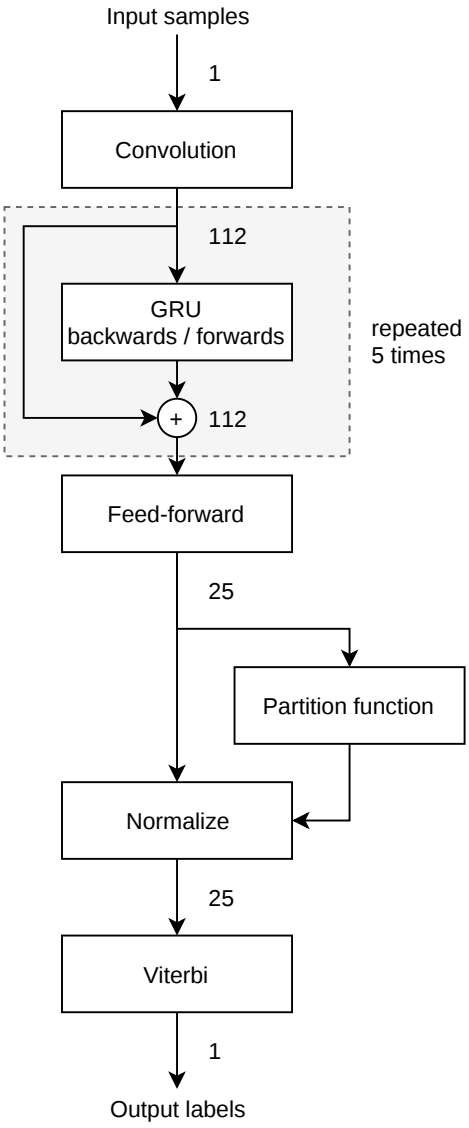


Figure 3.1: RNNRF R94 Model

decoding steps. The result of this is visible in table 3.1. The GRU layers take by far the most time (93.87%), and is the only type of layer that should be accelerated considering just the amount of time taken per layer.

Suppose we would accelerate just the GRU layers on the DFE. Data would enter and leave the GRU layers through PCIe streams. From figure 3.1 we can see that GRU layers take streams of vectors of 112 elements, and produce vectors of the same size. If we assume a clock frequency of $f_{clk} = 300MHz$, 1 DSP per multiplication, and 32 bits per element, then the amount of elements that can be streamed in/out of a GRU layer is given by equation 3.1. Note that PCIe bandwidth in both directions simultaneously for a MAX5C is 24 Gb/s.

Layer	Single-threaded time (seconds)	Multi-threaded time (seconds)	% Of total time
Convolutional Layer	7.09	1.12	1.46%
GRU Backwards / Forwards	455.11	52.51	93.87%
Feed-forward	3.68	0.58	0.76%
CRF Partition	17.49	2.77	3.61%
CRF Normalize	0.19	0.03	0.04%
Decode CRF (Viterbi)	1.25	0.20	0.26%
Total	484.82	76.86	

Table 3.1: Profiling results and derived metrics

$$Bandwidth_{PCIe} = n \times BitsPerElem \times f_{clk} \tag{3.1}$$
$$\Rightarrow\ n = \frac{Bandwidth_{PCIe}}{BitsPerElem \times f_{clk}} = 2.5 \text{ Elements/tick}$$

In chapter 4, where the performance model is described, it becomes clear that the amount of DSPs is the limiting factor in terms of hardware resources. From section 2.2.4 we know that we need $N_{matmul} = 6$ square matrix-vector multipliers, which will make up the bulk of the computations of this layer. An $m \times m$ matrix-vector multiplication takes m^2 multiplications per input vector of size m. If we process $n < m$ elements per clock-tick, rather than the whole vector, we need $m \times N_{matmul}$ multipliers per element, hence the amount of DSPs needed is given by $n \times m \times DspsPerMult \times N_{matmul}$.

In the RNNRF R94 network, $m = 112$. If we assume 1 DSP per multiplication, and we use all DSPs on the MAX5C ($N_{dsp} = 6{,}840$), then the amount of elements n we can stream in/out is given by equation 3.2.

$$n = \frac{N_{dsp}}{N_{matmul} \times m} \approx 10.18 \text{ Elements/tick} \tag{3.2}$$

From equations 3.1 and 3.2 it is clear that we would be bottlenecked by PCIe speed. Therefore it is advantageous to implement more layers on the DFE, if only to reduce the input and output bandwidth requirements. The GRU layer would still have the same bandwidth requirement, but would read from / write to DDR instead of PCIe, which supports much higher bandwidths.

The layer before the GRU layers is the convolutional layer. It produces data of width 112, but only takes a single input element, thus effectively having a factor 112 lower input rate. This would remove the PCIe bottleneck on the input side completely.

Similarly, the feed-forward layer after the final GRU layer compresses data by a factor $\frac{112}{25} = 4.48$, resulting in an output rate of approximately 2.27 elements/tick. We could

further reduce the output rate by implementing (part of) the Viterbi algorithm, which could reduce output size by another factor of 25, for an overall reduction of a factor 112 compare to the GRU layer output rate. The Viterbi algorithm has not been accelerated in this work, due to the simple fact that it would take too much time to implement.

In the next chapter (chapter 4) the performance of the layers is modeled in more detail.

3.2 High-Level Design

To understand the manager-level design, let us first examine the steps the hardware has to perform to implement the network:

1. Stream data from the CPU through the convolutional layer, to DDR;
2. Stream data from DDR through a GRU layer, to DDR again. This step is repeated five times;
3. Stream data from DDR through the feed-forward layer, to the CPU.

Note that these are consecutive steps, therefore none of the layers run concurrently.

Figure 3.2 gives an overview of the manager-level design. The layer implementations reside in the neural network kernel (red), which communicates with the CPU and DDR. Since data from DDR has to be read in reverse order in the even numbered GRU layers, data reordering kernels are added, which reorder bursts of data received from DDR before they are forwarded to the neural network kernel. Read/write addresses for the DDR are generated by the read- and write address generator kernels. The address generator kernels are controlled by sending instructions from the CPU over PCIe, shown here in green. The CPU and neural network kernel both need to be able to write to DDR, so a write kernel is put into place to multiplex the corresponding data streams. Each of these blocks will be explained in more detail in chapter 5.

3.2.1 Application of the Design Methodology

The performance aspects of the design were first modeled, the results of which are presented in chapter 4. Subsequently, the complete design is modeled in software. This means that the layers that are accelerated have been rewritten to more closely resemble their hardware implementation. The software model was tested for functional correctness with the original application. Hardware blocks were gradually developed, and tested against the software model implementations in simulation. The final hardware implementation was tested against the software implementation in simulation, and ultimately in hardware.

The performance model will be discussed in the next chapter (chapter 4), and hardware implementation details in chapter 5. The accuracy of predicated performance will be evaluated in chapter 6.

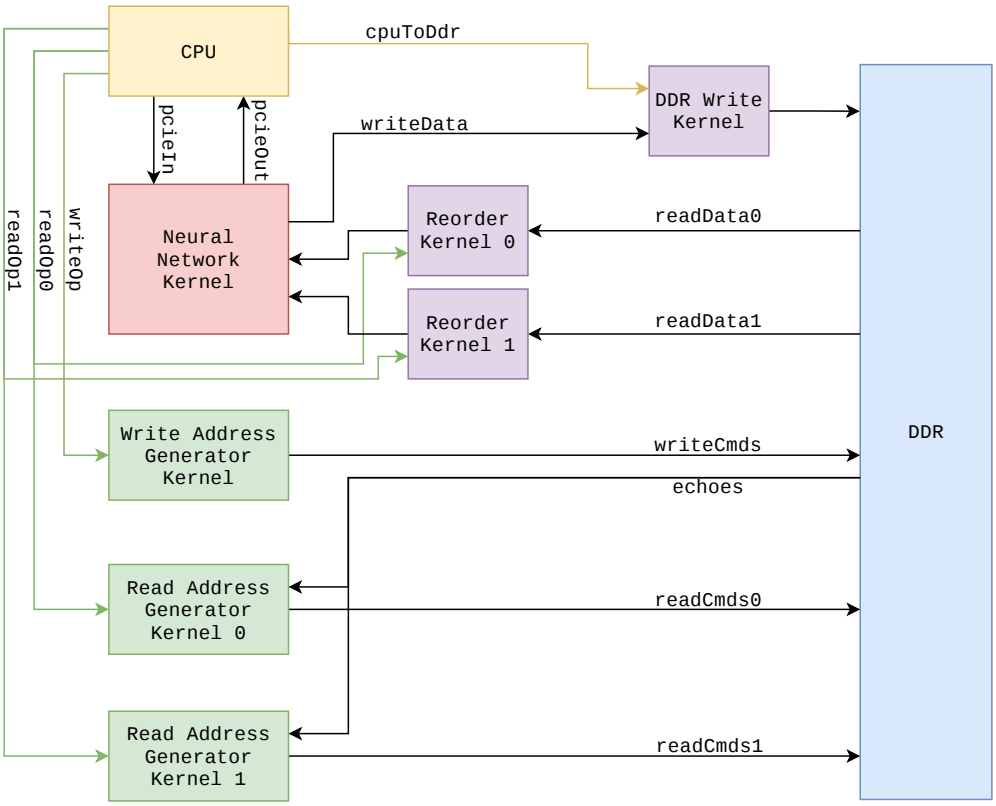


Figure 3.2: Manager-level design

4 Performance Model

Before starting hardware implementation, system performance has been modeled in terms of throughput, area utilization, and IO bandwidth utilization. We will consider each layer separately in the following sections. In the last section we will combine all information obtained in earlier sections to estimate how many copies of the design can be instantiated on the chip. Total system performance can then be estimated.

4.1 Area Utilization

In this section, the area cost in terms of lookup tables (LUTs), flip-flops (FFs), Block RAMs (BRAMs), and Digital signal processing blocks (DSPs) is estimated. We will do so first for all major components individually, and add up the total in the end.

Some resources (except DSPs) will be used by MaxCompiler to generate logic, e.g.: control logic, FIFOs between kernels, the MCP, PCIe controller etc. Experience has shown that reserving about 20% of the chip (excluding DSPs) for compiler generated logic is sufficient.

4.1.1 Elementary Blocks

There are some elementary modules (e.g., FIFOs, serial to parallel converters) and operations (e.g., additions, multiplications) that are used to build most of the hardware needed for the neural network. The resource costs associated with each of these is approximated by synthesis of a small representative design, or with an analytical formula.

Arithmetic Operations

Addition/subtraction, multiplication and division operations have been synthesized with 27-bit integer operands to get an approximation of the resources used per operation. 27-bits were chosen since this seemed a reasonable upper bound to the amount of bits used in the internal fixed point format at the time. Table 4.1 shows the cost of each type of arithmetic operation.

	LUTs	FFs	BRAMs	DSPs
Addition / subtraction	27	28	0	0
Multiplication	6	17	0	2
Division	824	4,702	0	0

Table 4.1: Resource costs per elementary operation

	LUTs	FFs	BRAMs	DSPs
Exponential function	4,653	9,477	9	54

Table 4.2: Exponential function area utilization

On-chip Memory

On-chip memory is implemented using BRAMs, which have a total size of roughly $18Kb$. BRAM usage is approximated using equation 4.1.

$$BRAMs \approx \left\lceil \frac{BitsUsed}{BitsPerBRAM} \right\rceil \tag{4.1}$$

Serial to Parallel & Parallel to Serial

MaxCompiler lists only flip-flops as being used for serial to parallel converters. It might be the case that it uses part of the LUTs in the same CLB, but the tools do not report anything on it. Cost is approximated using equation 4.2.

$$FFs \approx BitsUsed \tag{4.2}$$

Parallel to serial converters are used a lot with an extra register so the parallel output can be held for multiple ticks while new inputs are already shifted in. This effectively doubles the cost. Otherwise the cost is the same as in equation 4.2.

Exponential Function

Exponential functions used in this design only operate on IEEE single precision floats. As such, an exponential unit that takes and produces floats is synthesized to approximate its cost, as shown in table 4.2.

4.1.2 Matrix-vector Multipliers

We need six 112×112 matrix-vector multipliers, of which one is shared among multiple layers. The cost of a single 112×112 matrix-vector multiplier is given in table 4.3.

	LUTs	FFs	BRAMs	DSPs
112 multiplications	672	1,904	0	224
112 additions	3,024	3,136	0	0
Total	3,696	5,040	0	224

Table 4.3: Matrix-vector multiplier area utilization

	LUTs	FFs	BRAMs	DSPs
5 Serial to parallel convertors	0	15,120	0	0
5 streamHolds	0	15,120	0	0
16 to 112 serial to parallel	0	3,024	0	0
112 wide streamHold	0	3,024	0	0
1 to 112 serial to parallel	0	3,024	0	0
112 wide streamHold	0	3,024	0	0
11 wide MUX	297	0	0	0
112 wide MUX	3,024	0	0	0
Total	3,321	42,336	0	0

Table 4.4: Selection logic area utilization

The total cost of all matrix-vector multiplier is predicted by simply multiplying these amounts by six.

The cost of the selection logic for the shared matrix-vector multiplier are approximated analytically using the equations discussed before. The resulting area utilization is given in table 4.4. The total resource utilization of 6 matrix-vector multipliers and selection logic is given in table 4.5.

4.1.3 Weight & Bias Memory

The weight and bias memory sub-system contain three bias memory blocks, and six weight memory blocks.

In addition, six serial to parallel converter and a parallel to serial converter are needed to convert the input stream from DDR to the correct width of the memories. The cost all of these components is given in table 4.6.

	LUTs	FFs	BRAMs	DSPs
6 Matrix-vector multipliers	22,176	30,240	0	1,344
Selection logic	3,321	42,336	0	0
Total	25,497	72,576	0	1,344

Table 4.5: Matrix-vector multipliers + selection logic total cost

	LUTs	FFs	BRAMs	DSPs
6x weight memory	0	0	114	0
3x bias memory	0	0	3	0
Parallel to serial	0	3,024	0	0
6x serial to parallel	0	18,144	0	0
Total	0	21,168	117	0

Table 4.6: Weigh & bias memory area utilization

	LUTs	FFs	BRAMs	DSPs
5 Additions / subtractions	135	140	0	0
3 Multiplications	18	51	0	6
3 Exponential functions	13,959	28,431	27	162
Total	14,146	28,746	27	168

Table 4.7: GRU Layer resource utilization

4.1.4 GRU & Residual Layers

Most of the cost of the GRU layer is in the matrix-vector multipliers, which are not counted here. All other operations are either addition, subtraction, multiplications, divisions or exponential functions.

We simply count all the elementary operations in the GRU layer (excluding matrix-vector multiplications), and the residual addition to get an estimate of its resource utilization, which is shown in table 4.7.

4.1.5 Convolutional Layer

The convolutional layer uses a single matrix-vector multiplier, the cost of which is not counted here. The window selection logic consists mainly of a set of multiplexers and a FIFO. Resource usage for these parts is calculated as follows:

$$LUTs_{mux} \approx BitsPerElem \times WindowSize \tag{4.3}$$

$$FFs_{FIFO} \approx BitsPerElem \times ILD \tag{4.4}$$

Where $BitsPerElem$ is the amount of bits of the internal fixed point type (27), $WindowSize$ is 11, and ILD is the interleave degree (5). The actual cost is shown in table 4.8.

4.1.6 Feed-forward Layer

All resource costs associated with the feed-forward layer are considered part of the shared matrix-vector multiplier, and are given in the corresponding section.

	LUTs	FFs	BRAMs	DSPs
Multiplexers	297	0	0	0
FIFO	0	1,485	0	0
Total	297	1,485	0	0

Table 4.8: Convolutional layer cost

	LUTs	FFs	BRAMs	DSPs
Multiplexer	512	0	0	0
Burst memory	0	0	1	0
Total	512	0	1	0

Table 4.9: Reorder kernel resource utilization

4.1.7 Reorder Kernel

The reordering kernel consists mainly of a memory capable of storing two complete size 112 vectors, and a burst-size (512 bits) width multiplexer. Total predicted resource utilization for the reorder kernel is given in table 4.9.

4.1.8 Overall Area Utilization

Area utilization of all parts are added up in table 4.10, along with percentages of total resources used. These numbers will be used in section 4.5 to estimate how many instances of the design can be put on the chip to run in parallel.

4.2 Bandwidth Utilization

This section presents the input and output bandwidth utilization of the different layers in the network. We will characterize PCIe and DDR input/output rates per layer in the following sub-sections. In the end, we will summarize the resulting bandwidths.

	LUTs	FFs	BRAMs	DSPs
Matrix-vector multipliers	25,497 (2.16%)	72,576 (3.07%)	0 (0%)	1,344 (19.65%)
Weight & Bias Memories	0 (0%)	21,168 (0.9%)	117 (2.71%)	0 (0%)
GRU & Residual Layers	14,112 (1.19%)	28,622 (1.21%)	27 (0.63%)	168 (2.46%)
Convolutional Layer	297 (0.03%)	1,485 (0.06%)	0 (0%)	0 (0%)
Reorder Kernel	512 (0.04%)	0 (0%)	1 (0.02%)	0 (0%)
Total	40,418 (3.42%)	123,851 (5.24%)	145 (3.36%)	1,512 (22.11%)

Table 4.10: Total area utilization

4.2.1 Convolutional Layer

The convolutional layer reads data from PCIe, and writes results to DDR. Recall that the convolutional layer performs the function:

$$\begin{aligned} \vec{y} &= W\vec{k} + b \\ \vec{y} &\text{ is } N_y \times 1 \ (112 \times 1) \\ W &\text{ is } N_y \times N_k \ (112 \times 11) \\ \vec{k} &\text{ is } N_k \times 1 \ (11 \times 1) \end{aligned} \tag{4.5}$$

One element of the feature vector $\vec{y}$ is produced every tick, which means a new window $\vec{k}$ should be ready every 112 ticks. A window is updated by shifting one element into the previous window. Therefore, the input bandwidth needed is given by equation 4.6. The output bandwidth is given by equation 4.7.

$$PCIe_{in} = \frac{1}{N_y} \times bitsPerElem \times f_{clk} \tag{4.6}$$

$$DDR_{out} = 1 \times bitsPerElem \times f_{clk} \tag{4.7}$$

4.2.2 GRU & Residual Layer

The GRU layer takes a single element, and produces a single element each tick. Its input and output bandwidth are therefore straightforward, and are given by equations 4.8 and 4.9. Note that the maximum bandwidth between the input and output to/from DDR is shared. The total bandwidth is the sum of the two (equation 4.10).

$$DDR_{in} = 1 \times bitsPerElem \times f_{clk} \tag{4.8}$$

$$DDR_{out} = 1 \times bitsPerElem \times f_{clk} \tag{4.9}$$

$$DDR_{tot} = DDR_{in} + DDR_{out} \tag{4.10}$$

4.2.3 Feed-forward Layer

The feed-forward layer reads data from DDR and writes to PCIe. The layer performs the following computation:

$$\begin{aligned} \vec{y} &= W\vec{x} + b \\ \vec{y} &\text{ is } N_y \times 1 \ (25 \times 1) \\ W &\text{ is } N_y \times N_x \ (25 \times 112) \\ \vec{x} &\text{ is } N_x \times 1 \ (112 \times 1) \end{aligned} \tag{4.11}$$

	PCIe in (Mb/s)	PCIe out (Mb/s)	DDR total (Mb/s)
Convolution	68.12 (0.3%)	0 (0%)	7,629.39 (2.22%)
GRU	0 (0%)	0 (0%)	15,258.79 (4.44%)
Feed-forward	0 (0%)	7,629.39 (33.33%)	34,179.69 (9.96%)
Max	68.12 (0.3%)	7,629.39 (33.33%)	41,809.08 (12.18%)

Table 4.11: PCIe/DDR Bandwidth utilization per layer

As discussed in the implementation section of this layer, one element is produced every tick, and $\frac{N_x}{N_y}$ elements is read every tick. Input and output badwidth can therefore be calculated using equations 4.12 and 4.13.

$$DDR_{in} = \frac{N_x}{N_y} \times bitsPerElem \times f_{clk} \tag{4.12}$$

$$PCIe_{out} = 1 \times bitsPerElem \times f_{clk} \tag{4.13}$$

4.2.4 Bandwidth per Layer

The PCIe/DDR bandwidths of each layer is summarized in table 4.11. The last row of the table is the maximum bandwidth among all layers. This information is used in section 4.5 to estimate how the design will scale with multiple instances.

4.3 Memory Utilization

4.3.1 Weight & Bias Memory Utilization

Weights and biases are stored in DDR and are loaded into on-chip memory before a layer starts processing sequence data. Each layer will need at most $6 \times 112 \times 112 \times BPE$ bits of weight memory, and $3 \times 112 \times BPE$ bits of bias memory, where BPE is the amount of bits per element in memory (32). This adds up to a total of $2.4Mb + 10kb \approx 2.4Mb$ of memory.

4.3.2 Layer Memory Utilization

At any time, only one layer is actively processing data. Data of the layer preceding layer must be kept in memory until the active layer finishes processing. The active layer can also produce data and write it to DDR. For each layer, we can compute the memory utilization using equation 4.14. In this equation ILD is the interleave degree (5), and BPE is the amount of bits per element (32). w_{in} and w_{out} are the input and output vector widths respectively. Since ILD sequences are interleaved, n is the length of the longest of these sequences.

$$M_{layer}(n, w_{in}, w_{out}) = n \times ILD \times BPE \times (w_{in} + w_{out}) \tag{4.14}$$
$$M_{conv}(n) = M_{layer}(n, 0, 112) \tag{4.15}$$
$$M_{GRU}(n) = M_{layer}(n, 112, 112) \tag{4.16}$$
$$M_{FFL}(n) = M_{layer}(n, 112, 0) \tag{4.17}$$

Equations 4.15–4.17 show the memory usage per layer. Note that the width of the input to the convolutional layer and the output of the feed-forward layer have been set to zero, since the data associated with these inputs/outputs are not stored in DFE memory, but are streamed through PCIe. It is clear that the GRU layer will be the limiting factor. To obtain the maximum sequence length, we solve $M_{GRU} = 48GB - 2.4Mb$ for n, and obtain a maximum sequence length of $n_{max} \approx$ 10,714,218.

4.4 Throughput Prediction

4.4.1 Throughput per Layer

The network receives a raw sequence as input of length n. For each sample, a vector of a layer-specific size is produced by any layer. In general, the amount of time spent by a layer to produce all its output vectors (and thus its total runtime) is given by equation 4.18. In this equation n is the sequence length, f_{clk} is the compute kernel clock frequency, and TPO is the amount of clock ticks spent to produce a single output vector.

From our general equation, we can derive time spent for each layer, and the total amount of time spent by the network on a sequence of length n. These are shown in equations 4.19–4.22.

$$T_{layer}(n, TPO) = n \times \frac{TPO}{f_{clk}} \tag{4.18}$$
$$T_{conv}(n) = T_{layer}(n, 112) \tag{4.19}$$
$$T_{GRU}(n) = 5 \times T_{layer}(n, 112) \tag{4.20}$$
$$T_{FFL}(n) = T_{layer}(n, 25) \tag{4.21}$$
$$T_{DFE}(n) = T_{conv}(n) + T_{GRU}(n) + T_{FFL}(n) \tag{4.22}$$

Of the total time spent processing a sequence, we can derive the percentage of time spent for each layer. Because the run-time of each layer is linear with respect to n, these simplify to constant percentages as shown in equations 4.23–4.25.

$$T_{conv\%} = \frac{100\% \times T_{conv}(n)}{T_{DFE}(n)} \approx 16.07\% \tag{4.23}$$

$$T_{GRU\%} = \frac{100\% \times T_{GRU}(n)}{T_{DFE}(n)} \approx 80.34\% \tag{4.24}$$

$$T_{FFL\%} = \frac{100\% \times T_{FFL}(n)}{T_{DFE}(n)} \approx 3.59\% \tag{4.25}$$

4.5 Total Resource Utilization Estimate

The FPGA on the MAX5C contains multiple SLRs as discussed in section 2.5.4. Because SLR crossings are costly, and 1 DIMM is connected to each SLR, it is a good idea to instantiate the entire design one or more times per SLR.

Table 4.12 shows resource utilization, bandwidth utilization per layer, and a speedup compared to the profiling results in chapter 3. T_{accel} is the predicted time that the DFE part of the application takes (in seconds). S_{accel} is the speedup of the accelerated part over the same layers in software. S_{tot} is the speedup of the complete accelerated system compared to the original software implementation, and is approximated as follows:

$$S_{tot} = \frac{T_{accel} + T_{prof,tot} - T_{prof,accel}}{T_{prof,tot}} \tag{4.26}$$

where $T_{prof,tot}$ is the total runtime of the profiling run and $T_{prof,accel}$ is the runtime of the part that is to be accelerated of the profiling run.

The feed-forward layer is bottle-necked by the PCIe output bandwidth at more than 3 instances, indicated by the percentages marked in red. Overall, the design is limited by the amount of DSP blocks available at 4 or more instances. For an initial implementation, we chose to instantiate 3 copies of the design, yielding a predicted speedup of 3.31 times.

No. instances	LUT%	FF%	BRAM%	DSP%	Conv BW%	GRU BW%	FFL BW%	Peak DDR%	T_{accel}	S_{accel}	S_{tot}
1	3.42	5.24	3.36	22.11	2.22	4.44	33.33	17.39	57.75	1.29	1.27
2	6.84	10.48	6.71	44.21	4.44	8.89	66.67	34.77	28.88	2.59	2.36
3	10.26	15.71	10.07	66.32	6.67	13.33	100.00	52.16	19.25	3.88	3.31
4	13.68	20.95	13.43	88.42	8.89	17.78	133.33	69.54	14.44	5.18	4.13
5	17.10	26.19	16.78	110.53	11.11	22.22	166.67	86.93	11.55	6.47	4.86
6	20.52	31.43	20.14	132.63	13.33	26.67	200.00	104.31	9.63	7.77	5.51
7	23.94	36.67	23.50	154.74	15.56	31.11	233.33	121.70	8.25	9.06	6.09
8	27.36	41.90	26.85	176.84	17.78	35.56	266.67	139.08	7.22	10.35	6.61
9	30.78	47.14	30.21	198.95	20.00	40.00	300.00	156.47	6.42	11.65	7.08

Table 4.12: Resources utilization for varying number of instances

5 Hardware Implementation

In this chapter, we will present hardware implementation of the complete system in detail. In addition, we will discuss the software side of the system, namely reading sequence data and streaming it do the DFE, and streaming the result from the final layer back to the CPU.

5.1 Neural-Network Kernel

Figure 5.1 shows how the different layer implementations are connected to inputs and outputs. The first layer is the convolutional layer. Data is sent from the CPU through PCIe directly into the convolution layer. Convolution results are written to LMEM. The subsequent GRU layers read from, and write to LMEM. After the final GRU layer, the feed-forward layer reads data from LMEM and writes output data directly to the CPU through PCIe.

Additionally, there is an on-chip memory block that is initialized with weights and biases from LMEM just before any of the layers start processing data.

Not shown here is the control logic, and the shared matrix-vector multiplier. Control logic is implemented as a finite state machine (FSM). The relevant parts of the FSM are explained in their respective sections. Matrix-vector multipliers are used by all layers.

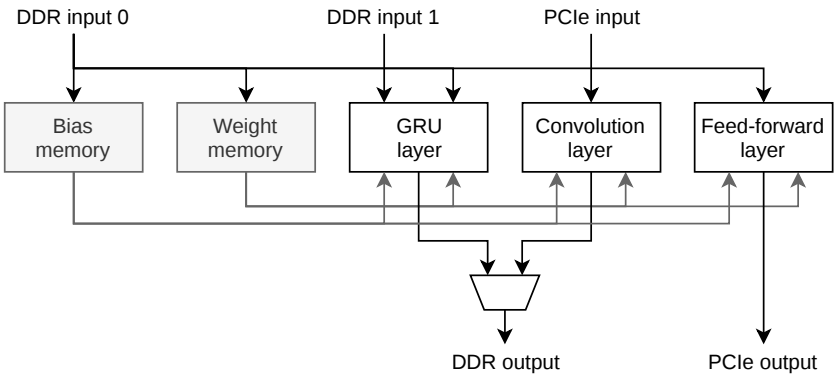


Figure 5.1: Neural-network kernel internal organization

5.1.1 General Layer Operation

All layers roughly take the following steps:

1. Read weights & Read biases;
2. Stream data through computation pipeline;
3. Flush.

Recall that many layers perform linear transformations ($y = Wx$), or affine transformations ($y = Wx + b$), with constant weight matrices W and bias vectors b. These weights and biases should be loaded into the on-chip weight/bias memory before the layer that needs them can begin computation.

All weights and bias vectors are written to DDR by the CPU once. After that, Each time a different layer start processing, the neural network kernel will read weights and biases from DDR and store them in the on-chip weight/bias memory.

When weights and biases are loaded, computation can start. What happens here is different per layer, and will become clear in subsequent sections.

When a layer finishes processing, all data is flushed from the kernel before the next layer starts. To see why flushing is necessary, we consider the alternative: Suppose a layer (layer 1) is processing input data, and is writing output data at the same time. When layer 1 finishes, some data will still be in the internal pipeline of the neural network kernel. The next layer (layer 2) wants to start reading data, but has to wait for layer 1 to write all its output data. Layer 1 will never write its output data because no new data is entering the pipeline, since layer 2 is waiting. This results in a deadlock. The solution is to flush data from the pipeline after any layer finishes processing.

5.1.2 Weight & Bias Memory

Figure 5.2 shows a diagram of the bias and weight memory system. There are three bias memory blocks, and six weight memory blocks. Bias memory is one element wide, whereas weight memory is 112 elements wide to match the 112 by 112 matrix-vector multiplier input. Data from DDR arrives in the neural network kernel in bursts of 16 elements. Six serial to parallel converters, and one parallel to serial converter convert the data to the correct width. All bias memories share the same write address, and the same read address. Same goes for the weight memories. Read addresses are generated by layer specific control logic. Write addresses are generated by the weight and memory programming logic, which will be explained shortly.

Not shown here are several other control signals. The weight memory blocks are each connected to a write enable line, contained in bit-vector `weightMemoryEnables`, where `weightMemoryEnables[i]` write-enables the i'th memory block. The bias memories have similar control signals, called `biasMemoryEnables`. In addition, the serial to parallel converter has an enable signal `serialToParallelEnable`.

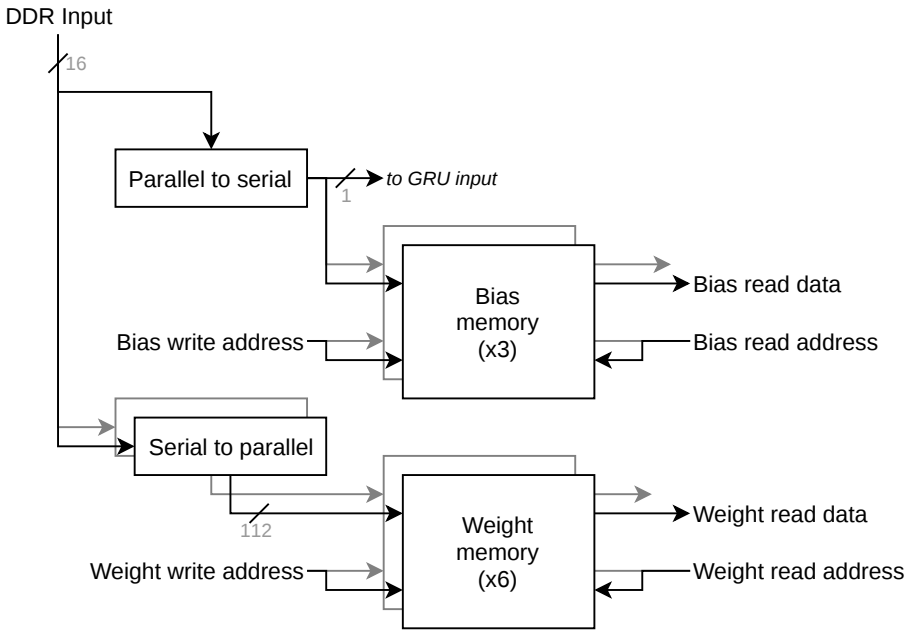


Figure 5.2: Weight & bias memory

Programming the Weight Memory

Listing 5.1 shows the pseudo-code used to control the programming of the weight memory. Memory blocks are programmed in sequence. For each memory block, rows are written one after another. for each row, we have to wait $burstsPerRow = 7$ ticks until an entire weight matrix row is in the serial to parallel register.

Listing 5.1: Weight memory programming control

```
1  for weightMemIndex = 0...weightMemCount do
2      for rowIndex = 0...weightMemWidth do
3          for 0...burstsPerRow do
4              matMemEnables ← 0
5          end
6
7          matMemEnables ← 1 ≪ memBlockIndex
8          weightWriteAddress ← rowIndex
9      end
10 end
```

Programming the Bias Memory

The control logic for programming biases is very similar, and is shown in listing 5.2. The only significant difference is that we do not have to wait for a serial to parallel register to fill up.

Name	No. instances	Input size	Output size
Convolution	1	11	112
GRU	6	112	112
Feed-forward	1	112	25

Table 5.1: Matrix-vector multiplier input/output sizes

Listing 5.2: Bias memory programming control

```
1 for biasMemIndex = 0...biasMemCount do
2     for elemIndex = 0...biasMemDepth do
3         biasMemEnables ← 1 ≪ biasMemIndex
4         biasWriteAddress ← elemIndex
5     end
6 end
```

5.1.3 Matrix-vector Multipliers

Matrix-vector multiplications are used extensively in the network. All matrix-vector multiplications are performed with constant weight matrices, which are stored in on-chip memory.

Table 5.1 gives an overview of matrix-vector multiplications used in all the different layers. The GRU layers need 6 matrix-vector multipliers that will be used simultaneously, so they cannot share hardware. These multipliers are the biggest ones, with size 112 by 112. Convolution can effectively be implemented as a matrix-vector multiplication, as seen in section 2.2.3, by multiplying a weight matrix with input windows to create feature vectors. One of the multipliers used in the GRU layer is reused in the convolution layer. Finally, the feed-forward layer uses a 25 by 112 multiplier. The same multiplier that is reused in convolution is also reused here.

Square Matrix-vector Multiplier Design

Figure 5.3 shows the design of an $n \times n$ (with $n = 4$ in this case) matrix-vector multiplier design. In this design, the matrix rows are stored in on-chip memory. The address to this memory is incremented every tick, starting at 0, and going up to $n - 1$. The input vector at the top should be kept stable for n ticks for the entire computation to complete. Elements of the output vector are produced one element at a time.

Input Selection

Since one matrix-vector multiplier is shared between the convolution, GRU, and feed-forward layers, input data has to be selected to the multiplier at the right time. Figure 5.4 shows the design for this selection mechanism.

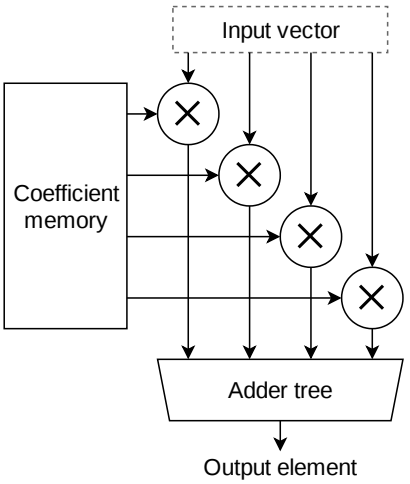


Figure 5.3: 4×4 Matrix-vector multiplier design

Input to the matrix-vector multiplier from the GRU layer arrives element by element. Since the multiplier needs a whole vector for 112 ticks, the serial data goes into a serial to parallel converter, the output of which is held in by a register every 112 ticks.

Windows coming from the convolution layer can be selected by a multiplexer. Another multiplexer can select feed-forward layer data. This data arrives in bursts of 16 elements, and has its own serial to parallel converter and stream hold. The reason for this will be explained in section 5.1.6.

The output from the matrix-vector multiplier is fanned out to all the different layers.

5.1.4 GRU & Residual Layers

Figure 5.5a and 5.5b show forwards and backwards layers respectively, as they are implemented in the original application. The DDR nodes are annotated with the order in which sequence data is stored (normal- or reverse order). Red edges mean that sequence data is read in reverse, in contrast to the black edges.

In the original application, data is always stored in the same order, and is either read in forwards order (forwards GRU layers), or in reverse order (backwards GRU layers). To exploit pipelining between the GRU and residual layers, we would like to eliminate the need for the memory block in between them. This is possible by changing the way that data is read.

To understand the modified design, we first turn our attention to the backwards layer, shown in figure 5.5d. We will assume that input data to a backwards layer is stored in normal order, because this is the order in which the preceding convolution layer will store it. Data is read in reverse order into the GRU layer. Output data is streamed from the GRU layer directly into the addition node, which will need to add the original input data to it. Since data from the GRU layer is reversed, so is the other input to

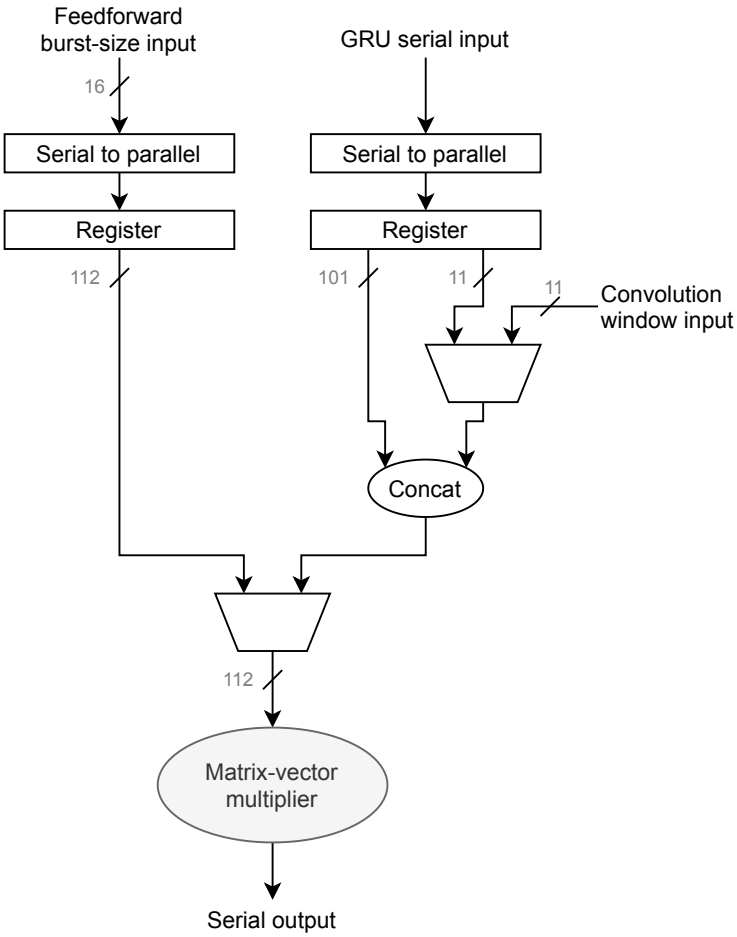
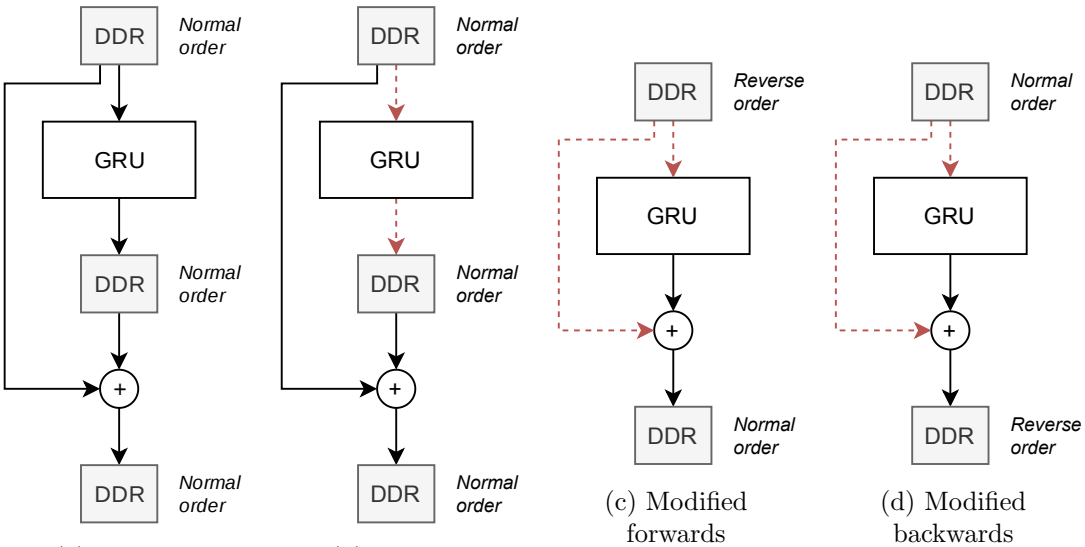


Figure 5.4: Matrix-vector multiplier input selection

the addition. From the addition, data is streamed into memory again. This means that data is stored in reverse order after a backward GRU layer.

Forwards GRU layers are always preceded by a backwards GRU layer in this design. Therefore, its input data will be in reverse order. Since it needs its data in normal order, it reads data in reverse. This means that the second input to the addition node also needs to be read in reverse. Coincidentally, the forwards and backwards layers have the same read/write characteristics. Note that the final GRU layer is a backwards layer. Therefore, data is stored in reverse order, and must be read in reverse by the succeeding feed-forward layer.

In the following sub-sections, we will look at the initial GRU layer design to explain the basics of operation of a GRU layer. We will then introduce an improved GRU layer design which improves on the shortcomings of the initial design.



(a) Original forwards (b) Original backwards (c) Modified forwards (d) Modified backwards

Figure 5.5: Original & modified GRU layer designs (red edges mean that data is read in reverse order)

Initial Design

Recall that GRU layers produce output $h(t)$ from inputs $x(t)$ and $h(t-1)$. Figure 5.6 shows the design of the initial implementation of the GRU layer. Inputs $x_i(t)$ and $h_i(t-1)$ denote the i'th element of vectors $x(t)$ and $h(t-1)$ respectively, i.e.: Inputs are streamed in element by element. This is to match the way that data is streamed through the major components of the layer: The matrix-vector multipliers. The feedback of $h_i(t)$ is realized with a FIFO of depth $vecSize$, where $vecSize$ is the size of vector x and vector h. This way, $h(t-1)$ is effectively looped back as an input. Additionally, i is used as an address to the on-chip weight memory to read rows from the weight matrices, which are used by the matrix-vector multipliers.

Listing 5.3 shows pseudo-code that expresses the same computation as figure 5.6. Here we can see clearly how the control signals i and t are generated. The control logic implementing the loops is implemented in the neural network kernel FSM. The GRU equations from section 2.2.4 are implemented in function $gruStep$.

Listing 5.3: Initial GRU implementation

```
1 function gru(x) ⇒ h
2     for t = 0...seqLen do
3         for i = 0...vecSize do
4             if t = 0 then
5                 hPrev ← 0
6             else
7                 hPrev ← streamOffset(h, −vecSize)
8             end
```

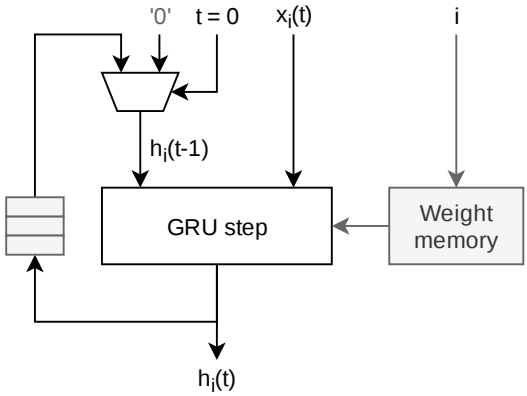


Figure 5.6: GRU, non-interleaved

```
9                h ← gruStep(hPrev, x, weights[i])
10           end
11       end
12   end
13
14   function gruStep(hPrev, x, weights) ⇒ h
15       z ← σ(weights.W_z × x + weights.U_z × hPrev)
16       r ← σ(weights.W_r × x + weights.U_r × hPrev)
17       h' ← tanh(weights.W_h' × x + r × weights.U_h' × hPrev)
18       h ← z × hPrev + (1 − z) × h'
19   end
```

Interleaved Design

In order for $h(t-1)$ to be valid at the start of the computation to produce $h(t)$, we need the total latency of the loop to be equal to $vecSize$:

$$\begin{aligned} & T_{fw} + T_{bw} = vecSize \\ \Rightarrow\ & T_{bw} = vecSize - T_{fw} \geq 0 \end{aligned} \tag{5.1}$$

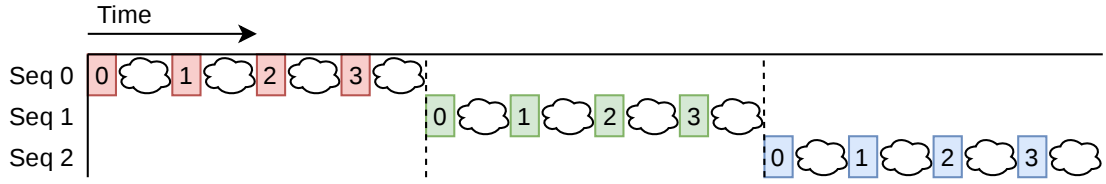
Where the forwards latency T_{fw} contains the multiplexer and the GRU step block, and the backwards latency T_{bw} contains just the FIFO.

Due to the amount of relatively time consuming arithmetic computations in $gruStep$, the forward latency is greater than $vecSize$, causing T_{bw} to become negative. If T_{bw} is negative, the design becomes unschedulable, since a FIFO of negative depth is impossible to implement. The simplest counter measure would be to delay the arrival of input $x(t)$ until $h(t-1)$ becomes valid, introducing bubbles in the pipeline. This is visualized in figure 5.7a. Another counter measure, is to interleave multiple input sequences.

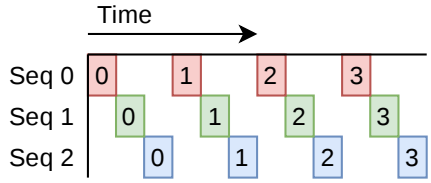
The principle of interleaving is explained by example in figure 5.7. In the first case, three different sequences are processed in sequence. Note that there are no data dependencies

between the three sequences. Each colored rectangle in the figure represents a vector in the sequence. Like explained in the last paragraph, we require bubbles in the pipeline to wait for $h(t-1)$. In these bubbles we could, instead of waiting, take an input from a different sequence and work in that in the meantime, like shown in figure 5.7b.

In the interleaving example, three sequences are interleaved, and the length of each sequence is the same (4). In our actual design, T_{fw} is so big that the design becomes schedulable only if we interleave at least five sequences. In addition, sequences are not necessarily of equal length. This means that all but the longest sequence must be padded to make interleaving possible. This also has implications for the other layers, as will become evident in their respective sections.



(a) Non-interleaved



(b) Interleaved

Figure 5.7: Example of interleaved sequences

Modified pseudo-code for the interleaved version of the GRU layer is shown in listing 5.4. There is another for-loop to iterate over the ILD different interleaved sequences. Sequence lengths are stored in the array $seqLens$, where $seqLens[i]$ contains the length of the i'th sequence. Since input data is read in reverse order in backwards GRU layers, the i'th sequence length is stored at $seqLens[ILD-1-i]$ in this case. In addition, all but the longest sequence is padded at the end in forwards layers. When data is read in reverse order in the backwards layers, padding appears at the beginning of the reversed sequence. Therefore the $inputValid$ signal is determined differently in the case of a backwards layer.

Zeroes are streamed out as padding when the input is not valid, instead of the result of $gruStep$. In the backwards case – when padding is at the front – this also serves as a way to stream in zeroes at the $h(t-1)$ input. The design of the $gruStep$ block does not need to change.

Listing 5.4: Interleaved GRU implementation

```
1 function gruInterleaved(x) ⇒ h
2     for t = 0...paddedMaxSeqLen do
3         for interleaveIndex = 0...ILD do
```

```
4               for i = 0...vecSize do
5                   if isForwardsLayer then
6                       seqLen ← seqLens[interleaveIndex]
7                       inputValid ← t < seqLen
8                   else //backwards layer
9                       seqLen ← seqLens[ILD − 1 − interleaveIndex]
10                      inputValid ← t ≥ paddedMaxSeqLen − seqLen
11                  end
12
13                  if t = 0 then
14                      hPrev ← 0
15                  else
16                      hPrev ← streamOffset(h, −vecSize × ILD)
17                  end
18
19                  if inputValid then
20                      h ← gruStep(hPrev, x, weights[i])
21                  else
22                      h ← 0
23                  end
24              end
25          end
26      end
27  end
```

5.1.5 Convolutional Layer

Since the convolutional layer is followed by a GRU layer that expects interleaved input sequences, we have two options:

1. Implement an interleaved convolutional layer; or
2. Read output from a non-interleaved convolutional layer in an interleaved fashion.

Reading non-interleaved data from DRAM results in a fragmented memory access pattern, which is sub-optimal in terms of throughput. Interleaving the convolutional layer, however, requires input to the convolutional layer to be interleaved. Luckily, input to the convolutional layer comes from the CPU, which is in many cases more efficient with a fragmented memory access pattern. For these reasons the convolutional layer is designed to take and produce interleaved sequences to match the GRU layer input requirements.

In subsequent sub-sections, we will first explain a non-interleaved implementation by example, before explaining the interleaved design.

Non-interleaved Design

As discussed in section 2.2.3, convolution can be implemented by multiplying a weight-matrix with an input window that slides over the input sequence. The sliding window

$window(t)$ is implemented with a shift register, shown in figure 5.8a. In this example the window size is 3. In the actual design, the window size is 11.

Zeroes can conditionally be set in right-most position (when *zeroRight* is high), or in the remaining positions on the left (when $zeroLeft$ is high). The entire window register has a load-enable signal called $winEn$. The input $x(t)$ is shifted into the window register through PCIe. The PCIe stream can be enabled/disabled with the control signal $pcieEn$.

Recall that the first window must contain $x(0)$ in the center, $x(1)$ up to $x(\frac{winLen-1}{2})$ to the right, and zeroes as padding to the left. This means that initially the left side of the register must be set to zero, by setting $zeroLeft$ high, while $x(0)$ up to $x(\frac{winLen-1}{2})$ are shifted in over the course of $\frac{winLen-1}{2}$ ticks to fill up the window. After these initial fill steps, computations can be performed every subsequent time a new $x(t)$ is shifted in. In the last $\frac{winLen-1}{2}$ ticks, *zeroRight* is high to zero-pad windows on the right.

Every time the window is updated, a feature-vector is produced by performing a matrix-vector multiplication, followed by an addition and ELU activation function (equations 5.2 and 5.3). Since a matrix multiplication takes multiple cycles, the window needs to be held in the register for the time being. This is accomplished by setting the $winEn$ and $pcieEn$ signals low during computation.

$$conv(window) = ELU(W \times window + b) \tag{5.2}$$

$$ELU(x) = \begin{cases} x, & \text{if } x \geq 0 \\ e^x - 1, & \text{otherwise} \end{cases} \tag{5.3}$$

Interleaved Design

Figure 5.8b shows the interleaved convolution window design. In this design, the register that was present is replaced with a feedback FIFO of depth a depth equal to the amount of sequence we interleave (ILD). The control signal $winEn$ is now connected to the new FIFO's input enable.

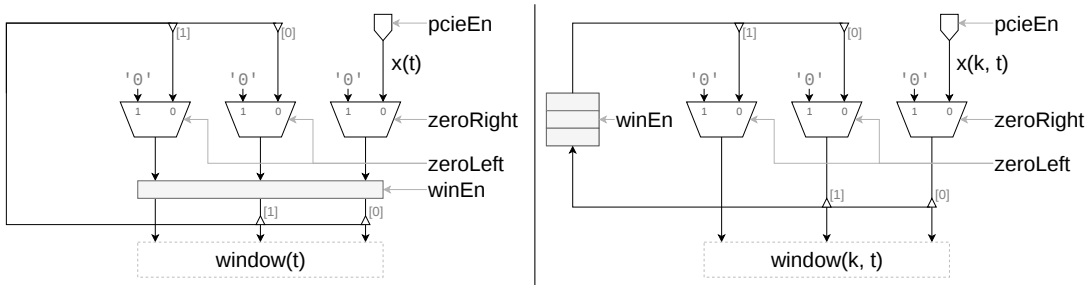
Control logic is implemented in the neural network kernel FSM, where it goes through the following states:

1. Fill windows (listing 5.5);
2. Compute (listing 5.6);
3. Pad right & compute (listing 5.7).

The control logic is mostly the same as explained for the non-interleaved case, except for the addition of the "`for` $k = 0...ILD$ `do` ... `end`" loops to account for interleaving.

Listing 5.5: Fill initial windows state

```
1 for 0...padLen do
2     for k = 0...ILD do
```



(a) Non-interleaved (b) Interleaved

Figure 5.8: Convolution window buffering

```
3           pcieEn ← True
4           winEn ← True
5           zeroLeft ← (t = 0)
6           zeroRight ← False
7       end
8   end
```

Listing 5.6: Process windows state

```
1   for 0...(maxSeqLen − padLen) do
2       for k = 0...ILD do
3           for i = 0...outputWidth do
4               pcieEn ← (i = 0)
5               winEn ← (i = outputWidth − 1)
6               zeroLeft ← False
7               zeroRight ← False
8           end
9       end
10  end
```

Listing 5.7: Process windows & pad right state

```
1   for 0...padLen do
2       for k = 0...ILD do
3           for i = 0...outputWidth do
4               pcieEn ← False
5               winEn ← (i = outputWidth − 1)
6               zeroLeft ← False
7               zeroRight ← True
8           end
9       end
10  end
```

5.1.6 Feed-forward Layer

The feed-forward layer takes an input stream of vectors of size 112. The output is a stream of vectors of size 25. Since the shared matrix-vector multiplier can produce one

element per tick, it takes 25 ticks to produce an output vector. In the same amount of time (25 ticks), 112 input elements must be read

Input to the feed-forward layer comes from DRAM, which is read in bursts of 64 bytes, which fits 16 elements. A burst can be read by the neural network kernel every tick. Therefore we can read 112 input elements in $\frac{112}{16} = 7$ ticks, which fulfils our requirement of reading 112 elements in at least 25 ticks.

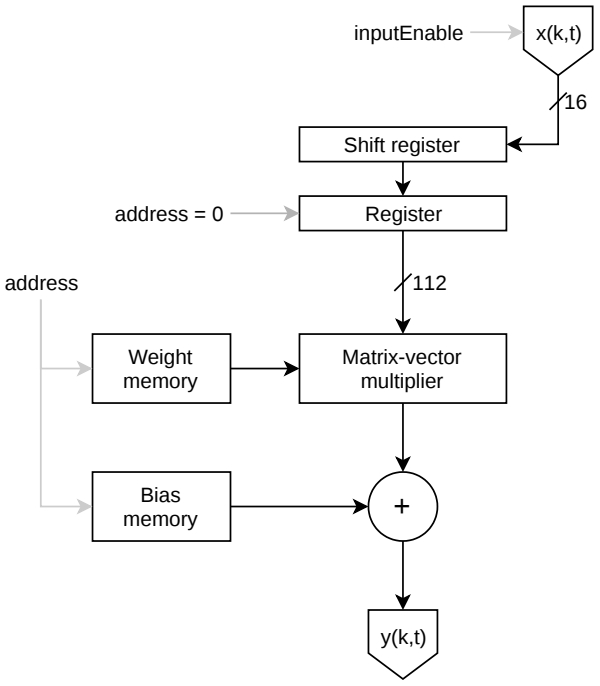


Figure 5.9: Feed-forward design

Figure 5.9 shows the design of the feed-forward layer. Bursts are read into a shift register, which is clocked into the subsequent register only when a new matrix-vector multiplication starts (when $address = 0$). Weights and biases are read from on-chip memory. `inputEnable` is only set high for 7 of the 25 ticks it takes to produce an output vector. The code that controls this process is shown in listing 5.8.

Listing 5.8: Feed-forward control code

```
1  for t = 0...maxSeqLen do
2      for k = 0...ILD do
3          inputSrCounter ← 0
4
5          for address = 0...elemsPerOutputVec do
6              if inputSrCounter < burstsPerInputVec then
7                  inputSrCounter ← inputSrCounter + 1
8                  inputEnable ← true
9              else
10                 inputEnable ← false
11             end
12         end
```

```
13      end
14  end
```

5.2 Address Generators

Recall from section 2.5.3 that in order to stream data from DRAM, commands for the MCP must be generated. These commands are generated based on higher-level memory read/write instructions which are sent by the CPU. In the following sections, the memory instruction format will be explained. Subsequently, the read/write address generator kernels that interpret these instructions are explained.

5.2.1 Memory Instructions

There are two types of memory instructions: Read instructions and write instructions, each of which will be discussed below.

Read Instructions

Read instructions have the following format:

- `address` (31 bits);
- `count` (30 bits);
- `reverse` (1 bit);
- `sync` (1 bit);
- `noop` (1 bit).

The `address` field contains the address (in bursts) to read memory from. `count` is the amount of bursts to read. When `reverse` is high, data should be read in reverse order. When `sync` is high, the read address generator kernel should wait for an echo generated by a write action. This effectively makes the system wait to read data until all previous write actions are completed. When `noop` is high, this operations does nothing, except for (possibly) waiting for an echo if `sync` is also high.

Write Instructions

Write instructions are very similar, but with a small difference:

- `address` (31 bits);
- `count` (30 bits);
- `reverse` (1 bit);
- `sync` (1 bit);

- `tagEnable` (1 bit).

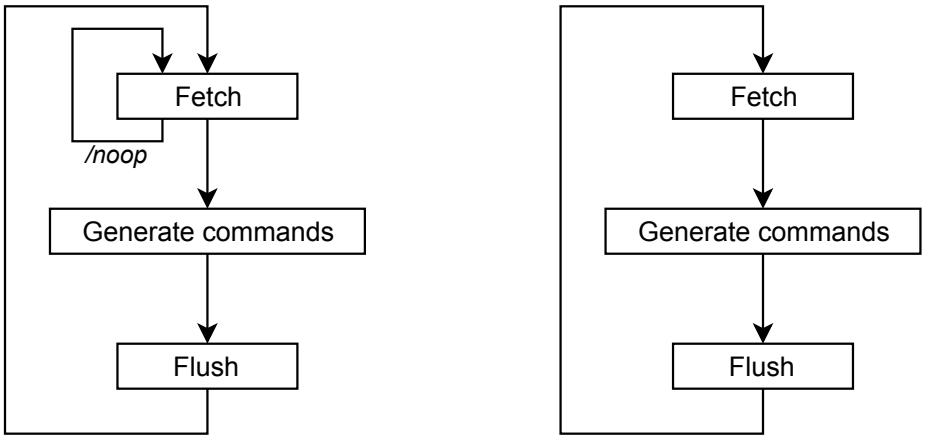
The fields it has in common with read instructions have a similar meaning. Write instructions do not have the `noop` field, but instead contain the `tagEnable` field. When `tagEnable` is high, an interrupt will be raised to the CPU to signify completion of the write instruction. This is not needed during normal operation of the system, but instead is used during testing.

5.2.2 Read Address Generator Kernel

Read address generator kernels take a stream of read instructions, and generate the appropriate MCP commands from each instruction. Additionally, echoes are streamed in to allow read actions to wait for write actions to complete.

Control

Figure 5.10a shows the FSM that controls the read address generator kernel. In the `Fetch` state, a single instruction is read. If `sync` is high, the kernel stalls until an echo is received. If `noop` is high, a new instruction is immediately fetched afterwards. Otherwise, the kernel will start generating commands. After all commands have been generated, the kernel flushes its command stream, and will go back to the `Fetch` state.



(a) Read address generator (b) Write address generator

Figure 5.10: Address generator kernel FSMs

Generating MCP Commands

Recall the MCP command structure:

- `sendEcho` (1 bit);

- `address` (31 bits);
- `size` (8 bits);
- `inc` (8 bit);
- `stream` (15 bits);
- `tag` (1 bit).

For each read instruction, one or more commands are generated as shown in listing 5.9. Commands with `size` set to `burstsPerCommand` will be generated, up to the point where the remaining amount of bursts to be read is less than `burstsPerCommand`. It is advantageous to set `burstsPerCommand` as high as possible, since reading large amounts of bursts at once increases memory throughput.

Listing 5.9: Process windows state

```
1  function generateCommands(instr)
2      offset ← 0
3
4      while offset < instr.count do
5          cmd.address ← if instr.reverse
6              then instr.address + instr.count − 1 − offset
7              else instr.address + offset
8
9          cmd.size ← if offset + burstsPerCommand < instr.count
10             then burstsPerChunk
11             else instr.count − offset
12
13         cmd.inc ← if instr.reverse
14             then −1
15             else 1
16
17         cmd.sendEcho ← 0
18         cmd.stream ← 0
19         cmd.tag ← 0
20
21         offset ← offset + burstsPerCommand
22     end
23 end
```

Reading sequence data in reverse is realized by setting *cmd.inc* to -1. This makes it so that bursts are read from high to a low address. Data within bursts will still be in forward order. Reordering this data is done in the reorder kernel, which is explained in section 5.4.

While designing the memory sub-system, negative increments were not yet supported by MaxCompiler. As part of this work, we changed the MCP to accept increments between -128 and 127, rather than between 1 and 255.

5.2.3 Write Address Generator Kernel

The write address generator kernel is very similar to the read address generator kernel, but with some key differences.

Control

Figure 5.10b shows the write address generator kernel FSM. Like the read address generator, the write address generator fetches a memory instruction, but never has to wait for another write to complete. Subsequently, commands are generated, the kernel is flushed, and the process starts over again.

Generating MCP Commands

Write command generation pseudo-code is shown in listing 5.10. The difference to the read command generation is in the last couple of lines. `sendEcho` is set high in the last iteration if `sync` is enabled, to make only this last write command generate an echo. Similarly, `tag` is set high in the last iteration if `tagEnable` is set high, to instruct the MCP to interrupt CPU when the last command has finished processing.

Listing 5.10: Generating MCP write commands

```
1  function processInstruction(instr)
2      offset ← 0
3
4      while offset < instr.count do
5          cmd.address ← if instr.reverse
6              then instr.address + instr.count − 1 − offset
7              else instr.address + offset
8
9          cmd.size ← if offset + burstsPerCommand < instr.count
10             then burstsPerChunk
11             else instr.count − offset
12
13         cmd.inc ← if instr.reverse
14             then −1
15             else 1
16
17         isLastCmd ← (offset + burstsPerCommand ≥ instr.count)
18
19         cmd.sendEcho ← instr.sync ∧ isLastCmd
20         cmd.stream ← 0
21         cmd.tag ← instr.tagEnable ∧ isLastCmd
22
23         offset ← offset + burstsPerCommand
24     end
25 end
```

5.3 DRAM Write Kernel

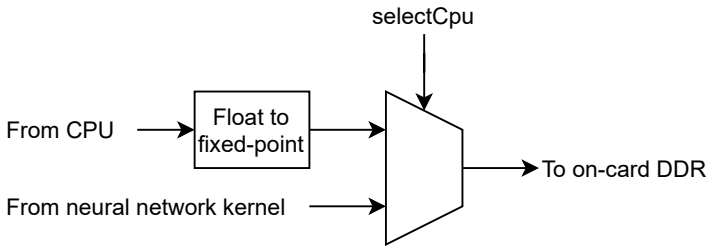


Figure 5.11: DRAM Write kernel design

Figure 5.11 shows the design of the DRAM write kernel. The DRAM write kernel writes either data from the neural network kernel, or data from the CPU to DRAM, selected by the control signal `selectCpu`, which is set by the CPU. Addresses for these write actions are generated by the write address generator kernel.

The CPU can stream weights and biases for the different neural network layers in single precision floating point numbers. The write kernel converts these to the internally used fixed-point format, and forward the data to DRAM.

5.4 Reorder Kernel

As discussed in section 5.1.4, GRU layers require sequence data to be read in reverse order. The input to a GRU layer is a stream of vectors of 112 elements, each of which is 4 bytes in size. This makes for a total of 448 bytes per vector. Since DRAM bursts have a size of 64 bytes, a vector fits in exactly 7 bursts. The layout of this sequence data in DRAM is shown in figure 5.12.

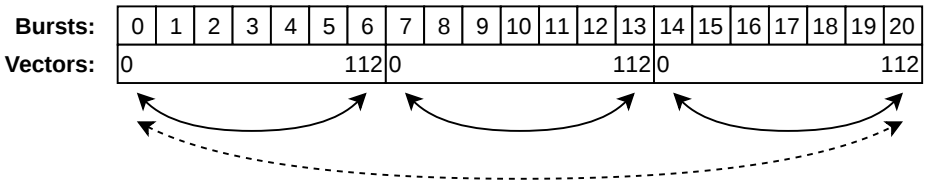


Figure 5.12: GRU sequence data memory layout with a sequence length of 3

When bursts are read in reverse order, the order of the data within vectors is also reversed, which is not what we want. To solve this problem, vectors are buffered in the reorder kernel, and reversed again to obtain the correctly ordered vectors. This is visualized in figure 5.12, where the large double-sided arrow indicates reversing the sequence of bursts that constitute a sequence, and the small double-sided arrows indicate reversing the vectors.

The reversal of vectors happens in the reordering kernel by buffering vectors in on-chip memory and forwarding them in reverse to the neural network kernel where the GRU layers are implemented. In the following sub-sections we will first see how on-chip memory is utilized to reorder vectors. Subsequently, we will look at the control logic of the reorder kernel.

5.4.1 Buffering & Reordering

Figure 5.13 shows the design of the buffering and reordering logic. The Burst memory block is an on-chip memory where vectors are double-buffered. Double buffering is implemented by alternating between:

- Writing to odd-numbered addresses and reading from even-numbered addresses;
- Writing to even-numbered addresses and reading from odd-numbered addresses.

The `Remove padding` block removes padding from individual vector elements. In the current implementation, elements are of a 27 bit fixed-point type, which is padded to 32 bits. The `reverse` control signal can be set low to disable reversing vectors, in case sequence data is read in forward order from DRAM. `inputEnable` and `outputEnable` can be set to fill/flush the burst memory at appropriate times.

The actual reversal of vector elements is implemented by generating increasing write addresses, and decreasing read addresses.

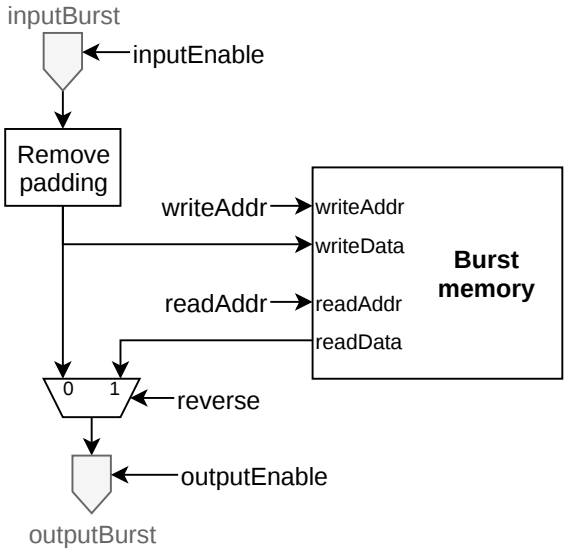


Figure 5.13: Reorder kernel design

Control

Figure 5.14 shows the state machine that controls the reorder kernel. First, a memory instruction is fetched. These are the same instructions that are received by the address generator kernels. In the decoding state, the reorder kernel remembers the amount of data that should be read, and goes to the `Forwards` state, or to the `Backwards fill` state based on the `reverse` bit in the memory instruction. The forwards state is straightforward: The `reverse` signal is set low and the `inputEnable` and `outputEnable` signals are set high. This allows data to stream directly from the input to the output.

In the `Backwards fill` state, only the `inputEnable` is set high to fill the burst memory with one vector. Then we transition to the `Backwards stream` state. In this state data is both read and written to/from the burst memory. When all but the last vector of data are processed, we transition to the `Backwards flush` state, in which only `outputEnable` is set high to flush the last vector from the burst memory without writing new data to it.

In any case (forwards or backwards) the last state before looping back to `Fetch` is the `Flush kernel` state, which flushes all data out of the kernel.

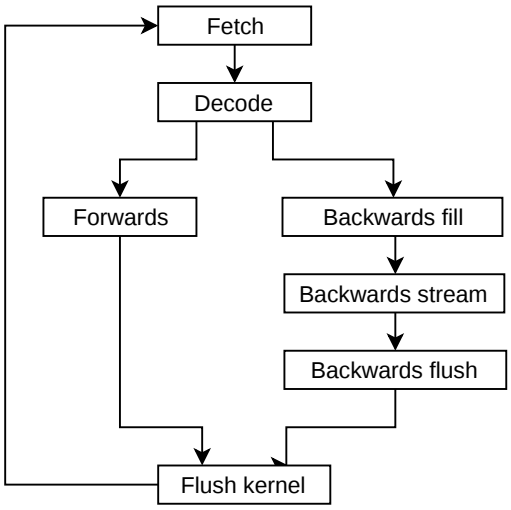


Figure 5.14: Reorder kernel FSM

5.5 Host Application

To fill up the chip, three instances of the complete design are instantiated (for reasoning, see chapter 4) which can process sequences independently. Each instance takes 5 interleaved sequences, which means that batches of 15 sequences are processed at a time.

The complete process of basecalling an arbitrary amount of raw sequences contained in files is described in listing 5.11. In the following sub-sections we will explain the additional software that was developed to facilitate DFE acceleration.

Listing 5.11: Complete basecalling program

```
1  function basecall(sequenceFileNames)
2      writeWeightsAndBiasesToDRAM()
3      sortedSequenceInfo = sortSequenceFileNamesByLength(sequenceFileNames)
4
5      while there are sequences left to process do
6          dfeInputBatch = readBatch()
7          dfeOutputBatch = dfeRun(dfeInputBatch)
8          finalSequenceData = decodeCRF(dfeOutputBatch)
9          writeToFiles(finalSequenceData)
10     end
11 end
12
13 function dfeRun(inputBatch)
14     initializeMemoryPrograms(inputBatch)
15     setInputStream(inputBatch)
16     setOutputStream(outputBatch)
17     run()
18     return outputBatch
19 end
```

Most of the software of the above process already existed as part of Scrappie. Software developed to facilitate DFE acceleration include: Reading sequence data, interleaving, and un-interleaving. Each of these steps will be described in the following sections.

5.5.1 Reading Sequence Data, Interleaving & De-interleaving

Before reading raw sequences, they are first sorted by length. Batches of sequences are then created with similar-sized raw sequences. This is necessary because, as explained in section 5.1.4, when sequences are interleaved, padding is inserted at the end of the shorter sequences. The greater the difference between the sequence length, the more padding is inserted. The more padding is inserted, the more time is wasted by the DFE.

The batches of raw sequences are interleaved, and streamed to the DFE. As the DFE processes that batch of raw sequences, data is streamed back to the CPU. This data is un-interleaved for further processing.

5.5.2 Initializing Memory Programs

Recall from section 5.2.2 that the CPU provides the DFE with a stream of memory operations in order for the address generators to generate the correct MCP commands. Listing 5.12 shows how these instructions are generated.

Listing 5.12: Initializing memory programs

```
1  function initializeMemoryPrograms(inputBatch)
2      // convolution
3      read convolution weights and biases
4      write result, synchronize with GRU layer 0
```

```
5
6       // GRU layers
7       for i = 0...5 do
8           read GRU layer i weights
9           read GRU layer i input in reverse, synchronize with last layer
10          write GRU layer i result, synchronize with next layer
11      end
12
13      // feed-forward
14      read feed-forward weights and biases
15      read feed-forward input in reverse, synchronize with last layer
16  end
```

The convolutional layer does not have a read operation, since it takes data from PCIe. Every layer synchronizes its writes with the next layer's read operation. In addition, every layer has a separate read operation to load weights and biases before it starts processing. The last layer (feed-forward) does not have a write operation, since it will send its output data back to the CPU through PCIe.

6 Experimental Results

In this chapter predicted performance and area utilization are compared to the measured performance and area utilization of the final implementation. In addition, performance of the system is compared to server-grade CPUs with several different workloads.

In addition, power usage of power efficiency of the accelerated implementation and original CPU-based implementation is compared.

6.1 Performance & Area

In the following subsections, the time taken by the accelerated layers (as opposed to total run-time) is hard to measure in a multi-threaded program. Therefore, it is approximated by assuming constant speedup S for each layer. S is calculated by measuring the total run-time in multi-threaded mode $T_{mt,tot}$, and the total run-time in single-threaded mode $T_{st,tot}$. We then determine that $S = \frac{T_{mt,tot}}{T_{st,tot}}$. In single-threaded mode, the time per layer $T_{st,layer}$ can be measured. The time taken by this layer in multi-threaded mode can then be approximated: $T_{mt,layer} = \frac{T_{st,layer}}{S}$.

6.1.1 Comparison with the Performance Model

In order to asses the accuracy of our predicted performance and area utilization, the same workload used in chapter 3 is run again on the final system. Table 6.1 shows the predicted area usage and run-time of the system, derived from the profiling run presented in chapter 3 and the performance model (chapter 4, table 4.12). Note that the area-percentages in the "Predicted" column include the 20% overhead of resources for compiler generated logic mentioned in the performance model.

The third column shows the actual area utilization, as reported by MaxCompiler, and run-times measured using the same workload. T_{accel} is time spent in the layer that are accelerated, whereas T_{tot} is the total runtime, including software layers. The fourth column shows the percentage of the values of the "Actual" column, compared to the "Predicted" column.

The fourth column is the percentage of the actual utilization/run-time vs the predicted utilization/run-time, and serves as a metric of accuracy of the predictions.

In terms of area utilization, the amount of BRAMs and URAMs are used a lot more than predicted. This is thought to be mostly due to long and wide pipelines in the design.

	Predicted	Actual	% of prediction
LUT%	30.26 %	22.24 %	73.50 %
FF%	35.71 %	33.41 %	93.55 %
DSP%	66.32 %	61.32 %	92.47 %
BRAM%	30.07 %	50.63 %	168.38 %
URAM%	20.00 %	35.63 %	178.15 %
T_{accel}	19.25 s	19.44 s	100.99 %
T_{tot}	24.06 s	28.04 s	116.52 %

Table 6.1: Predicted metrics vs. actual metrics

If we were to scale up the amount of instances of the design on the chip, this would certainly require further investigation.

In terms of runtime, the accelerated run-time is predicted to within 1% accuracy. The total run-time is slightly longer than expected. This is most likely due to the fact that the performance model does not take into account the time needed to interleave sequences before they are streamed to the DFE. In addition, there is an un-interleaving step after data is streamed out of the DFE.

6.1.2 Comparison with Server CPUs

The final system is compared against a server with two *Intel Xeon Processor E5-2643 v4* CPUs. Each package contains 6 cores each with hyper-threading. This makes for a total of 16 physical cores, i.e.: 24 logical cores, each running at a peak frequency of 3.4 GHz. The final DFE based system is compared to the CPU implementation in terms of performance and accuracy.

Accuracy is determined by comparing the basecall produced by the original CPU implementation, and the basecall produced by the accelerated version. The two DNA sequences are aligned using the NCBI Basic Local Alignment Tool (BLAST) [21]. BLAST will find the best alignment between two sequences, and determines a score for this alignment. By default, 1 point is added to the score if two bases match, whereas a penalty of -2 is added if bases do not match.

Figure 6.1 shows an example of two aligned sequences. In this example the query sequence has 32 matches, 1 mismatch and a gap of size 2. The gap is scored as if it were 2 mismatches, giving this alignment a total score of 26 out of a maximum of 35 (the subject sequence length).

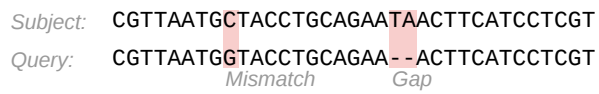


Figure 6.1: Example BLAST alignment of two sequences
(1 mismatch, 2 gaps, and 32 matches)

Input size (millions)	Mean score	Score stdev	Wall-clock time CPU	Wall-clock time DFE	Wall-clock time Speedup	Accelerated part time CPU	Accelerated part time DFE	Accelerated part time Speedup	kbps
21.5	0.9986	0.00150	47.53	36.15	1.31	38.02	20.21	1.88	66.34
50.9	0.9911	0.08150	115.07	79.64	1.44	92.26	47.91	1.92	59.39
81.0	0.9947	0.05756	181.12	125.01	1.44	144.90	76.12	1.90	59.37
111.0	0.9978	0.00219	251.43	170.56	1.47	201.14	104.30	1.93	57.50
141.0	0.9970	0.01386	315.05	217.11	1.45	252.04	132.53	1.90	57.49
171.2	0.9975	0.00277	375.22	262.69	1.42	300.18	160.86	1.86	56.22
201.2	0.9969	0.01335	438.5	305.36	1.43	350.80	188.99	1.86	55.30
230.9	0.9975	0.00239	513.65	348.54	1.47	410.92	216.92	1.89	54.65
262.1	0.9975	0.00213	593.9	395.43	1.50	475.12	246.33	1.93	53.25
295.7	0.9958	0.02369	675.64	445.18	1.51	540.51	278.43	1.94	50.87

Table 6.2: Benchmarking results table

In our comparisons, scores are normalized, i.e.: divided by the maximum sequence length. This gives us a normalized score between zero and one.

Table 6.2 shows CPU and DFE runtimes of the whole program, and just the accelerated part for several different input sizes. In addition, the speedup of both parts is listed. Each run is performed with 300 input sequences of different sizes. The mean and standard deviation of the scores of the sequences is given in column 2 and 3 respectively.

The mean accuracy is nearly perfect, with low standard deviations for all tested input sizes. The reason the system does not produce identical results, is because the CPU implementation implements all arithmetic using IEEE floating point numbers, whereas the DFE implementation does so mostly in fixed point.

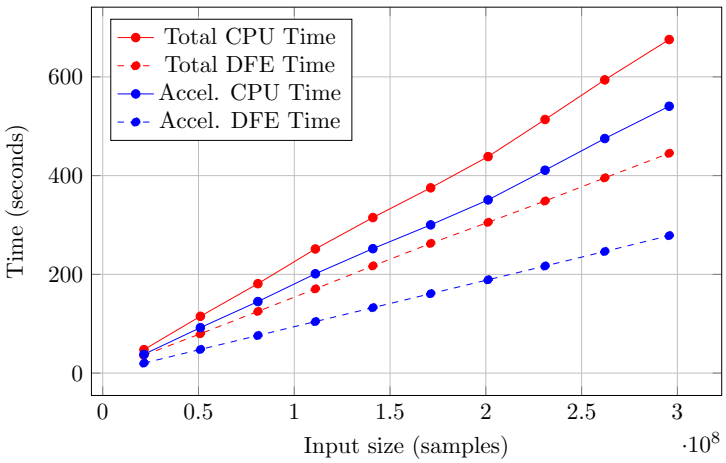


Figure 6.2: Benchmarking results graph

Figure 6.2 visualizes the results from table 6.2. Total run-time as well as time taken by the accelerated part increases approximately linearly with input size.

6.1.3 Comparison to Related Work

The speedup of the accelerated part of the network is 1.9x. This is slightly lower than the the related work by Li et al [13] (3.12x), and Zhang et al [15] (3.1x). However, a speedup of 3.8x is thought to be possible (see section 7.2) when some further optimizations are made to this work.

The mean output rate of the accelerated system is 57.04 kbps (kilo bases per second). This is faster compared to other opens-source basecallers. e.g., Chiron (2.7 kbps), and Flappie (14 kbps). Guppy outperforms this work with an output rate of 1.5 Mbps, but will most likely consume a lot more energy, since it uses high-performance GPUs to accelerate computation.

6.2 Power Efficiency

To make a fair comparison in terms of energy efficiency, we introduce the metric Energy per Sample (EPS), given in equation 6.1. Power was measured on a machine with an *Intel Xeon W3565* and a MAX5C, running both the original CPU based implementation, and the DFE implementation. Since the Xeon W3565 is an older CPU (from 2009), power was also measured on a workstation with a newer CPU (*Intel Core i7-7800X*, from 2017). Power measurements over a period of two minutes are presented in figure 6.3.

$$EPS = \frac{Power \times Time}{\#samples} \tag{6.1}$$

Power usage of the DFE based implementation fluctuates significantly more than the CPU based implementation. This is thought to be because of the fact that more work is done on the DFE than on the CPU. This causes the CPU to have to wait for the DFE at times, at which point the power usage drops briefly. [1]

Using these measurements we can calculate the EPS for each run, and make a comparison in terms of energy efficiency. This is summarized in table 6.3. We can conclude that the DFE based implementation uses only 9.66% of the energy per sample as compare to the CPU based implementation on the same machine. Compared to the i7-7800X, the DFE implementation uses 50.44%.

[1] Depending on the amount of idle time available to the CPU, this means you could service multiple DFEs with a single CPU.

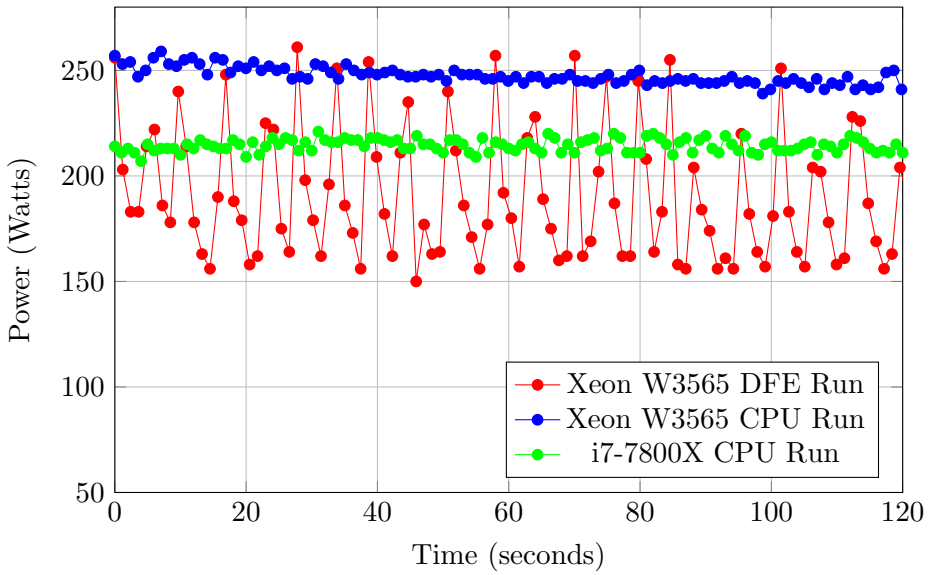


Figure 6.3: DFE Power usage graph

	CPU+DFE (Xeon W3565)	CPU (Xeon W3565)	CPU (i7-7800X)
Average power usage	190.8 W	247.5 W	214.4
Running time	358.6 s	2,863.15 s	77.15 s
No. input samples	169.9e6	169.9e6	20.7e6
EPS	0.4028 mJ	4.1711 mJ	0.7985 mJ

Table 6.3: EPS per implementation

7 Conclusion & Future Work

In this thesis a neural-network based ONP basecaller was accelerated using dataflow programming. To do this, a design has been rigorously modeled, implemented, and tested. The final system provides roughly 1.5 times speedup over a CPU based two-socket server using a single FPGA accelerator card, providing nearly perfect accuracy when compared to the results of the original CPU based implementation. The design presented here is predominantly bound by the amount of available DSPs. We have clear indications that reducing the number of DSPs, and/or using a chip with more DSP resources in favour of generic reconfigurable logic will improve performance. In addition, our final system uses up to 90.27% less energy compared to the original implementation.

7.1 Answers to the Research Questions

Which existing basecalling algorithm is best suited for FPGA acceleration?
According to our research ONPs Scrappie was most suited to implementation on an FPGA. This was decided because of the following reasons:

1. Scrappie is open-source;
2. Scrappie had promising speed and accuracy to begin with;
3. It is easily readable self-contained source-code;
4. The neural network topology is suitable for FPGA implementation.

Can FPGAs provide an interesting alternative for accelerating basecalling/neural-networks in terms of throughput and power consumption as compared to CPUs and GPUs?
The accelerated system has an output rate of 57.04 kbps, which is 1.5x faster than a CPU implementation of the same network. The fastest GPU based basecaller produces 1.5 Mbps. However, this was produced by Oxford Nanopore using a closed-source implementation that is expected to be drastically different from Scrappie.

In any case, the reason that GPUs can be so fast here is that many neural-network layers are implemented using big matrix multiplications. This is something GPUs excel at. It should be noted however, that we can only speculate how Guppy is implemented.

In terms of power efficiency FPGAs could be an interesting alternative. Especially when power efficiency is an important design criterion. Regrettably, energy-efficiency numbers on Oxford Nanopore's Guppy are not publicly available.

Is there a way to achieve maximum DRAM read and write bandwidths in the context of an FPGA basecalling application?
Because data is accessed always in a linear (increasing or decreasing addresses) fashion, DRAM can efficiently be used by the FPGA to store intermediate results, without the need for an elaborate cache system. This is achieved with separately clocked memory address generators and sufficiently deep FIFOs to and from the DRAM.

7.2 Future Work

If we can reduce the width of the fixed-point variables, we will be able to use one DSP per multiplication instead of two DSPs as in the current design. If we also manage to reduce the amount of on-chip memory usage, the amount of instances may be increased to six, doubling the throughput of the GRU layers. Since the GRU layers take roughly 80% of the running time (see section 4.4), that would result in a 2.5x speedup of the accelerated part of the design. The accelerated part takes about 70% of the total running time in our benchmarks. This means that the overall speedup for the complete application would be a 2.05x.

The convolutional and feed-forward layers share only one matrix-vector multiplier with the GRU layers. Using a more sophisticated hardware sharing mechanism, matrix-vector multipliers can be shared and hence the throughput of these layers will increase. However, the feed-forward layer will then be bottle-necked by the PCIe output speed. To alleviate this, we may implement the first part of the CRF decoding layer, which reduces output bandwidth requirements by a factor of 25.

The accelerator card used in this work (the MAX5C) uses PCIe gen 3 by default. There is experimental support for PCIe gen 4, which delivers double data-rates. Using PCIe gen 4 would also remove the output bottleneck of the feed-forward layer, until six or more instances are instantiated.

Bibliography

[1] C. Lannoy, D. Ridder, and J. Risse, "The long reads ahead: De novo genome assembly using the minion," *F1000Research*, vol. 6, p. 1083, 12 2017.

[2] "Image showcasing the gradient descent behavior," https://upload.wikimedia.org/wikipedia/commons/7/79/Gradient_descent.png, accessed: 2019-10-08.

[3] R. R. Wick, L. M. Judd, and K. E. Holt, "Performance of neural network basecalling tools for oxford nanopore sequencing," *Genome Biology*, vol. 20, no. 1, p. 129, 2019.

[4] *UltraScale Architecture and Product Data Sheet*, Xilinx, 8 2018, v3.5.

[5] "Oxford nanopore technologies: About us," https://nanoporetech.com/about-us, accessed: 2019-10-12.

[6] "Scrappie basecaller (github)," https://github.com/nanoporetech/scrappie, accessed: 2019-10-12.

[7] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter, *Molecular Biology of the Cell*, 6th ed. The address: Garland Science Publishing, 2014.

[8] B. E. Slatko, A. F. Gardner, and F. M. Ausubel, "Overview of next-generation sequencing technologies," *Current Protocols in Molecular Biology*, vol. 122, no. 1, p. e59, 2018.

[9] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 10 2014, pp. 1724–1734. [Online]. Available: https://www.aclweb.org/anthology/D14-1179

[10] V. Boa, B. Brejov, and T. Vina, "Deepnano: Deep recurrent neural networks for base calling in minion nanopore reads," *PLOS ONE*, vol. 12, no. 6, pp. 1–13, 06 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0178751

[11] H. Teng, M. D. Cao, M. B. Hall, T. Duarte, S. Wang, and L. J. Coin, "Chiron: Translating nanopore raw signal directly into nucleotide sequence using deep learning," *bioRxiv*, 2017. [Online]. Available: https://www.biorxiv.org/content/early/2017/09/12/179531

[12] "Epi2me," https://nanoporetech.com/nanopore-sequencing-data-analysis, accessed: 2019-10-08.

[13] Q. Li, X. Zhang, J. Xiong, W.-m. Hwu, and D. Chen, "Implementing neural machine translation with bi-directional gru and attention mechanism on fpgas using hls," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ACM, 2019, pp. 693–698.

[14] "Vivado high-level synthesis," https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, accessed: 2019-10-08.

[15] X. Zhang, X. Liu, A. Ramachandran, C. Zhuge, S. Tang, P. Ouyang, Z. Cheng, K. Rupnow, and D. Chen, "High-performance video content recognition with long-term recurrent convolutional network for fpga," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–4.

[16] "Tensorflow," https://www.tensorflow.org/, accessed: 2019-10-08.

[17] "Flappie basecaller (github)," https://github.com/nanoporetech/flappie, accessed: 2019-12-10.

[18] H. T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, Jan. 1982.

[19] J. B. Dennis, "Data flow supercomputers," *Computer*, vol. 13, no. 11, pp. 48–56, Nov. 1980.

[20] N. Voss, "Methodology for complex dataflow applications," Ph.D. dissertation, Imperial College London, South Kensington, London SW7 2AZ, United Kingdom, 2020.

[21] "Blast: Basic local alignment search tool," https://blast.ncbi.nlm.nih.gov/Blast.cgi, accessed: 2019-10-08.