HARDWARE ACCELERATED DNA SEQUENCING

ZHONGPAN WU

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCES
YORK UNIVERSITY
TORONTO, ONTARIO

FEBRUARY 2019

Abstract

DNA sequencing technology is quickly evolving. The latest developments exploit nanopore sensing and microelectronics to realize real-time, hand-held devices. A critical limitation in these portable sequencing machines is the requirement of powerful data processing consoles, a need incompatible with portability and wide deployment. This thesis proposes a first step towards addressing this problem, the construction of specialized computing modules – hardware accelerators – that can execute the required computations in real-time, within a small footprint, and at a fraction of the power needed by conventional computers. Such a hardware accelerator, in FPGA form, is introduced and optimized specifically for the basecalling function of the DNA sequencing pipeline. Key basecalling computations are identified and ported to custom FPGA hardware. Remaining basecalling operations are maintained in a traditional CPU which maintains constant communications with its FPGA accelerator over the PCIe bus. Measured results demonstrated a 137X basecalling speed improvement over CPU-only methods while consuming 17X less power than a CPU-only method.

Keywords: DNA Sequencing, Nanopore, Real-time, Hand-held, Sequencing device, FPGA, Hardware Accelerator, PCIe, CPU, Basecalling speed, Power Consumption

Acknowledgements

First and foremost, I would like to express my deepest appreciation to my supervisor, Sebastion Magierowski. He guided me during my study in my master study as an old close friend. He inspired and encouraged me when I was disappointed and got lost in my research. He was always patient and warm-heart and taught me how to be more excellent and more efficient. I am very glad to have a good time with him in my study life. I hope I would become the kind of an enthusiast in the research like him in the future.

Second, I also would like to express my gratitude to my co-supervisor Prof. Ebrahim Ghafar-Zadeh. He is always willing to help his students not only in their professional research but also their future life and career. His keen and vigorous academic observation enlightens me not only in this thesis but also in my future study. Thanks for his encouragement and advises, he made my master life more fulfilling.

Of course, I am very grateful of my dear friends and colleges, Karim Hammad ,Yunus Dawji and Sumaia Atiwa. This accomplishment would not have been completed without out their help. Thank you all for always supportive.

Finally, I must express my sincere appreciation goes to my parents for their love toward me, because of their warm care I can grow up well.

Table of Contents

List of Tables

List of Figures

Abbreviations

FPGA	Field-Programmable Gate Array
DNA	Deoxyribonucleic Acid
HMM	Hidden Markov Model
NGS	Next-Generation Sequencing
HGP	Human Genome Project
SBL	Sequencing by Ligation
SBS	Sequencing by Synthesis
ONT	Oxford Nanopore Technologies
DC	direct current
ASICs	Application-Specific Integrated Circuits
RIFFA	Reusable Integration Framework for FPGA
PCIe	Peripheral Component Interface Express
CPU	Central Processing Unit
NNs	Neural Networks

ASR	Automatic Speech Recognition
RX	Receive
TX	Transmit
PFD	Process Flow Diagram
GPU	Graphics-Processing-Unit
COPS	Core-Operations-Per-Second
FIFO	First-In-First-Out
IP	Intellectual Property
CMOS	Complementary Metal-Oxide-Semiconductor
ASIC	Application Specific Integrated Circuit
MMCM	Mixed-Mode Clock Manager
PLL	Phase-Locked Loop
DLL	Delay-Locked Loop

1 Introduction

1.1 Background and Motivation

Deoxyribonucleic nucleic acid (DNA) is a biomolecule present in the cells of living organisms. It is a complex organic compound of which the chromosomes in the nucleus of an animal's cell mainly consist. As the "blueprint of life", DNA is famously understood as the carrier of its host creature's genetic information. The structure of DNA molecules consist of two polymer chains symmetrically coiled around one another to form the famous double-helix shape. Each chain is composed of four monomeric nucleotide molecules, each nucleotide distinguished by the *base* molecules contained within them: Adenine (`A`), Thymine (`T`), Cytosine (`C`), Guanine (`G`). These bases appear in some unique sequence in each chain of an organism's DNA. The coiled chains form links (pairs) with one another via their base constituents. The links occur in a "complementary" fashion (`A` links to its complement `T` and vice-versa while `C` links to its complement `G` and vice-versa) [1] as shown in Fig. 1.1. Two linked bases are referred to as a *base pair* (bp).

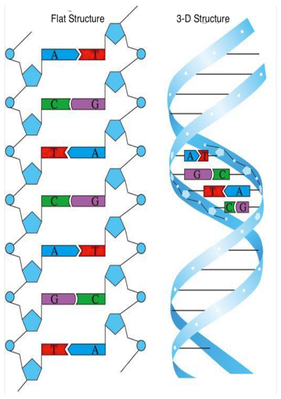


Figure 1.1: Double helix structure of the DNA.

Sequencing is the process of identifying the sequence of bases comprising a DNA molecule [2]. An efficient means of executing this process was forwarded by Fredrick Sanger over 40 years ago and has rapidly evolved since [3]. Today, high-end "next-generation sequencing" (NGS) machines such as the Illumina Hi-SeqX (pictured in 1.2) can sequence quantities of DNA equivalent to roughly 10 human genomes per hour [4, 5].

1.1.1 First Generation Sequencing

DNA sequencing machine evolution is mainly divided into three generations as shown in Fig. 1.3. As early as the 1980s, the first-generation sequencing technology, distinguished by its reliance on Maxam-Gilbert and Sanger dideoxy methods (of chain termination and chemical degradation), was widely used [8]. The Human Genome Project (HGP) heavily relied on this technology [9].

Figure 1.2: The Illumina Hi-SeqX mode ultra high throughput sequencing system [6].

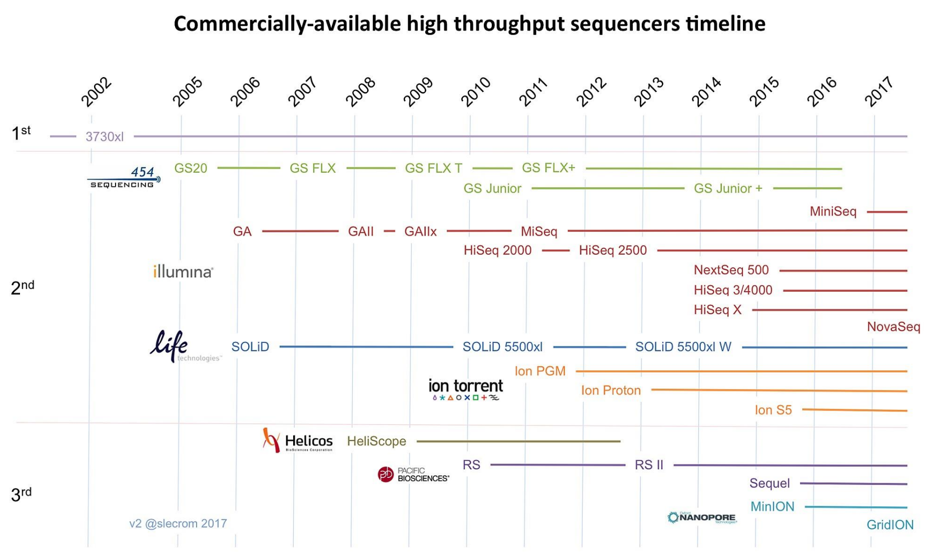


Figure 1.3: Currently available commercial sequencing technologies and corresponding manufacturers [7].

1.1.2 Second Generation Sequencing

With the arrival of second-generation sequencing technology (the aforementioned NGS) in 2005, tremendous gains in DNA sequencing rates were achieved. NGS came up with two novel sequencing approaches: sequencing by ligation (SBL) and sequencing by synthesis (SBS). Briefly, these utilize *DNA polymerase* to synthesize DNA one-base-at-a-time. By utilizing modified bases, each synthesis reaction emits a unique signal allowing a suitable sensor to then infer the identity of the base and thus implement a form of sequencing [10, 11, 12].

Although extremely accurate, the chemical challenges of this process only allow it to handle DNA samples 100 to 200 *base-pairs* (bp) long (so-called "short reads"). Thus, long DNA molecules must first be carefully fragmented into short samples before measurements with NGS technology. Prominent manufacturers of NGS machines include Illumina and Ion-Torrent (now Thermo-Fisher).

NGS has led to the realization of unprecedented sequencing throughputs and has driven a rapid acceleration in genomic information gathering. It is now the most mature and dominant sequencing solution on the market. However, it requires not only a large amount of chemical processing (e.g. PCR amplification) but also increased costs before sequencing. Meanwhile, its capability of only handling short reads makes the genome assembly more sophisticated.

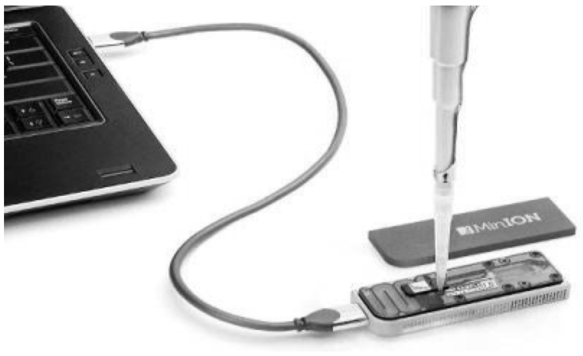


Figure 1.4: The first handheld sequencer developed by Oxford Nanopore [15].

1.1.3 Third Generation Sequencing

In 2014, ONT released a handheld sequencer, the MinION (Fig. 1.4). This is essentially a single-molecule DNA sequencing technology as it can directly sequence from native DNA without the need for chemical amplification or reconstruction as with NGS [13]. Its significant departure from existing methods has prompted the MinION's designation as a third generation sequencing technology [14].

The MinION measures DNA by deploying an electrochemical direct current (DC current) through a nanoscale hole (a sensor referred to as a *nanopore*) made of a protein or synthetic material immersed in an ionic bath [16]. When DNA also translocates through this hole it disrupts the ionic current in ways indicative of its structure [17].

ONT's nanopore technology holds several advantages over its NGS counterparts:

- Real-time sequencing (DNA can be physically introduced into the device and measured in a continuous, streaming, fashion).

- Ultra-long reads (average read lengths exceeding 2,000 bp are standard in the technology with some researchers developing protocols allowing in excess of 1-Mbp DNA strands to be read).
- Miniaturization (the nature of this sensing technology has allowed $+100\times$ size reduction by volume over the next smallest sequencer).

As part of its DNA sequencing process, the MinION (or rather the computer to which it feeds its data) "figures out" the most likely DNA sequence by inspecting this signature current [18]. This process is referred to as *basecalling* and addressing the computational challenges associated with it through custom hardware is the main focus of this thesis.

1.1.4 Nanopore Sequencing: Limitations and Motivation

Some caveats to the breakthroughs realized by the MinION technology are worth noting. The accuracy of the MinION sequencers are currently worse, significantly worse for some applications, than NGS technologies. Roughly, perhaps a 10%-20% error rate is present in the MinION relative to less than 1% in Illumina machines [19, 20, 21].

Also, no bioinformatic computing is done within the MinION itself, but rather an adjoining computer; arguably making the standalone MinION more of a DNA sensor rather than a DNA sequencer [22]. Nonetheless given the heavy reliance of this device on microelectronics and the historical trends exhibited by such technologies it is highly probable that the MinION's computing needs will eventually be closely integrated with the MinION itself. It is with this anticipated trajectory in mind that this thesis focuses on custom compute hardware implementations of critical bioinformatics functions required by the MinION and nanopore-based sequencing in general.

The importance of this effort is amplified by the small footprint of sequencers such as the MinION. The portability of this device has already started to have fascinating impacts on genomic field science, the ability to carry out analyses of DNA molecules outside the lab. Many other applications will likely be facilitated by the miniaturized molecule measurement opportunities opened up by nanopore-based sequencers such as the MinION.

But to truly realize the scales at which nanopore-based sequencing technology may be deployed, associated functions, such as computing, must scale in proportion; a small sensor attached to a large computer is inadequate for field genomics. The saturation of Moore's Law [23], and the associated slow-down in traditional microprocessor computing speed (per unit power) [24] can no longer be expected to provide sufficient computing capability at low-power (to maintain nanopore-based sequencer mobility) from commodity devices. This is certainly the case for the compute-intensive bioinformatic operations required of noisy nanopore sequencer signals. Rather, custom microelectronic computing hardware solutions are needed to assist the advancement of nanopore-based technologies towards the vision of "ubiquitous sequencing" [25].

1.2 Thesis Goals

Only the basecalling function of a nanopore-based DNA sequencing pipeline is considered in this thesis. As already alluded to above, basecalling is the process of converting noise current signals gathered by a nanopore sequencer such as the MinION to a text representation of its base sequence, that is, a sequence of letters drawn from the alphabet `A`, `C`, `G`, `T`. This is the first intense computation faced by the nanopore sequencing pipeline and will likely have to be solved natively (within the sequencing device) rather than remotely (i.e. in the cloud) as the cost and latency of moving high-fidelity raw measurement current signals to remote computing centres at large scales does not seem imminently feasible.

A semi-custom hardware computing implementation of the basecaller is the target in the form of a field-programmable-gate-array (FPGA). FPGAs are chip containing thousands of copies of standard digital logic gates (AND, OR, etc.), digital signal processing blocks, and memory units. In contrast to fully-custom chips (namely: application specific integrated circuits – ASICs) where the designer is free to construct any component feasible within the semiconductor substrate, FPGA designers can only reconfigure the connections between the aforementioned pre-made computing blocks on their FPGA chip to realize their intended hardware. Nonetheless, a great deal of flexibility remains to parallelize operations and achieve efficient computations in FPGAs [26].

Relative to an ASIC, the FPGA suffers in achievable speed and power efficiency, however it offers tremendous savings in up-front costs (no need to fabricate a chip from scratch) and can be reconfigured and updated after initial roll-out. Relative to a traditional CPU microprocessor (incommensurate semiconductor technologies) the FPGA may still offer orders of magnitude improvement in speed and power consumption if well designed [27].

The goal of this thesis is thus to realize an FPGA-enabled basecaller capable of demonstrating orders-of-magnitude improvements in speed and power efficiency over CPU-only equivalents [28]. For maximum flexibility an *hardware accelerated* basecaller realization in targeted wherein particularly intense basecalling sub-operations are delegated to an FPGA working in tandem with a traditional CPU. Thus, the flexible computing and data processing functions of the CPU can be leveraged for complex controls and data management tasks, while intense streaming operations are handled by the adjoining FPGA accelerator.

1.3 State-of-the Art and Thesis Outline

For deeper context we describe in more detail the operation of a state-of-the-art nanopore-based sequencer. In commercial units from ONT the nanopore sensor is a protein molecule in the shape of a hole positioned in a synthetic bilayer membrane [29]. An array of such sensors is usually realized, around 2,000 over a thumbnail area resting in close proximity to an analog/digital chip which detects sensor currents (essentially a noisy time-series signal), amplifies, filters, and digitizes them for the ensuing bioinformatics computational pipeline (basecalling being the first step). As already noted, this sensor array is immersed in an ionic fluid and a small baseline DC current (about 200 picoamperes) is made to flow through each pore.

Although the nanopore's diameter is small enough to accommodate only one DNA molecule at-a-time, the barrel of the sensor is long, housing roughly 10 DNA bases at-a-time. Thus, the ionic current flowing through the sensor is simultaneously interrupted by $k > 1$ bases at a time. Assuming these interrupting bases are contiguous we would say the interruptions are generated by some k-Mer.

The measured time-series signals produced by the nanopore sensor's interaction with DNA (see Fig. 1.5 for details), are logged and saved in data files that are then passed onto the basecaller.

The basecaller, as a tool for converting the measured time-series to its (molecu-

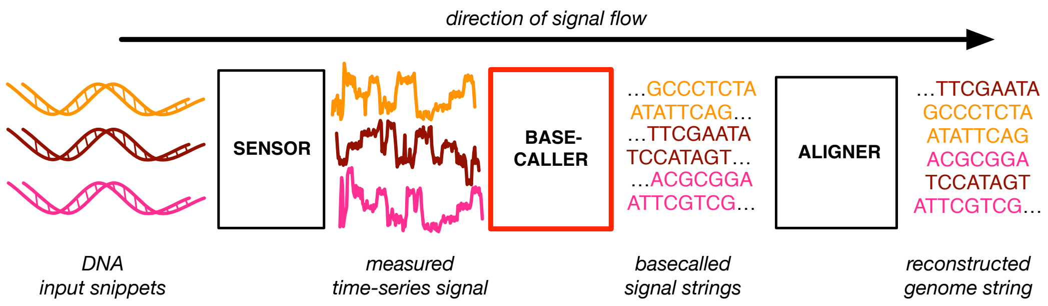


Figure 1.5: A simple DNA processing steps. DNA input snippets are sensed and converted to corresponding electronic time-series signals; these are basecalled into separate reads and finally assembled into a complete genome text string [30].

lar) text equivalent, plays a crucial role in the ultimate extraction of the complete sequences associated with the DNA being measured. ONT offers open-source and proprietary software, some of which even runs on proprietary FPGA-based hardware accelerators (in their enterprise-level machines). Although ONT has made announcements of portable FPGA-based basecalling acceleration methods, none have yet been released [31].

1.4 Contributions of the Thesis

The contributions of this thesis are list as follows.

1. The HMM-based Viterbi basecalling algorithm was implemented on MATLAB, and the its functionality validation was passed by processing the simulated event sequence.

2. A high-performance communication implementation which based on an open-sourced project was achieved and tested to satisfy the requirement of high-throughput sequencing.

3. A hardware solution of the basecalling accelerator on the FGPA was implemented. Its functionality and speed was verified.

4. A basecaller algorithm was completed on a system that constructs with CPU and FPGA involved.

5. The whole measurements which includes power consumption, maximum clock frequency and resource utilization were finished.

1.5 Thesis Outline

There are six chapters in the thesis, and the second chapter describes the basic concepts of HMM nanopore DNA sequencing and the associated Viterbi algorithm for basecalling. The third chapter describes the actual FPGA design. In the fourth chapter, we introduce the practical application principle of hardware communication via the Reusable Integration Framework for FPGA Accelerators (RIFFA) and the adaptation of these solutions for our basecaller. In chapter five, measurement results of the implemented basecaller are described. Chapter six summarizes the thesis, discusses the challenges and makes proposals for future work.

2 Basecalling Algorithm

2.1 Overview

Basecalling is a time-series to text-sequence conversion process [32]. Machine learning methods based on neural networks (NNs) and hidden Markov models (HMMs) are two well known ways of addressing this problem. This thesis focuses on HMM-based methods, a computationally milder approach than NNs [33, 34, 35], albeit with lower accuracy in certain nanopore-based sequencers. Still, HMMs have been shown to achieve satisfactory accuracy in cases where nanopores of sufficient quality are available [36]. In this chapter, we discuss an HMM of a nanopore-basecalling process and the Viterbi algorithm applied to this model to compute time-series to text conversion.

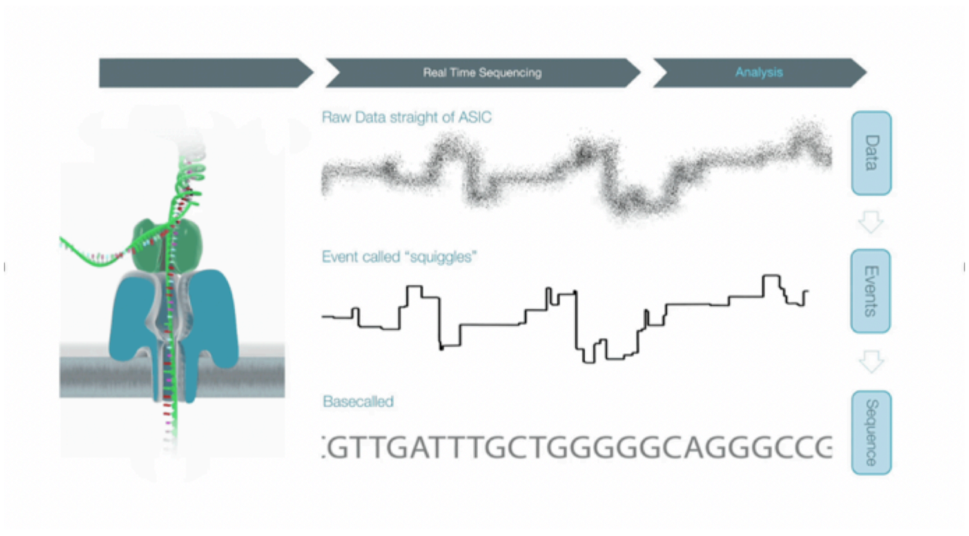


Figure 2.1: The raw signal from the nanopore [39].

2.2 Nanopore Signals

The discrete nature of DNA, as imposed by its discernible set of molecular building blocks (i.e. the four bases), tends to result in a nanopore output signal, the "raw" signal, approximating a step-wise time-series, albeit one with significant noise disturbances atop it [37]. An example of this signal is shown in Fig. 2.1. Before submitting such a signal to a basecaller it is typical to extract statistical features from this raw output indicative of this underlying step-wise pattern [38]. These features are typically referred to as *events*, which themselves form a time-series (an event-series) where each event consists of a three-element vector

$$e_i = [x_i, y_i, t_i] \tag{2.1}$$

where i is the event index in the sequence, x_i is the mean value of the raw signal samples comprising the event, y_i is the standard deviation value of these raw signal, and t_i stands for the relative time at which the event starts.

The basecaller's job is then to convert some sequence of n events, $(e_i)_{i=0}^{n-1}$, emerging from a nanopore sequencer into their underlying text equivalent, the basecall sequence drawn from the alphabet, `A`, `C`, `G`, `T` [40].

2.3 State Sequence Detection

Given that our idealized DNA consists of only four base-molecule building blocks it would be ideal if the event-series emerging from a nanopore sequencer – the input to the basecaller – consisted of only four unique levels. In this case it would be straightforward to implement the basecaller as some base-by-base statistical threshold detector that effectively looks at a single event at a time and determines which of four bases that singular event corresponds to.

As already mentioned in the previous chapter, the realistic scenario is not so simple. The current, at the very least, depends on a k-Mer sequence of DNA bases. That is, at any i, the value e_i is a reflection of k bases, not one. If k is say 3, then we would expect there to be $m = 4^3 = 64$ different levels that any e_i may take. Of course due to noise, in the raw signal from which the e_i's are derived, no event will exactly assume any of these assumed 64 levels. We must then resort to statistical methods of *sequence detection* (in contrast to the base-by-base approach mentioned above) to associate an event measurement with an expected level. An elaboration on these concepts follows.

To allow for the construction of a more formal model we map the constituent bases A, C, G, T to the number key 0, 1, 2, 3, respectively. This base-4 number system allows us to replace any k-Mer with a numerical label that we refer to as a state number. For example, the 3-Mer CGT denotes the state $1 \cdot 4^2 + 2 \cdot 4^1 + 3 \cdot 4^0 = 27$. Thus, in a 3-Mer model, our event current is indicative of the states 0 to 63 (in increments of 1). As a DNA sequence progresses through the pore so does the state sequence. That is, choosing the convention that a k-Mer label read right-to-left denotes DNA progression through the pore, then after our CGT state we expect some GTγ state to excite the next event where $\gamma \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$.

Conceptually, this does not change the basecalling situation from the simple 4-level case at all. Instead of detecting four different levels, now 4^k levels must be detected. Just a more finely tuned version of the statistical threshold detector may be considered. With each identity a new state is identified (or *statecalled*) and from this it is trivial to identify the base sequence. For example, referring to our example, if the state 27 is identified by the detector we convert to its base equivalent and basecall the left most letter, C in this example.

Practically however the situation is dire. Just because more bases are encoded within each fluctuation of the current (i.e. a k-Mer rather than 1 base) does not mean that the physical signal (the peak-to-peak current span say) is bigger. The signal "strength" is the same, but now the detector is forced to identify a factor of $4^k/4 = 4^{k-1}$ more unique signals within it. Since the noise fluctuations do not depend on the k-Mer to first order, effectively the signal-to-noise ratio has dropped by 4^{k-1} relative to the ideal case. The implication is that the accuracy of our basecalls will get worse.

Luckily the nature of this signal opens the door for more complex detection and hence better accuracy. Rather than detecting one event at a time, we can take advantage of the fact that consecutive states overlap. That is, as noted above, one state ($27 \equiv$ `CGT`) is related (overlaps) with the ensuing state (say $44 \equiv$ `GTA`). *Sequence detection* methods (in contrast to *symbol detection*) do exactly this. They observe an extended run of events, then based on their internal model of the possible state relations from one index i to the next $i+1$ they make a more educated guess of the underlying state (base) sequence and hence achieve more accurate basecalling.

2.4 HMM Model

An HMM is an internal model of the possible state relations from one index i to the next index $i + 1$ [41]. A pictorial depiction of this model, "unwrapped" over time (i.e. over the succession of observations $\ldots, i-1, i, i+1, \ldots$) – also referred to as a *trellis* model – is shown in Fig. 2.2.

The ith column of circles denotes the possible states, 0 to 4^k-1, that might be measured by the nanopore at some time i. The straight lines, the *transition* branches, emerging from the right side of each state circle denote the possible (probable) *state transitions* (weighted by the probability terms τ). These transition branches indicate how the model may evolve over time, that is which states in column i may transition to states in column $i+1$ as time increments. For example, we showed above that state 27 at i could transition to state 44 at $i+1$, but we can also show that state 27 cannot transition to state 5 at $i+1$ (i.e. there would not be a transition line joining these two).

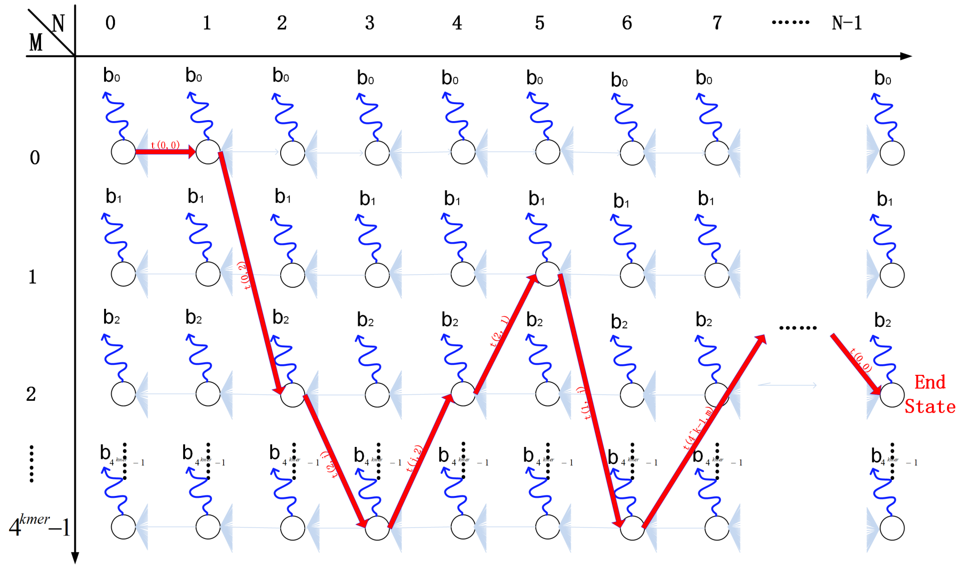


Figure 2.2: HMM with the optimal path traversing the trellis as found by the Viterbi algorithm.

Formally, the wavy lines emerging from each state denote the probable *state emission* from each state. In other words, these lines, weighted by the terms b_j, denote the probability that the state at i (in the HMM trellis) actually corresponds to the (noisy) event observed at state i.

Together, this model – its states, transition (probabilities), and emission (probabilities) – forms the means by which a formal detection algorithm can seek to identify the most likely state sequence based on a sequence of observations. The essential result sought by such an algorithm is embodied by the thick red path illustrated in Fig. 2.2. This path denotes the sequence of states in the trellis that the detection algorithm thinks are most likely behind the sequence of events observed. This estimate is based on the emission and transition probabilities the algorithm computes by considering the measured events and HMM parameters. The algorithm that achieves this, and the main object for hardware acceleration in this thesis, is the Viterbi algorithm and it is discussed next in the context of basecalling.

2.5 Viterbi Algorithm

The Viterbi algorithm (VA) is a dynamic programming algorithm that finds the most probable sequence of states (the most probable path) in the HMM based on its consideration over a sequence of event observations [42].

.

Algorithm 1 Optimized basecaller algorithm pseudo-code

```
1: for $j = 0 \rightarrow m-1$ do                    ▷ initializing the initial posterior and pointers
2:     $v_j(e_0) = b_j(e_0)$
3: end for
4: for $i = 1 \rightarrow n-1$ do                    ▷ updating the posterior and record the pointers
5:     for $j = 0 \rightarrow m-1$ do
6:         for $\nu = \{\omega_j\}_0^{20}$ do
7:             $v_j(i) = b_j(e_i) \max_{\nu \in \boldsymbol{\omega}(j)} [v_\nu(i-1)\tau(\nu, j)]$
8:             $\mathrm{ptr}_j(i) = \arg\max_{\nu \in \boldsymbol{\omega}(j)} [v_\nu(i-1)\tau(\nu, j)]$
9:         end for
10:    end for
11: end for
12: $\zeta_{n-1} = \arg\max_j(v_j(n-1))$                ▷ finding the end state
13: for $i = n-1 \rightarrow 1$ do                     ▷ traceback
14:     $\zeta_{i-1} = \mathrm{ptr}_(\zeta_i)$
15: end for
```

The algorithm's operation is summarized with the pseudocode in Algorithm 2.5. We elaborate on it below, for now we draw the reader's attention to key properties. The methodical nature of the VA is apparent from the three nested for-loops iterating over the heart of this algorithm. These for-loops iterate over:

- **Events**: $i = 1 \rightarrow n-1$. The total number of events in a measurement. In practice this approximately corresponds to the number of bases in a DNA strand that passed through a nanopore. This varies and depends on, among other things, the manner in which the DNA to be measured is experimentally prepared, but $n \approx 1{,}000$ is a representative approximation.

- **States**: $j = 1 \rightarrow m-1$. The total number of states in the model. For a sensor modelled with k-Mer response of $k = 3$, $m = 64$.

- **Transitions**: $\nu = \{\omega_j\}_0^{20}$. The state numbers (from the set $\{\omega_j\}_0^{20}$) of states that may transition into any particular state j. In the model considered here, 21 such transitions are possible (hence the set of possible state numbers for any state j is indexed from 0 to 20).

The core of the algorithm is to repeatedly calculate:

$$v_j(i) = b_j(e_i) \max_{\nu \in \boldsymbol{\omega}(j)} [v_\nu(i-1)\tau(\nu, j)] \tag{2.2}$$

$$\mathrm{ptr}_j(i) = \operatorname*{arg\,max}_{\nu \in \boldsymbol{\omega}(j)} [v_\nu(i-1)\tau(\nu, j)] \tag{2.3}$$

From a high-level, (2.2) calculates the *posterior* (probability), $v_j(i)$, that at time i the observation e_i corresponds to state j given that the most likely preceding state j at time $i-1$ is accounted for. The product of probabilities (i.e. the product of the emission probability, $b_i(e_i)$, with the most likely (max$[\cdot]$ term in (2.2)) preceding probability (the product of the preceding state's posterior, $v_\nu(i-1)$, and its associated transition probability, $\tau(\nu, j)$, into state j) achieves an accumulation of state probabilities from $i-1$ to i. As noted above, this operation is carried out iteratively for all states m. As a result of this iteration, the posterior $v_j(i)$ is an accumulated measure that accounts for all the (most likely) states preceding it (i.e. from $i = 0$ on).

At the same time (2.3) extracts the state number, ν, of the state at $i-1$ which most likely transitioned into the state with state number j at i. This extracted state number is associated with the *pointer* $\beta_j(i)$.

After the VA's iterative procedure completes the terminal state of the most likely path is identified by choosing the state j corresponding to the maximum value of the sequential probability. This is calculated in Algorithm 2.5 by

$$\zeta_{n-1} = \arg\max_j [v_j(n-1)]. \tag{2.4}$$

Finally, the rest of the states making up the most likely sequence are recovered by applying a traceback procedure (lines 13 to 15 in Algorithm 2.5) relying on the pointers accumulated during the iteration phase. This last loop accomplishes the aforementioned statecalling from which the final basecalling is trivially obtained.

2.6 Viterbi Algorithm: Component Details

In the following sub-sections we examine the details of the VA's component pieces. A summary of the key VA terms is provided in Table 2.1.

2.6.1 Emission Probability

The emission probability depicts a correlation between an observed event signal and the modelled event features associated with a state. In particular, the emission probability $b_j(e_i)$ reports the probability that observed event e_i is associated with

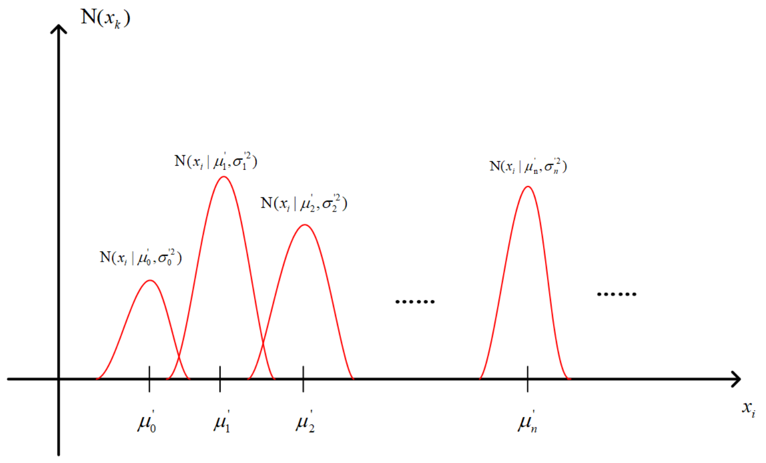
Table 2.1: Symbol Table

Parameters	Definition
i	index number of the event
j	index number of the state
n	number of events in the sequence
m	number of the states that related to the Mer size
Mer	number of DNA bases in the nanopore at one moment
e_i	i-th event in the sequence
$b_j(e_i)$	emission probability function in basecalling algorithm
ν	state number of neighbor to next state
$\tau(\nu, j)$	transition probability function
$\tau_j(i)$	sequential posterior probability
$v_j(i)$	accumulated posterior probability
$\text{ptr}_j(i)$	pointer that indicates the minimal path

state j. In our model we express the emission probability with (2.5)

$$b_j(e_i) = b_j\,(x_i, y_i) = \mathcal{N}(x_i|\mu_j, \sigma_j) \cdot \mathrm{IG}(y_i|\eta_j, \lambda_j) \tag{2.5}$$

where $\mathcal{N}$ denotes the Gaussian distribution and IG denotes the inverse-Gaussian distribution.



$N(x_k)$
$N(x_i \mid \mu_0', \sigma_0'^2)$
$N(x_i \mid \mu_1', \sigma_1'^2)$
$N(x_i \mid \mu_2', \sigma_2'^2)$
$N(x_i \mid \mu_n', \sigma_n'^2)$
......
......
μ_0'
μ_1'
μ_2'
μ_n'
x_i

Figure 2.3: Multivariate Gaussian emission probabilities for x_i.

The Gaussian models the likelihood that an event's mean-feature, x_i, is associated with state j. It is given by the expression

$$\mathcal{N}(x_i|\mu_j, \sigma_j) = \frac{1}{\sqrt{2\pi\sigma_j}} \exp\left(- \frac{(x_i - \mu_j)^2}{2\sigma_j^2} \right)$$

where μ_j and σ_j are the mean and standard deviation Gaussian model parameters associated with each state. In essence, we can think of our this model as a series of Gaussian distribution centred around 4^k μ_j level parameters with some spread proportional to σ_j (see Fig. 2.3)

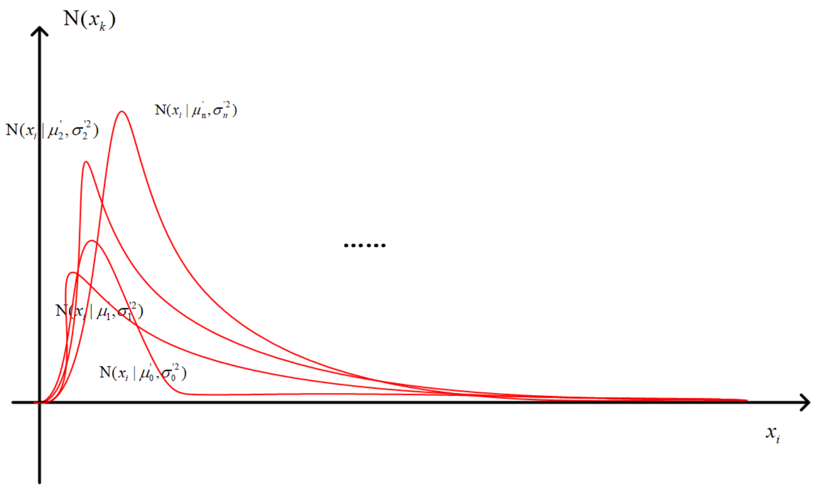


Figure 2.4: Multivariate Gaussian emission probabilites for x_i

Where the Gaussian associates the mean of our measurement with our model, the IG characterizes our measurement according to the nature of their variations within the pore. That is, it compares the standard deviation, y_i, of our event with another set of model parameters. Specifically, the IG expression is

$$\mathrm{IG}(y_j|\eta_j,\lambda_j) = \left[\frac{\lambda_j}{2\pi y_i^3}\right]^{\frac{1}{2}} \exp\left(\lambda_j \frac{(y_i-\eta_j)^2}{2\mu_j y_i}\right) \tag{2.6}$$

where, similar to the Gaussian discussion above, λ_j and η_j are another set of model parameters associated with each model state j. A qualitative depiction of the IG distributions is shown in Fig. 2.4.

2.6.2 Transition Types

The transition probabilities, $\tau(j, l)$ enumerates the probability that state j (computed for an event at time $i - 1$) will transition into state l (being computed for an event at time i). Modelling these transitions adequately requires some deeper thinking about the way molecule properties may be captured by the nanopore sensor.

2.6.2.1 Steps

Ideally, DNA bases flow through the sensor in an orderly manner, and each event corresponds to the entry/exit of a base to/from the pore. For example, suppose k is three and that a DNA sequencing fragment, `ACGTC`, is going through the nanopore. The first registered event is corresponds to the first three bases, `ACG`. The following event will be a signal indicative of `CGT` when the new base `T` enters the pore (and the "oldest" base, `A`, exits). Next, `GTC` will be responsible for the following event. This ideal progression of DNA through the pore is referred to as a *step* transition.

Ideally this would be the only such transition registered by our measurement system. Unfortunately non-idealities in our sensing system give rise to other possible mechanisms. The logic of these is illustrated in Fig. 2.5 (which considers transitions from the example state `AACGAC`) and elaborated below. Hence, we can only assume that step transitions occur with some probability p_{step} (black arrow shown in Fig. 2.5).

2.6.2.2 Stays

An easy alternative "transition" to imagine is the possibility that the same base has been accounted for twice. That is, a new event is reported at time i, but this event actually corresponds to exactly the same base molecule for which another event was registered at time $i - 1$. In essence, we have not measured a new state. In the HMM trellis we essentially have a transition from state j to state j and we should account for its possibility. This transition is referred to as a *stay* and is depicted by the blue arrow in the example of Fig 2.5. The probability that this transition happens is denoted with p_{stay}.

2.6.2.3 Skips

Alternatively, it is also possible that more than one bases pass through the pore between registered events at $i - 1$ and i. That is, our measurement system has effectively failed to provide events for intervening bases. This transition mechanism is called a *skip*(red arrow shown in Fig. 2.5). The probability that this transition happens is denoted with p_{skip}.

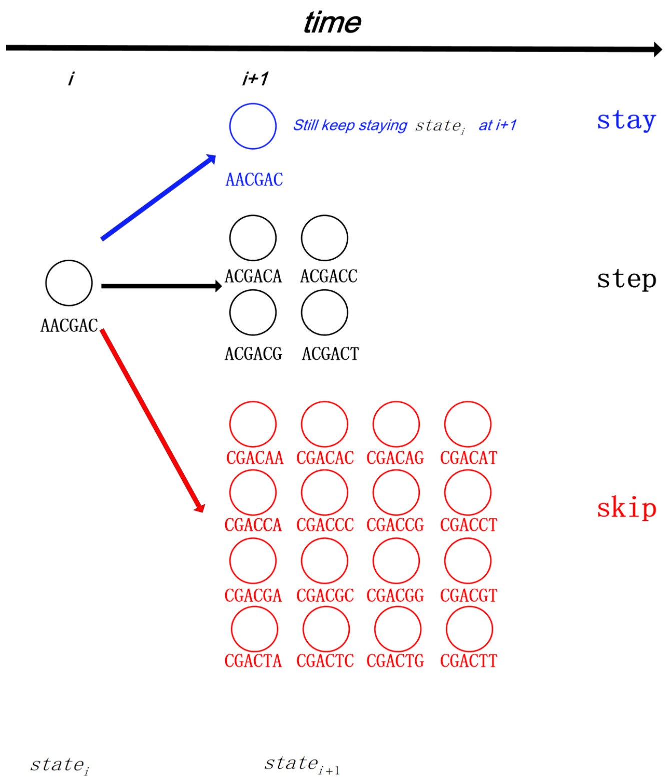


Figure 2.5: Three different types of transition mechanisms.

2.6.3 Transition Probability

From Fig. 2.5 it is clear that, under the above noted transition mechanisms, a state at time i has some chance of transitioning into any of a number of possible states at time $i+1$ (21 to be exact, under our assumptions elaborated below). All these transitions may be encapsulated by the one relation [43].

$$\begin{aligned}\tau(j,l) = &\delta_{l=j} \cdot p_{stay} \\ &+ \delta_{\text{suffix}(j,\text{Mer}-1)=\text{prefix}(l,\text{Mer}-1)} \cdot p_{step} \cdot \frac{1}{4} \\ &+ \sum_{i=2}^{\text{Mer}-1} \delta_{\text{suffix}(j,\text{Mer}-i)=\text{prefix}(l,\text{Mer}-i)} \cdot \frac{p_{skip1}^{i-1}}{4^i} \\ &+ \sum_{i \geq \text{Mer}} \cdot p_{skip1}^{i-1} \cdot \frac{1}{4^{\text{Mer}}}\end{aligned} \tag{2.7}$$

we elaborate on the structure and components of this expression now.

The expression is a sum of four main terms, each written in a separate row. The top three rows are filtered with a Kroenecker-delta adhering to the following behaviour: delta is 1 if its subscripts are equal, 0 otherwise. These identifier terms allow (2.7) to methodically assign appropriate probabilities to any transition (from any state j to any state l).

In (2.7), the prefix(a,b) (suffix(a,b)) functions, returns the equivalent base-4 numerical index of the left-most (right-most) b baes of state (number) a. They are essentially a way for us to relate the bases that are in the pore at consecutive times (i and $i+1$).

Thus the first addend in (2.7) denotes that a stay occurred if the function areguments are such that $l = y$. The second addend accounts for the probability of a simple step, the third addend accounts for all the ways that the next state could be reached via a skip between 2 and Mer $-$ 1 and the final addend (bottom row) accounts for all other potential skip contributors (i.e. greater than Mer $-$ 1 bases).

Of note is that p_{skip} is the overall probability of making a skip of any type. It says that a state was skipped, but it does not say how many states were skipped. In general one can express this as the combination of probabilities to skip 1, 2, 3, and so on states:

$$p_{skip} = p_{skip1} + p_{skip2} + p_{skip3} + \dots \tag{2.8}$$

Assuming that that multiple-state skips are just combinations of single-state skips we have

$$p_{skip} = p^1_{skip1} + p^2_{skip1} + p^3_{skip1} + \dots \tag{2.9}$$

which is

$$p_{skip} = \frac{p_{skip1}}{1 - p_{skip1}} \tag{2.10}$$

and therefore

$$p_{skip1} = \frac{p_{skip}}{1 + p_{skip}}. \tag{2.11}$$

An enumeration of all possibilities in our model (which assumed that only one state or more than $Mer - 1$ are skipped) results in 21 possible transitions from state j to l

$$\text{21 transitions} = \text{1 stay} + \text{4 steps} + \text{16 skips}. \tag{2.12}$$

2.7 Algorithmic Improvements

2.7.1 Logarithmic Probabilities

Algorithm implementation on hardware requires some optimizations to reduce hardware design complexity. A nearly universal simplification for problems of this type (in pure software implementation as well) is to replace absolute probabilities with their logarithms. This converts multiplications to sums, a great simplification. Since the logarithm is monotonic it does not compromise the *max* function's result in the VA. Thus, the critical calculations are transformed from

$$v_j(i) = b_j(i) \cdot \max_v [v_v(i-1) \cdot \tau(v,j)] \tag{2.13}$$

to

$$\alpha_j(i) = \ln v_j(i) = \ln b_j(i) + \max_v [\ln v_v(i-1) + \ln \tau(v,j)]. \tag{2.14}$$

Thus our emission calculation becomes

$$\begin{aligned}\ln b_j(x_i, y_i, t_i) = & - \ln 2\pi - \ln \sigma_j \\ & - \left[\frac{x_i - \mu_j}{\sqrt{2}\sigma_j}\right]^2 \\ & - \frac{\ln \lambda_j}{2} - \frac{3}{2} \ln y_i \\ & - \frac{1}{2y_i} \cdot \frac{\lambda_j}{\eta_j^2} \cdot (y_i - \eta_j)^2.\end{aligned} \tag{2.15}$$

Considering the negative of this is convenient (hardware additions can be made slightly more efficient than hardware subtractions) giving

$$\begin{aligned}- \ln b_j(x_i, y_i, t_i) = & \ln 2\pi + \ln \sigma_j \\ & + \left[\frac{1}{\sqrt{2}\sigma_j}(x_i - \mu_j)\right]^2 \\ & + \frac{\ln \lambda_j}{2} + \frac{3}{2} \ln y_i \\ & + \frac{1}{2y_i} \cdot \frac{\lambda}{\eta_j^2}(y_i - \eta_j).\end{aligned} \tag{2.16}$$

The proposed hardware design has not provisioned the IG parameters when implementing the emission calculation module, a simplification that has been left to future work. It should be noted however that ignoring the IG part of the emission calculation only marginally affects the accuracy for the scenarios considered; however it remarkably boosts the computational speed. After removing the IG related items, the new emission equation presented in hardware is

$$- \ln b_j(x_i, y_i, t_i) = \ln 2\pi + \ln \sigma_j + \left[\frac{1}{\sqrt{2}\sigma_j}(x_i - \mu_j)\right]^2. \tag{2.17}$$

For the transition probability $\ln \tau(\nu, j)$, a look-up table containing pre-computed transition probability logarithms for any combination of states can be used to achieve the calculation. This is the case since the transition structure (and values) are fixed and independent of any particular measurement. This trades computational power for storage area. The memory location to access for any particular transition is dictated by the results of the prefix and suffix functions which may be implemented with

$$\text{prefix}(i, k) = i \gg (2(\text{Mer} - k)) \tag{2.18}$$

$$\text{suffix}(i, k) = i \wedge (1 \text{ bit and}(2k) - 1) \tag{2.19}$$

where the $\gg$ operator denotes a bit shift to the right and where the *wedge* operator denotes an exclusive OR logic expression.

2.7.2 Continuous Traceback

As a reminder, the VA's traceback procedure refers to the backward progression through the trellis after the m posteriors corresponding to the last event (stage), n, have been computed. This backward progression is, of course, through the set of pointers, $\text{ptr}_j(i)$, that have been computed alongside the posteriors. The sequence of states traversed in this backwards recapitulation achieves the aforementioned statecall (and practically concludes the majority of the basecalling process).

In the VA summary provided in Algorithm 2.5, the traceback segment, in its entirety, follows the “forward” calculation. This obviously imposes a delay, a whole sequence of length n must be processed before traceback may commence. Besides compromising throughput this also has memory implications as the pointers for a whole strand sequence must be retained.

As the reader may appreciate however this is only an artificial constraint. There is nothing stopping the basecaller from starting traceback before the completion of a strand. Naturally this may compromise accuracy somewhat (as only partial sequence information is used to predict the most likely path up to the chosen traceback point), but this compromise is usually negligible if the “depth” at which an abbreviated traceback starts is not too small (often a matter of experimental adjustment given the non-linear nature of the sequence detection procedure).

What’s important about this procedure is that once our calculations have reached some chosen depth, ensuing outputs can be achieved in a continuous manner if the calculation is properly pipelined. That is, one does not have to wait for another depth to produce a sequence detection, basecalls may then emerge at the same rate as events are processed, albeit delayed by the size of the traceback depth. This *continuous traceback* [44] approach essentially results in (latent) real-time basecalling.

An abbreviarte pseudo-code description of the VA, this time incorporating continuous traceback, is given in Algorithm 2. As shown, a traceback window (i.e. depth) of length, l is defined, accumulated, and then progressively updated and traversed to produce continuous outputs.

Algorithm 2 VA Basecalling with Continuous Traceback

```
define $l$ is window size
define window array[$l$]
for $i = 1 \rightarrow n - 1 + l$ do
    for $j = 0 \rightarrow m - 1$ do
        for $v = 0 \rightarrow 20$ do
            compute posterior probability
            record pointer
        end for
    end forcontinuous traceback:
    if $i < l$ then
        push pointers to the window
    else if $i \geq l$ and $i \leq n - 1$ then
        for $z \rightarrow l$ do
            do traceback in the window
        end for
    else
        for $z = l - 1 \rightarrow 0$ do
            find the end state at z col.
            do traceback with the end state
        end for
    end if
end for
```

2.8 Chapter Summary

In this chapter, the structure and solution of an HMM-based DNA basecaller is described. The behaviour of a nanopore DNA sensor is explained in terms of an HMM. The associate of a sequence of (event) observations is then associated with this model and a means of detecting the underlying base sequence to traverse the sensor explained in terms of the Viterbi algorithm. Means of improving the algorithm by simplifying its calculations such that simpler hardware may be employed are given.

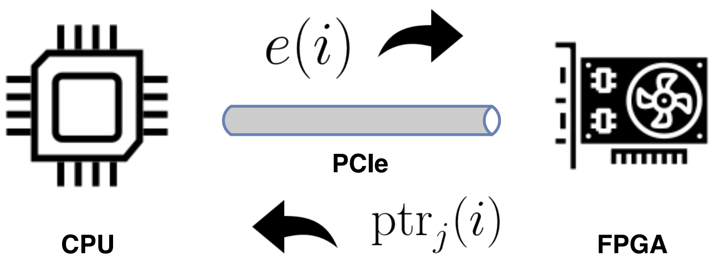
3 Basecalling Engine Implementation on FPGA

3.1 System Level Architecture Overview

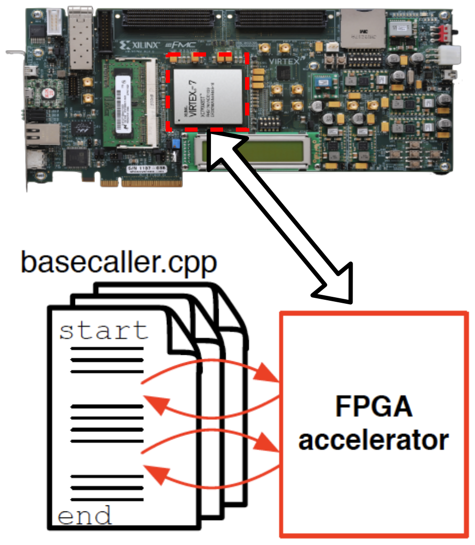
To maintain a flexible and scalable basecalling accelerator we implement an FPGA co-processor that communicates with a host CPU via PCIe as drawn in Fig. 3.1(a). More exactly, a portion of the basecaller is implemented in standard procedural code (e.g. a program called basecaller.cpp written in C++) that interacts directly with the FPGA accelerator (an illustration of this idea is given in Fig. 3.1(b)).

This communication link is facilitated by the Reusable Integration Framework for FPGA Accelerators (RIFFA), an open-source collection of software libraries and hardware designs to enable CPU-FPGA communication over PCIe [45]. A discussion of this link is given in Chapter 4.

The $e(i)$ term is the accelerator input, which is transmitted from the CPU. In response, M $\mathrm{ptr}_j(i)$ terms (M is the number of HMM states) are generated by the accelerator and are sent back to the CPU. These exchanges correspond to one event-loop iteration (see line 4 of Algorithm 1 in the previous Chapter). The CPU



(a)



(b)

Figure 3.1: The overview of the basecalling accelerator.

then starts embarking on a traceback procedure.

These main processing steps are summarized in Algorithm 3, below. The `fpga_send` and `fpga_recv` functions (handles) are RIFFA commands used by the CPU-based software to communicate inputs and outputs with the CPU (they are further discussed in Chapter 4).

Algorithm 3 An accelerated HMM statecaller

```
1: fpga_send($e(0), e(1), ..., e(N-1)$)
2: fpga_recv($\mathrm{ptr}_0(0), ..., \mathrm{ptr}_{63}(0), .., \mathrm{ptr}_0(N-1), ..., \mathrm{ptr}_{63}(N-1)$)
3: $\pi^*(N-1) \leftarrow$ fpga_recv($\arg\min_j [v_j(N-1)]$)
4: for $i \leftarrow N-1{:}1$ do
5:     $\pi^*(i-1) \leftarrow \mathrm{ptr}_{\pi^*(i)}(i)$
6: end for
```

From the CPU's perspective then, the states-loop (line 5 of Algorithm 1 in Ch. 2) is simply replaced by the aforementioned RIFFA application program interface (API) handles to the FPGA hardware (details in Ch. 4). This is shown in the updated basecaller code of Algorithm 3 where line 1 dispatches the new event $e(i)$ to the FPGA and line 2 records the state (at time $i-1$) most likely to have preceded the state j (at time i). The traceback procedure is contained in lines 4 and 5 of Algorithm 3.

3.2 The Potential of Hardware Solutions

An examination of the basecalling algorithm discussed in Chapter 2 shows that approximately 1.5×10^6 arithmetic operations are needed to process an event if M is 4096 ($k = 6$) [46]. Thus, for a device such as the MinION, which possesses about 500 simultaneous sensor channels, each channel working at a rate of 1000 events/sec, then the total number of operations are 750 GOPS (Giga-operations-per-second).

For reference, a performance-laptop processor such as the 14-nm, 2.6-GHz, 4-core, Intel®Core$^{\text{TM}}$i7-6700HQ is benchmarked at 83 (GOPS) on 56 W of power consumption [47]. According to the above estimate, this has the potential to achieve 5.5×10^4 events/s. An 8-thread run of a C++ basecaller implementation on such a machine achieved a total throughput of only 10,400 events/s [48] however, a 39$\times$ shortfall relative to the rate desired for real-time operation.

A number of technologies are becoming available to accelerate algorithms such as the basecaller under consideration here. Prominent examples include many-core coprocessors such as the Xeon Phi family which can sustain ~200 threads and graphics processing units (GPUs) which have the ability to launch 1000s of threads [49]. Such units regularly achieve +10$\times$ acceleration over standard implementations. Capable of interfacing to a standard computing platform via a PCIe bus, these units are very convenient for use, but their performance boost comes at too large a power requirement for the mobile solution sought here.

Systems-on-chip (SoC) approaches provide another possibility [50]. In this space high-end smartphones may be able to sustain 20 GFLOPS within a 3-W power window [51]. Alas the availability of such extreme solutions for bioinformatics remains distant and much more immediate solutions are necessary. We detail our approach next.

3.3 Basecaller Hardware Used

The FPGA hardware chosen for this work is a device from the Xilinx Virtex-7 (V7) family, specifically the VC707. The test-board housing the VC707 is shown in Fig. 3.2. This chip is built using a 28-nm planar CMOS technology [52]. The V7 device consists of 485k logic cells and 2800 DSP units each of which contains arithmetic components for addition/subtraction, multiplication, logic, and shifting [53, 54]. These components are used to implement computationally intensive parts of the basecaller, other parts remaining in the CPU working simultaneously in conjunction with the FPGA as already noted above.

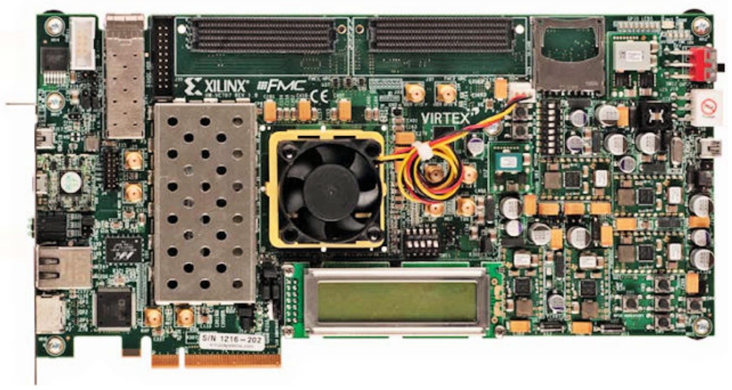


Figure 3.2: Xilinx VC707 FPGA board for the hardware implementation [52].

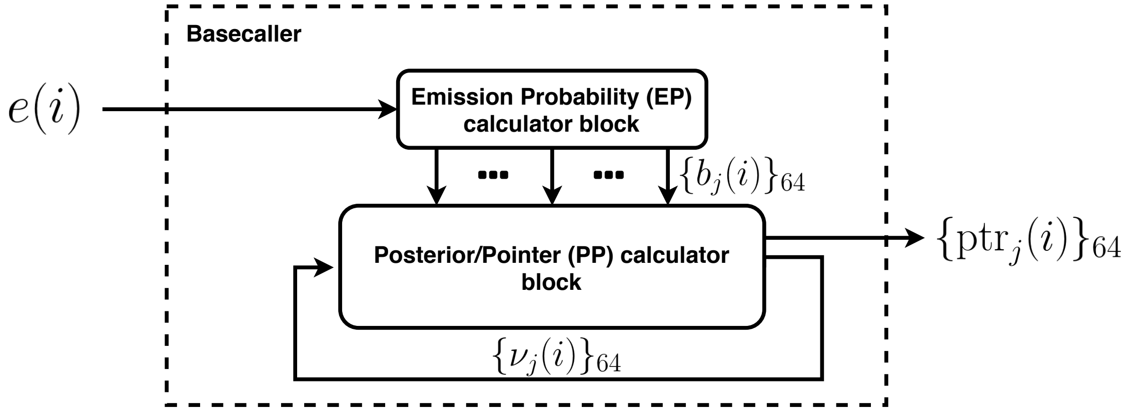

Basecaller
$e(i)$
Emission Probability (EP) calculator block
$\{b_j(i)\}_{64}$
Posterior/Pointer (PP) calculator block
$\{\mathrm{ptr}_j(i)\}_{64}$
$\{\nu_j(i)\}_{64}$

Figure 3.3: The accelerator architecture shown in high-level.

3.4 The Basecalling Accelerator Architecture

A high-level picture of the FPGA accelerator's architecture is shown in Fig. 3.3. As shown, the system is composed of two main blocks, an emission probability (EP) calculator block and a posterior/pointer (PP) calculator block, both ultimately operating in parallel.

At this level, the general operation of the accelerator is as follows (and is detailed in the following sub-sections): EP consumes events $e(i)$ (from the CPU) in a streaming fashion and produces corresponding emission probability calculations $b_j(i)$ (rather its logarithm, recall Eq. (2.17) in Ch. 2). The $M = 64$ $b_j(i)$, $\{b_j(i)\}_{64}$, outputs of EP are fed into the PP block. The PP block uses this to continually update the posterior probabilities v_j of all states ($j \in M$). This is an iterative process, updated with each new i, as indicated by the feedback loop drawn across the PP block in Fig. 3.3. At the same time, pointer values associated with each state are calculated by and output from the PP and sent back into the CPU which uses these to complete basecalling. We elaborate on these basecaller blocks now.

3.4.1 Emission Probability (EP) Hardware Architecture

The EP block is tasked with executing the following emission probability calculation for each input event $e(i)$

$$-\ln b_j(x_i, y_i, t_i) = \ln 2\pi + \ln \sigma_j + \left[\frac{1}{\sqrt{2}\sigma_j}(x_i - \mu_j)\right]^2. \tag{3.1}$$

As noted in Ch. 2, x_i is the mean feature of each event $e(i)$ and the standard deviation feature, y_i, is not considered in this work to simplify the complexity of the algorithm, but it still be considered as the constant value and pre-treated when going to the FPGA. Thus, the parameter to represent the event we will discard y_i so that parameters $\ln \sigma_j$ and $\frac{1}{\sqrt{2}\sigma_j}$ can be removed.

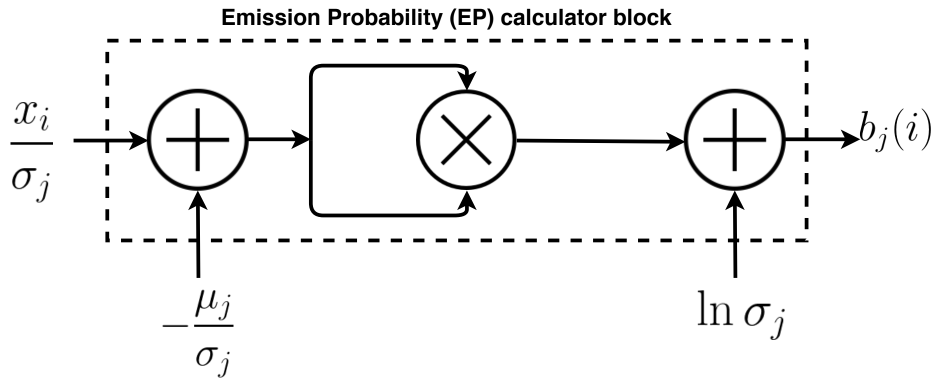

Emission Probability (EP) calculator block
$\frac{x_i}{\sigma_j}$
$-\frac{\mu_j}{\sigma_j}$
$\ln \sigma_j$
$b_j(i)$

Figure 3.4: The breakdown drawing of the EP calculator block.

Some simplifications to this expression are possible that make hardware implementation simpler without sacrificing accuracy. For example, the constant offset $\ln 2\pi$, does not numerically distinguish state emission probabilities from one another. Therefore, there is no need to compute it. A similar argument applies to the $\sqrt{2}$ value.

Also, we can avoid having to divide in the hardware, by pre-scaling our input (x_i) and model parameters (μ_j) with σ_j and pre-computing model terms such as $\ln \sigma_j$ which do not change during measurement. Thus, EP may be tasked with computing the following:

$$-\ln b'_j(x_i, y_i, t_i) = \ln \sigma_j + \left[\left(\frac{x_i}{\sigma_j} - \frac{\mu_j}{\sigma_j} \right) \right]^2. \tag{3.2}$$

A block diagram of EP (for just one $b_j(i)$ out of $M = 64$ operating in parallel) is shown in Fig. 3.4 corresponding to the elemental operations comprising (3.2).

3.4.2 Posterior/Pointer (PP) Hardware Architecture

The state-loop can be partitioned and mapped into identical, and independently operating hardware units. Each of these units is tasked with the calculation of (3.3) and (3.4)

$$\ln v_j(e_i) = \ln b_j(e_i) + \min_{\nu}[\ln v_\nu(e_{(i-1)}) + \ln \tau(\nu, j)], \tag{3.3}$$

$$\mathrm{ptr}_j(i) = \min_{\nu}[\ln v_\nu(e_{(i-1)}) + \ln \tau(\nu, j)], \tag{3.4}$$

for a particular state j. We refer to each such unit as a *state slice*. The architecture of one state slice is shown in Fig. 3.5 where double-stroke arrows indicate a bundle of signals (i.e. 21 transition signals $\tau_{\boldsymbol{\omega}_j}$ and 21 posteriors from the previous iteration $v_{\boldsymbol{\omega}_j}(i-1)$ in Fig. 3.5 where the ln function is dropped for symbolic brevity) and the single-stroke arrows indicate an individual signal.

Each state slice consumes at least 21 + 1 adders (recall there are 21 possible transitions into a state and one more addition needed to process the emission probability $b_j(i)$) and needs to achieve a 21-way sort (as shown in Fig. 3.5) that identifies the state with a minimum (negative logarithm) transition probability into a state.

3.4.3 Complete Basecaller Architecture

Finally, a blow-by-blow accelerator architecture as shown in Fig. 3.6 breaks down into an EP block, a PP block composed of M *state slices*, a transition model, a

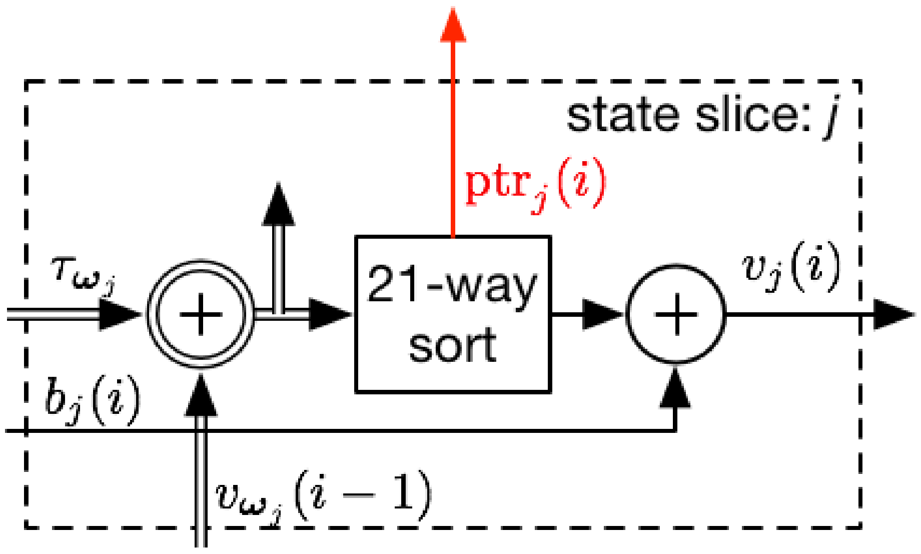


Figure 3.5: The breakdown drawing of the PP block (one of 64)

distributor, and a normalization block.

The transition model is a register array storing the pre-computed transition probabilities between the HMM states. The distributor block, a look-up-table (LUT), makes sure that the appropriate 21 posteriors are funnelled back to each state slice. The normalization block, composed of an M-input multiplexer and a subtractor, sets the current epoch's posterior calculations to

$$v_j(i) \leftarrow v_j(i) - \min\left(\{v_j(i)\}_M\right). \tag{3.5}$$

Thus, it prevents numerical overflow of the accumulated posteriors values.

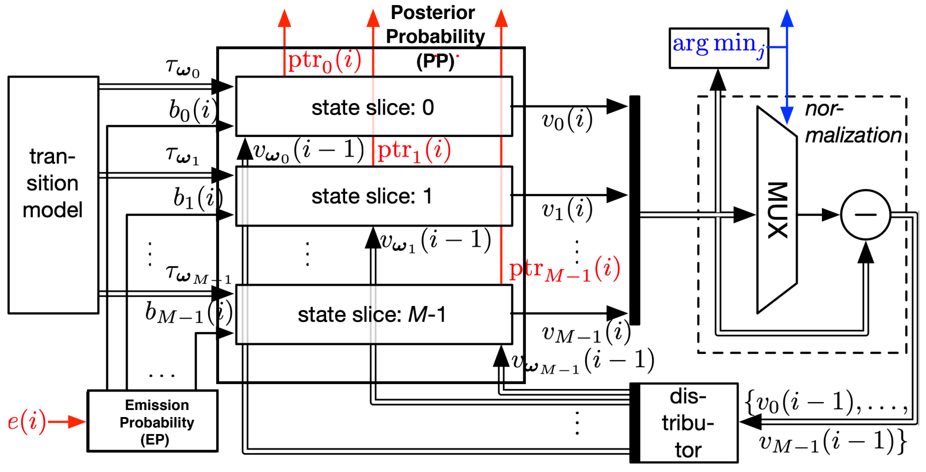


Figure 3.6: The thorough basecalling accelerator system architecture [30].

3.5 The Selection of Bitwidth

Another critical factor that should be considered in our basecaller IP design is the accuracy with which it represents its internal variables. That is, the number of bits (i.e. the *bitwidth*) with which the numerical quantities in the basecaller are described.

The input and the inner data variables are uniformly described with 40 bits in this design. This value, chosen as a conservative setting at the outset of the basecaller design efforts, allows essentially a 10^{12} fluctuation numerical fluctuation. The 48-bit width that could be accommodated by the FPGA's DSP hardware certainly makes this choice executable, but it is not an optimum setting to use.

The reason for this is that, for one, the data, $e(i)$, coming from the CPU to the FPGA is only 32 bits (the smallest floating-point format available in C). Second, the nanopore model used to generate the $e(i)$ values, produced quantities requiring only an 18-bit range. Accounting for the multiplication operation (i.e. the square in the emission block), only a maximum 36-bit range would have been required.

3.6 Chapter Summary

In this chapter, we described a scalable (via the state-slice architecture and modified emission computations) and robust (40-bit width, posterior normalization) architecture of the basecaller accelerator platform. The whole platform consists of a program running on the CPU that offloads computationally intensive workloads to the FPGA co-processor.

4 System Communication and Timing

4.1 Communication Interfaces

Establishing a robust communication link between the CPU and its FPGA accelerator is a critical part of this thesis work. A streaming (i.e. continuous) link between the CPU software and its FPGA co-processor is essential for realizing efficient computations, especially for real-time DNA processing systems as envisioned in this thesis. That is, as measurement data continually flows through the CPU, we expect a similar, uninterrupted exchange between the CPU and the FPGA. Thus, no need for excessive buffering or delay will be encountered during the basecalling process. The faster that this CPU-to-FPGA communication can work, the more data we can process per unit time (i.e. the higher the data throughput).

The FPGA board provides various external communication connections such as USB, Ethernet, GPIO, and PCIe through which links to outside peripherals can be made. Since the basecalling process needs high throughput data streaming transmission, we chose PCIe as the communication link between the CPU and

FPGA. For PCIe 2.0, a typical 16-lane PC motherboard can sustain peak data exchanges of 64-Gbps, far outstripping all other readily available communications options [55, 56, 57].

However, PCIe is a sophisticated, multi-layer, transmission protocol that is very hard to implement from scratch. For example, it must be able to handle disordered and overlapping communication packets and deal with data redundancy [58]. Specifically, it requires the designer to implement a virtual communications interface spanning several layers in a sophisticated communication hierarchy.

The RIFFA CPU-FPGA communications framework mentioned in the previous chapter helps address this challenge. It provides a streaming low-level protocol to our basecaller that can be interpreted with custom hardware implemented directly on the FPGA. All high-level PCIe protocol concerns are also implemented, but effectively hidden by RIFFA from the FPGA designer. The overall (low and high-level) RIFFA link allows the FPGA core to maintain coherent access to the CPU's main memory and thus the CPU itself (which communicates with the FPGA though main memory). To accomplish this, RIFFA is essentially a direct-memory-access (DMA) bus master powered by vendor-specific PCIe IP [59, 60], such as Xilinx, Altera. This IP implements the physical protocol implementation in FPGA hardware.

We summarize the advantages of employing RIFFA in our project:

- RIFFA has good flexibility with up to 12 independent channels. These channels share one PCIe bus at one time.
- RIFFA provides support for a communicating with multiple FPGAs in parallel, a facility important for our evolving research.
- RIFFA is an open-source project without financial and copyright issue [61].
- RIFFA provides a user-friendly communication interface on both the software and the hardware side (a compact handshaking protocol) of a CPU-FPGA implementation and hides complex PCIe protocol details from FPGA designers.
- RIFFA's maximum utilization takes advantage of up to 97% of the available PCIe bandwidth thus satisfying very high-speed communication needs [45].

The manner in which RIFFA may interface with the software and hardware components of an FPGA-accelerator project is detailed in the following two subsections.

4.1.1 RIFFA Software Interface

RIFFA provides designers with communication tools for both the software (CPU-side) and hardware (FPGA-side) of the accelerator interface. On the CPU side, RIFFA provides a C based API (the RIFFA API commands are listed in Table 4.1).

The API function `fpga_list()` enumerates all recognized FPGA devices and returns their unique IDs. This is essentially the means by which a program running on the CPU identifies the FPGAs with which it may communicate. Based on the ID, the corresponding FPGA is initialized by a CPU program through the function `fpga_open()`. Then, the function `fpga_send()` is used to send the content of some appointed memory address and its data size (both appearing as arguments in that function) from the CPU to the FPGA. Specifically, the data size denotes how many 32-bit words we intend to send to the FPGA.

After sending data to the FPGA, the CPU invokes the `fpga_recv()` RIFFA command. This command is called to look for and fetch any data that may have been transferred from the FPGA to the PC's memory. Also, `fpga_recv()` has to point out which FPGA and which channel should be monitored and where to store the returned data. Eventually, when the FPGA operation is done, the FPGA should be closed by invoking the RIFFA command `fpga_close()` from the CPU.

4.1.2 RIFFA Hardware Interface

RIFFA not only provides the software interface but also builds the hardware interface present on the FPGA itself. All ports are divided into two groups: Receive ("**RX**") and Transmit ("**TX**") as interpreted from our FPGA accelerator's perspective.

Table 4.1: RIFFA software API

function name	parameters	return value type
fpga_list	fpga_info_list* list	int
fpga_open	int id	fpga_t*
fpga_reset	fpga_t* fpga	fpga_t* fpga
fpga_send	fpga_t* fpga, int chnl, void* send_data, int len, int offset, int last, long timeout	int
fpga_recv	fpga_t* fpga, int chnl, void* recv_data, int len, long timeout	int
fpga_close	fpga_t* fpga	void

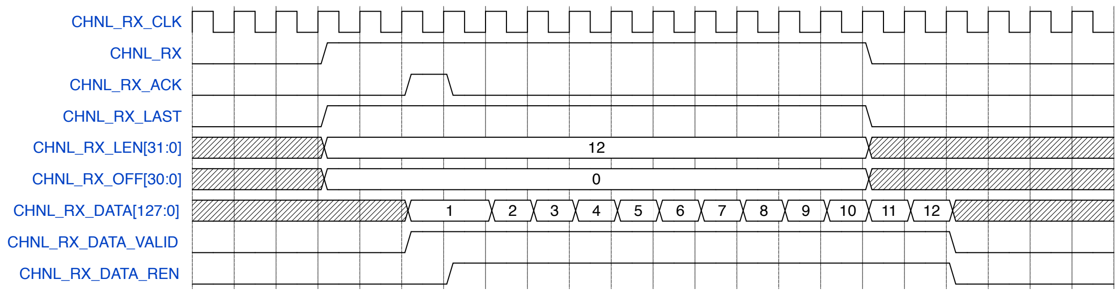


Figure 4.1: The diagram of the **RX** signal waveform.

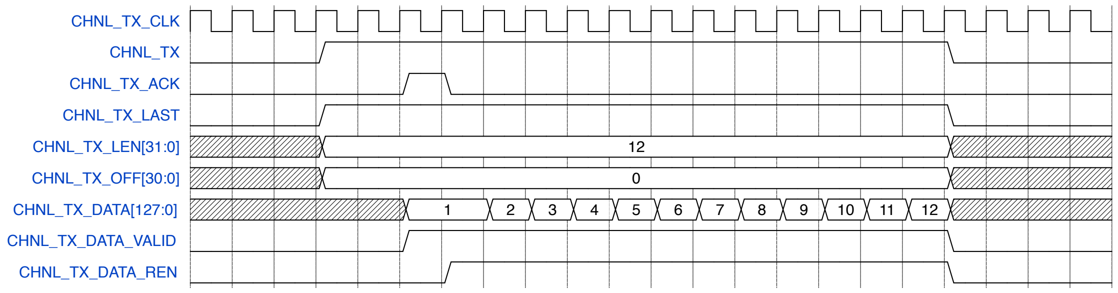


Figure 4.2: The diagram of the **TX** signal waveform.

In Table 4.2, the port names and their stream direction (again, from the FPGA accelerator core's perspective) are illustrated. The port *CHNL_RX_CLK* and the *CHNL_TX_CLK* separately define the clock signals for reading and writing FIFOs inside the RIFFA architecture. In our case, the clock signal is generated by the vendor's (i.e. Xilinx) clock generator IP.

Examples of, the waveforms on the RIFFA **RX** and **TX** ports are shown in Fig. 4.1 and Fig. 4.2, respectively. They illustrate the simulation waveform in a situation 12 RIFFA packets are sent and received. The purpose of the ports is explained further in the following paragraphs.

The signal on the *CHNL_RX* port from RIFFA informs the user's FPGA IP that the FIFO on the **RX** side has started to receive new data from the CPU. When the *CHNL_RX* signal goes high, the *CHNL_RX_ACK* signal should be pulsed high (for at least one clock cycle) by the user's custom FPGA communication logic to acknowledge the new incoming data.

The *CHNL_RX_DATA* port is the FIFO output, the data to be actually processed by the FPGA. The bitwidth of this signal is set by the **DWIDTH** parameter, which is also the reference value of the PCIe packet size defined in the vendor's IP. The **DWIDTH** parameter can be set to two values: 64 bits and 128 bits [45]. In order to maximize the number of data bits processed per cycle, and thereby maximize the throughput, we set this to 128 bits.

The signals on the *CHNL_RX_DATA_VALID* and *CHNL_RX_DATA_REN* ports implement a simple flow control, the former coming from the CPU side and indicating when data on the FIFO output port is safe to get and the ladder coming from the FPGA side and indicating when a FIFO output has been registered by the FPGA. When the *CHNL_RX_DATA_VALID* and *CHNL_RX_DATA_REN* are both high, the value on the *CHNL_RX_DATA* is considered to be consumed by the FPGA.

The **TX** ports are similar to the **RX**'s, the only difference is the direction of the ports are opposite, and the user needs to specify when the **TX** should be active

Table 4.2: RIFFA hardware API

RX I/O definition	**I/O**	**TX I/O definition**	**I/O**
CHNL_RX_CLK	O	*CHNL_TX_CLK*	O
CHNL_RX	I	*CHNL_TX*	O
CHNL_RX_ACK	O	*CHNL_TX_ACK*	I
CHNL_RX_LAST	I	*CHNL_TX_LAST*	O
CHNL_RX_LEN[31:0]	I	*CHNL_TX_LEN[31:0]*	O
CHNL_RX_OFF[31:0]	I	*CHNL_TX_OFF*	O
CHNL_RX_DATA[DWIDTH:0]	I	*CHNL_TX_DATA[DWIDTH:0]*	O
CHNL_RX_DATA_VALID	I	*CHNL_TX_DATA_VALID*	O
CHNL_RX_DATA_REN	O	*CHNL_TX_DATA_REN*	I

and what data should be sent to CPU. Similarly, the *CHNL_TX_DATA* is consumed only when both the *CHNL_TX_DATA_VALID* and the *CHNL_TX_DATA_REN* are high.

Because RIFFA IP generates one *REN* and one *VALID* on either **RX** or **TX**, the FPGA should fully concentrate on how to set the appropriate controlling signal to ensure the data is processed properly. Also, since the FIFO has an external operation to confirm the valid data, it causes the first data packets to always duplicate as shown in Fig. 4.1. The simple solution is to postpone the *CHNL_RX_DATA_REN* or to move *CHNL_TX_DATA_VALID* forward one clock period for inactivating the repeated data.

4.2 RIFFA Customization

For the purpose of improving the throughput, reducing the delay times between the FPGA and CPU memory, and increasing the efficiency of the RIFFA channels, we have slightly adjusted RIFFA and attached some other functional blocks. These changes included software and hardware additions as described below.

4.2.1 Software Configuration

On the software side, the main improvement was a multithreaded implementation of the *fpga_send* and the *fpga_recv* functions thus allowing simultaneous send-receive operation. A process flow diagram (PFD) indicating the incorporation of this into the CPU-side basecaller is shown in Fig. 4.3. As shown in this figure, the multi-thread implementations is capable of running the *fpga_send* and the *fpga_recv* in parallel.

The fixed point converter at the top is arranged to transform the input data (which is available in floating-point format) into fixed-point format for consumption by the FPGA. At the bottom, the traceback function is implemented as discussed in Ch. 2.

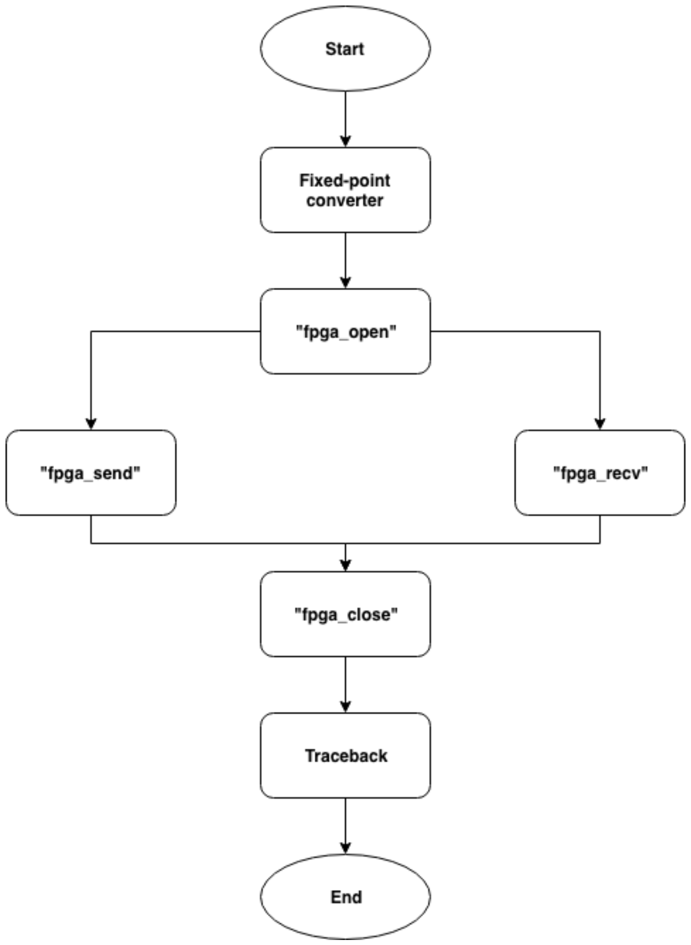


Figure 4.3: The PFD shows the program execution steps on the CPU side.

4.2.2 Hardware Configuration

A number of hardware additions were made to the base RIFFA FPGA offerings to make it suitable for the basecalling accelerator.

First, clock generator IP was employed to generate customized clock frequency for the basecaller (first mentioned in 4.1.2). The on-FPGA RIFFA hardware works with the PCIe 100-MHz reference clock. Since this setting is not adjustable, other clock sources are needed to accommodate the alternate clock cycle needs of custom FPGA basecalling hardware. For this purpose, the clock generator IP can easily generate any frequency clock cycle with resources such as delay-locked loops (DLLs) and mixed-mode clock managers (MMCMs) on the FPGA.

Second, we increased the FIFO depth to reduce the data swiping times between the CPU memory and the RX FIFO in RIFFA. The depth was enlarged from 64 to 1024. Under this setting, the data is never refreshed until all of the data is stored in the RX FIFO. Correspondingly, the TX FIFO is also enlarged to 1024 so that once the FIFO is full an interrupt will be made by the controller to notify the DMA on the PC side to fetch the data and store it into assigned memory space.

Third, we made a periodic pulse control for the *CHNL_RX_DATA_REN* port that asserts after each RIFFA packet completes processing. Through this change, the RIFFA **RX** FIFO only pumps out the new data in a certain time to make sure the basecaller IP core has enough time to process every 128-bit RIFFA packet (each packet contains 4 observed events mean value).

Last, the bitwidth of the sending and receiving data on the hardware side is adjusted due to the different definitions of the bitwidth in our IP's IO. As the previous Chapter mentioned, the input bitwidth of the basecaller is 40 bits while the output bit-width is 8 bits. Each RIFFA packet, both on the **RX** and **TX**, is 128 bits. On the one hand, we enlarge each incoming event from 32 bits to 40 bits on **RX** by filling to zero the left-most 8 bits; on the other hand, each 16 pointers are loaded into one packet on **TX**. As the matter of the fact, each event will produce $M = 64$ pointers, thus 4 RIFFA packets are required for them.

4.3 RIFFA Receiver and Transmitter Design

Although RIFFA provides a simplified interface protocol between FPGA IP and PCIe, still needed is on-FPGA receiver and transmitter hardware to manage our customized low-level communications between the FPGA IP and RIFFA interface. This includes not only the hardware to manage signalling with RIFFA's handshaking and flow-control ports (as discussed above), but also means of packing FPGA data into RIFFA packets. As previously mentioned, each **RX** packet has 4 event values, and each event will return $M = 64$ 8bits pointers per iteration process; means of efficiently loading and unloading such representations in/out RIFFA packets is

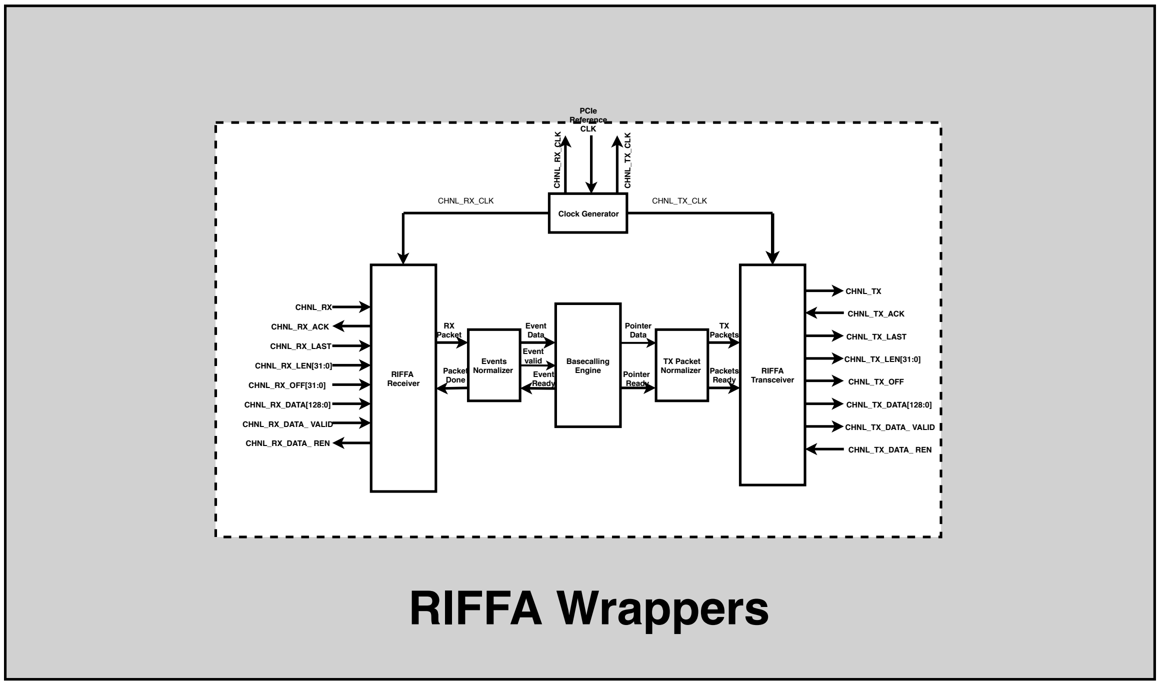


Figure 4.4: The whole accelerator design with RIFFA's hardware API

needed if efficient computational dataflow is to be maintained.

Fig. 4.4 showcases our design ideas on this subject. In the figure, the clock generator is used to generate two clock signal for all visible sub-modules. Our basecalling engine (BE) is naturally deployed in the middle position between *RX* (data arriving from the CPU via RIFFA) and *TX* (data being send to the CPU via RIFFA) modules.

As shown, two modules are used for handling RX, the RIFFA receiver (RR) and the Events Normalizer (EN). The RR is a FSM for controlling signal processing. The RR state machine consists of three states, **Initialization**, **Processing**, and

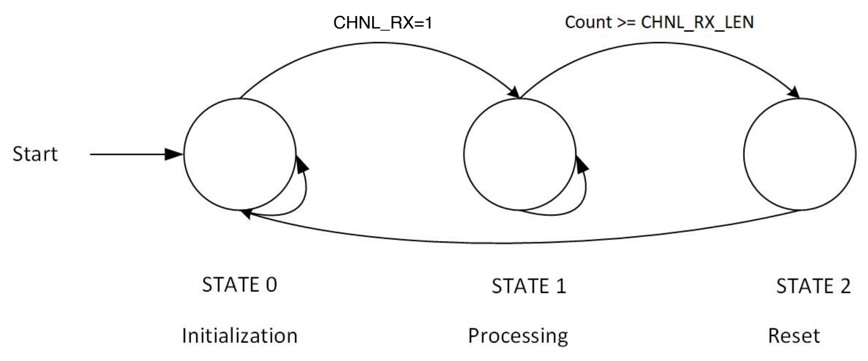


Figure 4.5: The state transition diagram of the RIFFA receiver (RR).

Reset (as shown in Fig. 4.5). In the **Initialization** state, all of RR's controlling signals are initialized and held until the CPU, via *CHNL_RX*, notifies the FPGA that there is the new data stream inflow. Upon such an assertion in *CHNL_RX* the RR is transferred to **Processing** state during which the arrived packet will be received by a temporary-hold register. This register is responsible for transferring the newly accepted packet to the EN. Then EN orderly extracts 4 the event values stored in the packet and provides the corresponding valid signal (Event valid in Fig. 4.4) to the BE. At this point, the **Processing** state stops updating the packet until it is completely digested by the EN. When RR detects the *Packet_Done* signal, its internal packet counter is +1. After all the packets are completely exhausted, the RR state advances to the **Reset** state and resets the packet counter and all controlling signals to wait for the next reception request.

On the **TX** side, once the pointer normalizer finds that there is a pulse output on the pointer ready, it will adjust the 64 pointers sent by the BE. This adjustment means to put every 16 8-bit pointers into a **TX** packet, and finally pass the prepared 4 packets and the corresponding Packets Ready signals to the RIFFA transmitter (RT).

The RT is used to transfer data packets back to the PC main memory for use by the CPU. To do this, first the RT FSM (as shown in Fig. 4.6), in its **Monitor** state, monitors the *Packets_Ready* signal. Once the packets are available, it will set *CHNL_TX* high in this state and move to the next state, **Wait**. After receiving the *CHNL_TX* signal, RIFFA will feed back a *CHNL_TX_ACK* signal indicating that RIFFA is ready to accept a transmission; thus, in conformity with RIFFA's flow control procedure (outlined above) RT sets the *CHNL_TX_VALID* signal high and waits for the *CHNL_TX_DATA_REN* signal in the **Dummy** state (the purpose of this is because the *CHNL_TX_DATA_REN* signal will have a clock delay relative to the *CHNL_TX_DATA_VALID* signal). In the **Send** state, the four packets will be sent completely and go to the **Check** state. It will only respond to the last event process since an additional last state index in the last event needs to be sent. All variables and counters are reset in the **Reset** state and wait for the next data reception.

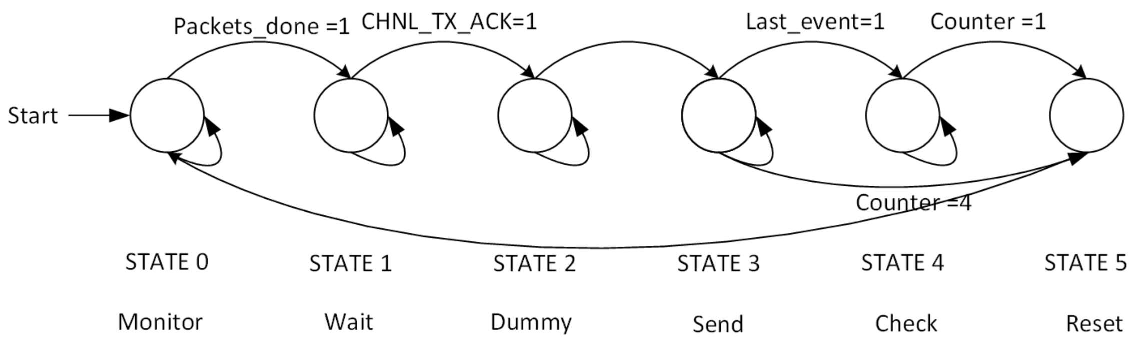


Figure 4.6: The state transition diagram of the RIFFA transmitter (RT).

4.4 Timing Analysis

The design of the hardware circuit cannot only be satisfied with the correct functionality but also must meet timing constraints.

As already noted, the 128-bit RIFFA packet sent from the CPU to the FPGA carries four 32-bit event values. Hence the RIFFA-handling FPGA hardware needs to make extra efforts to organize each event in order to fetch them from the packets in orderly fashion and ensuingly feed them to our IP (as shown in Fig. 4.7). Meanwhile, each valid event value should keep valid for some time until the posterior probability has been updated. In the current design, each event value requires 18 clock cycles by the basecaller (the red line shown in Fig. 4.7 signal) to complete this update. Thus a RIFFA packet should be kept for $18 \cdot 4 = 72$ clock cycles and then pulse on the *CHNL_RX_DATA_REN* to inform the **RX** FIFO update a new packet (the blue line shown in Fig. 4.7).

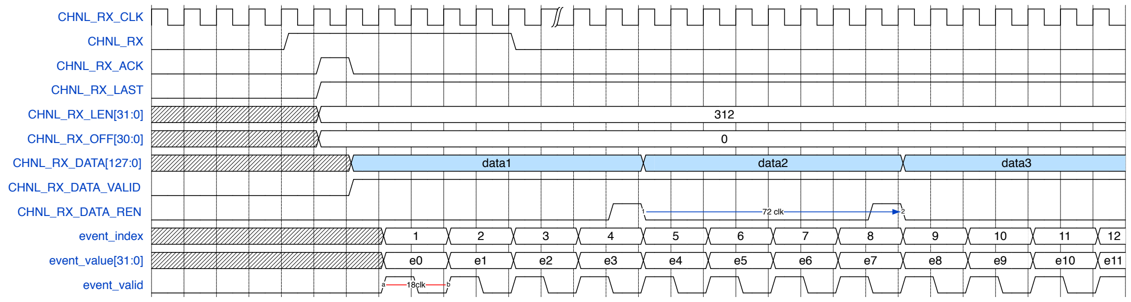


Figure 4.7: Timing waveform showing the organization of RIFFA received data converted to input data for basecaller.

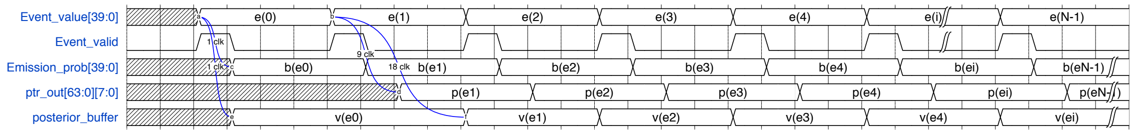

Event_value[39:0]
Event_valid
Emission_prob[39:0]
ptr_out[63:0][7:0]
posterior_buffer
e(0) e(1) e(2) e(3) e(4) e(i) e(N-1)
1 clk
1 clk
9 clk
18 clk
b(e0) b(e1) b(e2) b(e3) b(e4) b(ei) b(eN-1)
p(e1) p(e2) p(e3) p(e4) p(ei) p(eN-1)
v(e0) v(e1) v(e2) v(e3) v(e4) v(ei)

Figure 4.8: Timing waveform showing the basecaller's inner signals. The left-most curves illustrate the latency is 1 clock when the first incoming event is valid until the corresponding emission probability has been calculated. The following blue curves illustrate that pointers are produced 9 clocks after each event input, except the first event for initialization.

In Fig. 4.8, the event valid signal along with the event value both stream into the IP. The emission probability calculation model spends 1 clock cycle at the beginning to work out the corresponding event's (logarithmic) emission probability. Furthermore, the pointers are produced at the 9th tick of the entire 18-clock cycle; the remaining 9 ticks are spent on normalization and update of the posteriors.

The output of the IP has $8 \times 64 = 512$ bits, which is equivalent to four RIFFA packets (128 bit for each). As shown in Fig. 4.9, the *CHNL_TX _DATA* presents these four **TX** packets. *CHNL_TX_DATA_VALID* stays high for four periods until

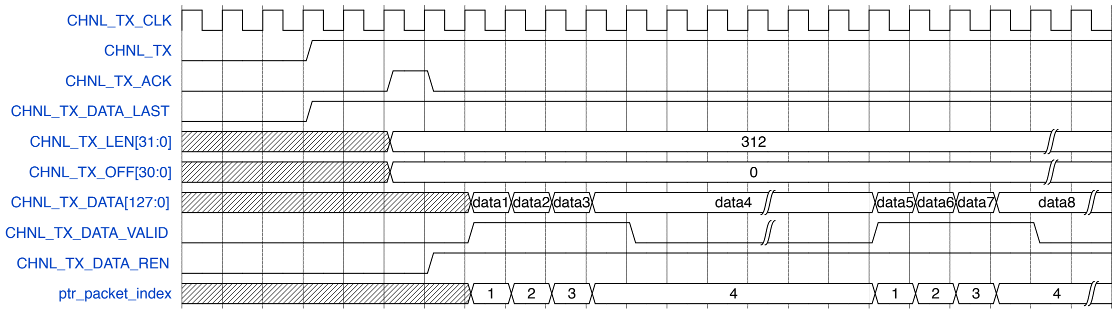


Figure 4.9: The waveform of TX part with the basecaller IP.

all valid pointers are safely sent back to the CPU. Correspondingly, the CPU will receive the packets, arranged in order by the pointer index.

According to the timing analysis, we made the timing overlaps that contribute to improving the processing speed. First, when **RX** keeps sending the new event, the pointers are sending back. Second, the basecaller processing continues finalizing the posteriors for the previous event by normalization when **TX** running its own duty.

4.5 Chapter Summary

The RIFFA framework helps to conceal the full complexity of the PCIe protocol allowing FPGA accelerator designers employing such an interface to focus on implementing their algorithm of interest. Still a number of customizations are needed to enable efficient communications even with the RIFFA framework in place. In this chapter, an explanation of the RIFFA framework and the manner it was used (custom receive and transmit on-FPGA systems) and customized (e.g. enlarging FIFO space, multithreaded communication) to our accelerator needs was given.

5 Measurements and Prospects for 4096-State Basecalling

5.1 Measurements

5.1.1 Resource Utilization

As noted in Chapter 3, the FPGA hardware used to implement the ideas described in the preceding chapters was the Virtex-7 (V7) model from Xilinx [62]. In a Virtex-7 implementation, the PCIe communication and RIFFA DMA blocks consume roughly 20% of the FPGA's available hardware resources. As such, an 8-lane PCIe 2.0 endpoint can be implemented with a peak throughput capability of 25.6 Gbps once on-FPGA implementation overhead is accounted for [63, 64].

After adding the basecaller design, the detailed resource utilization table is listed in Table 5.1. This low resource usage is, at least in part, a result of the effort to minimize DSP slice usage to only the EP block (which requires M multipliers in a fully parallelized implementation) while building state-slice arithmetic out of

Table 5.1: The FPGA Resource Utilization

Resource	Utilizaiton	Available	percentage
LUT	120,745	303600	39.97%
LUTRAM	6,064	130,800	4.62%
FF	102,468	607,200	16.88%
BRAM	73	1,030	7.08%
DSP	1,664	2,800	59.43%
GT	8	28	28.57%
BUFG	11	32	34.38%

the fabric's configurable logic blocks (CLBs). A more explicit and clear utilization diagram is shown in the Fig. 5.1: around half of the resource is taken by the accelerator design.

5.1.2 Power Consumption

When the accelerator is running at its maximum frequency –60MHz, the accelerating system can process about 5M events (shown in Fig. 5.3). In simulation the power consumption at this frequency is estimated to be 5.167 Watts. This number indicates the whole FPGA consumption rather than the used logic gates inside the chip. According to the Xilinx Power Estimator (XPE) [66, 67], the specific power

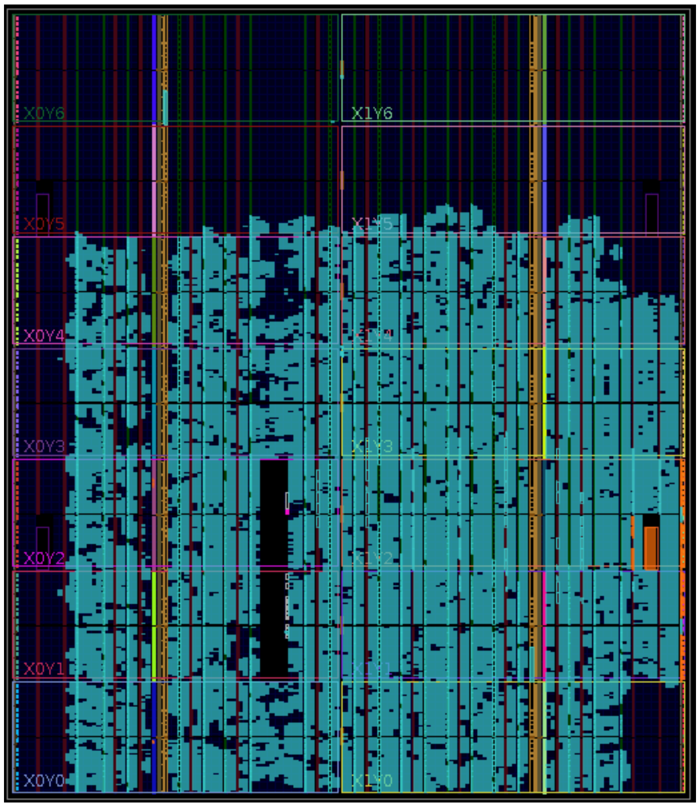
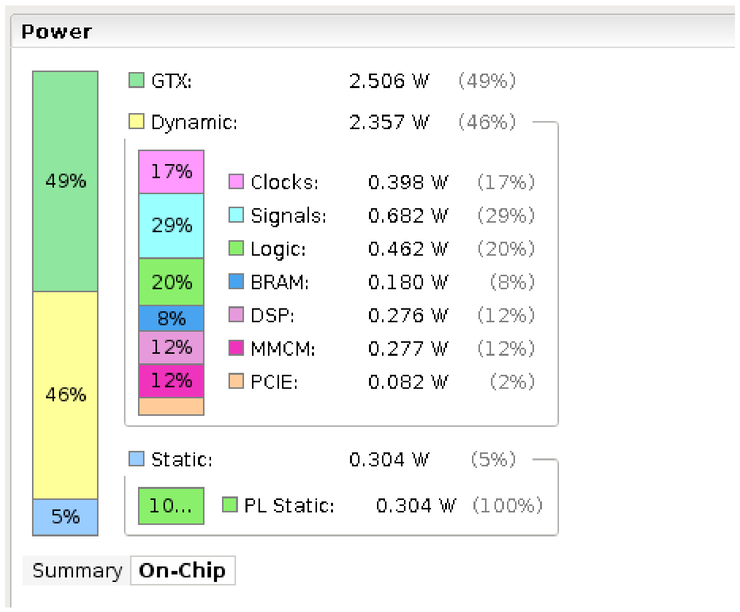


Figure 5.1: The chip layout of the accelerator on the FPGA, the highlighted part is the real assigned hardware resource [65].



Power

🟩 GTX:		2.506 W	(49%)
🟨 Dynamic:		2.357 W	(46%)
	🟪 Clocks:	0.398 W	(17%)
	🟦 Signals:	0.682 W	(29%)
	🟩 Logic:	0.462 W	(20%)
	🟦 BRAM:	0.180 W	(8%)
	🟪 DSP:	0.276 W	(12%)
	🟪 MMCM:	0.277 W	(12%)
	🟧 PCIE:	0.082 W	(2%)
🟦 Static:		0.304 W	(5%)
	🟩 PL Static:	0.304 W	(100%)

Figure 5.2: The specific total power consumption on the FPGA (statistic data is from Xilinx Vivado).

consumption is shown in Fig. 5.2, a screen shot taken from the Vivado FPGA design software suite.

5.2 Performance of the Accelerator Platform

In order to test speed improvement of the accelerator platform against the CPU solution, we set up a CPU centred testing object platform, Platform A. Its configuration includes an Intel Xeon E5-2620 v3 2.4GHz, 32GB of RAM and a 1 TB hard disk. The accelerated platform, Platform B, consists of the same CPU configuration, but with the addition of the PCIe-mediated Virtex-7 FPGA accelerator card. As the input, a sample file containing 7000 artificially pre-generated events is repeatedly presented to each platform 1000 times. The the equivalent of 7 million measured DNA events is presented to both platforms.

On Platform A, a C basecaller runs in multi-thread mode. During operation on the input sample set, the CPU utilization is 100% percent; memory usage is about 10%. The average processing time (over 10 runs, each of 7 million events) is 36.06k bp/s in average. On Platform B, the CPU utilization percentage is 1% and memory usage is 2%. The average speed can reach to 4949k bp/s with an FPGA clock speed set at 60 MHz.

Generally, the accelerator speed is dependent on the FPGA clock frequency. To demonstrate, we also tested Platform B at clock speeds ranging from 10 MHz to 60 MHz. According to the records of the experiment, the processing speed linearly increases with the rise of the frequency (Fig. 5.3).

For more detailed insight, we measured the transmission time on both platforms by employing a timer function in C. Through setting two timers for the two software communications functions: the `fpga_send()` and the `fpga_recv()`, the time of sending all files to be processed plus the overall time during the FPGA worked are measured. The exact processing time on the FPGA is calculated manually, based on the clock period and the 18 clock periods needed to consume each event. Thus, the **RX** and **TX** communication time can be extracted from our measurements.

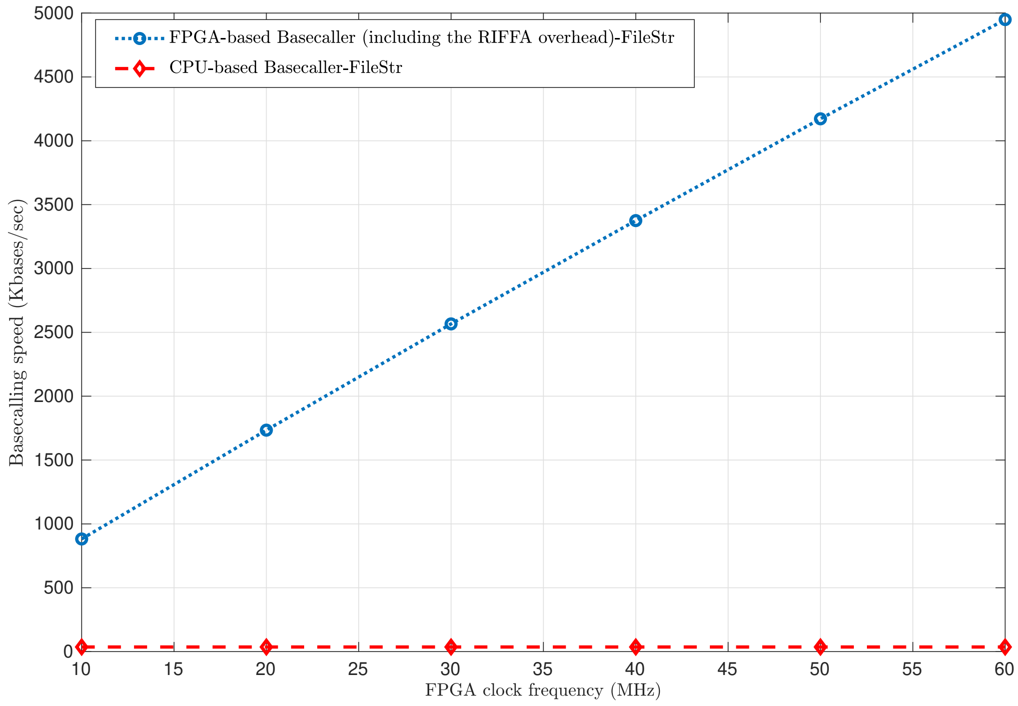
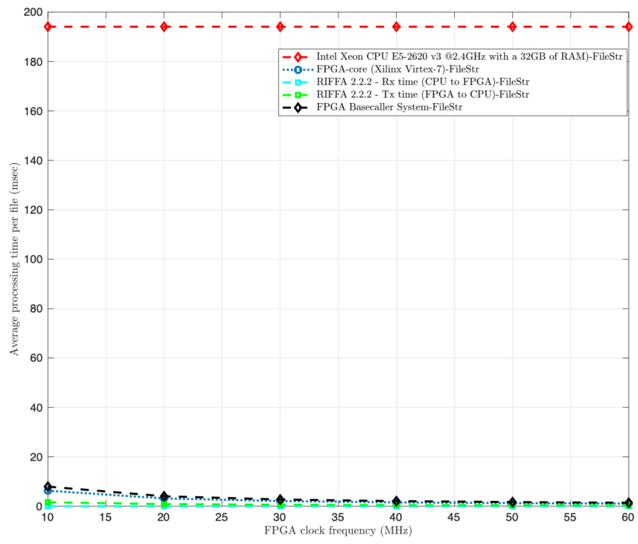


Figure 5.3: Basecalling speed comparison between the FPGA-based accelerator (Platform B) vs CPU centred platform (Platform A).

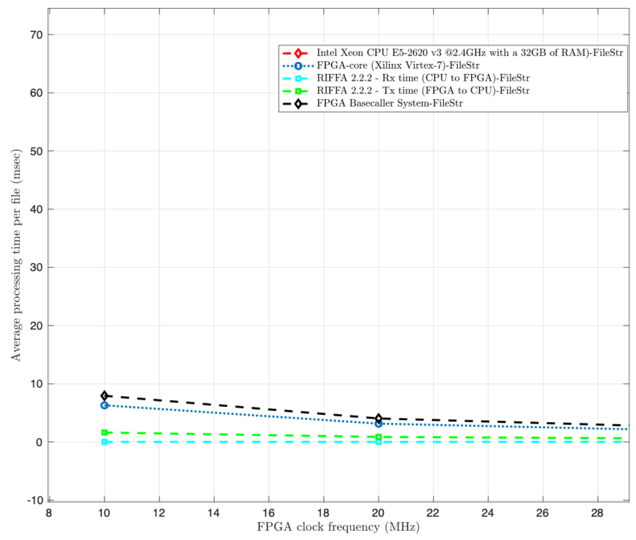
By measurement, the **RX** takes 0.0145 ms while the **TX** takes 0.8723 ms, and the processing time cost is 1.415 ms at 60 MHz. In contrast, the processing time on the CPU is 194.1 ms in a 2.4-GHz system, shown as the red dotted line at the top in Fig. 5.4.

As noted in the previous chapter the RIFFA communications wrapper is driven by the PCIe reference clock signal, a 100-MHz constant setting. While our IP runs on the flexible clock frequency. Hence, no matter what the frequency we set, there is almost no any influence on the communication as demonstrated by the nearly constant communications times over FPGA-core clock settings in Fig. 5.4.

In addition to the processing speed and resource usage mentioned earlier, another concern is to verify the correctness of our basecalling design. Since the basecalling engine's output on the FPGA is a pointer array which indicates the corresponding states for the events, the entire verification is divided into two parts. The first part is the error verification of the algorithm. This process is implemented through statistic of the mismatched number of the result from the basecallering algorithm on the MATLAB after inputting the emulated event vector. As shown in Fig. 5.5, we sent pass 312 events data to the basecaller and found that there are 3 states that do not match. This result is acceptable because it does not have a greater impact on the solved base sequence and satisfies the expected requirements. The second part of the verification is to compare the known error states with the



a)



b)

Figure 5.4: a). illustrates the processing time comparison between platform A and platform B. b). illustrates the partial communication speed cost and processing speed on the platform B.

result obtained on the FPGA. It has been found through experiments that the error state gained from the FPGA is exactly the same as the error states obtained from the first part. This step also proves that the basecalling algorithm runs perfectly on the FPGA.

5.3 4096-state Basealler Prospect

Contemporary nanopore-based DNA applications are better satisfied by larger HMMs, with $M = 4096$ being a more suitable size [68]. The resource overhead remaining in our design does not allow for a direct implementation of such a system, but it does leave enough room for the realization of a system that achieves such scales in a time-multiplexed manner. A particularly suitable arrangement would consist of 256 state slices operating in parallel on the FPGA. With such an implementation the V7's maximum resource consumption would rise about 40% leaving room for the overhead needed to manage time-multiplexing.

Increasing the number of parallel components of course has the potential to improve speed, but it costs even more power. More importantly, based on the present design rate, a 256 parallel state-slice system exhausts the throughput of the Gen2 PCIe system. To increase processing rates without incurring CPU-to-FPGA communication bottlenecks a faster endpoint would be needed. Such capability necessitates an upgrade to, for instance, PCIe Gen4 devices, a reality in contemporary

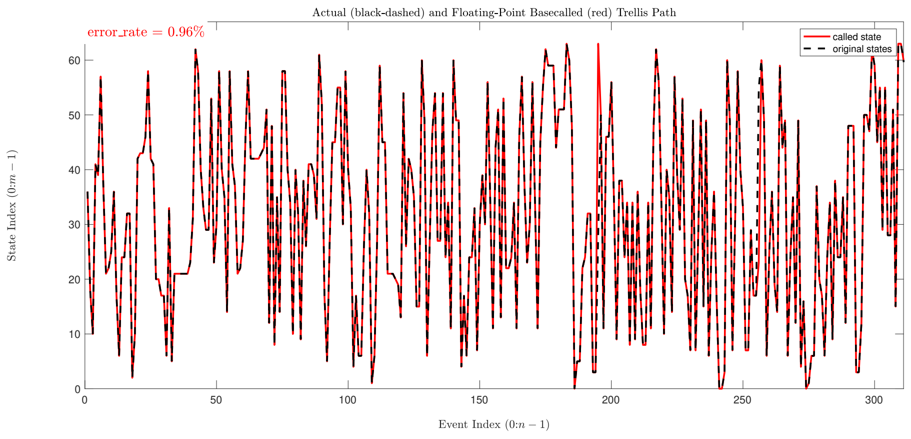


Figure 5.5: The error rate of the result from basecaller in MATLAB.

FPGA offerings (e.g. Xilinx Ultrascale+ devices) [69, 70, 71].

Time-multiplexing also adds to the required on-chip storage burden by virtue of the need to store previous posteriors $v_j(i-1)$ for ready access to state-slices at altering phases of operation. Emission model parameters must also be stored. At an 40-bit width, the total memory burden of a 4096-state implementation is around 4% of the V7's total block RAM (BRAM) memory capacity [72].

The power consumption of the core (plus communications) would rise to roughly 9.5 W while the overall event rate would drop to 0.78 Mevents/s. This still represents a substantial improvement over CPU-only systems.

5.4 Chapter Summary

The resource utilization and power consumption of the 64-states basecalling accelerator are presented in this chapter. Also, the extremely fast processing speed and lower error rate are both presented to illustrate the distinguished performance gap between two platforms (the platform A with accelerator while the platform B without accelerator).

In addition, we discuss about the feasibility of expanding the number of states to 4096 that matches the contemporary nanopore sensor system.

6 Conclusion and Further work

6.1 Conclusion

This thesis described an HMM basecaller partly implemented on an FPGA to accelerate the process of DNA sequencing using nanopore sensors. Although the accelerator system has some flaws such as fewer Mer hardly mapping in the state-of-art nanopore technology for now, it does demonstrate the means by which full-scale basecallers may be constructed at significant speed-up over CPU-only approaches. Compared with the CPU-only solution, the FPGA-accelerated approach speeds up basecalling by 137× with a 17× lower power consumption and a 0.96% error rate. Thanks to the RIFFA framework, which achieves a high-efficiency PCIe-mediated communication between the FPGA and the CPU, up to 4949k bp/s may are handled without any communication-induced bottlenecks.

6.2 Future Work

Further work needs to be done to establish a more robust DNA sequencing acceleration platform capable of working with commercial-grade nanopore-based sequencing data. Improvements should be considered from both the algorithm and hardware design perspective. Key objectives for future work are as follows:

- First, although we implemented a 64-state basecalling accelerator on FPGA current sensor behaviours require implementations capable of processing at least 1,024 states, with 4,096 states being desirable.

- Second, the current algorithm implementation, does not take the model parameter training part into account. The training is still a separate part that needs to be implement. In our project, these parameters are saved into the registers in the hardware. Second, we remove critical parts (inverse Gaussian model parameters) in the process of emission calculation in order to simplify the hardware design. These settings were not deleterious to the data processed in these tests presented, but would likely hurt accuracy on realistic data.

- Third, continuous traceback enables real-time processing capacity of the basecaller. Although, we described the algorithm needed to carry this out we did not implement it on the hardware. The next step is a full transplantation of basecaller, including traceback, onto the FPGA so as to render a pure hardware solution.

- Fourth, the bit-width in the FPGA is defined as 40 bits. The reason for this choice is to fit the data size. However, it takes too many resources on the FPGA. Thus an effort to reduce the processed data bitwidth should be undertaken. This may significantly shrink the resource utilization so as to preserve more available resources for the rest of the design and better enable scaling to larger state counts.

Bibliography

[1] Senior Writer Rachael Rettner. DNA: Definition, Structure & Discovery kernel description. https://www.livescience.com/37247-dna.html.

[2] Sara Goodwin, John D McPherson, and W Richard McCombie. Coming of age: ten years of next-generation sequencing technologies. *Nature Reviews Genetics*, 17(6):333, 2016.

[3] S.Nicklen A.R. F.Sanger. Dna sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences USA*, 74(12).

[4] Joseph F Boland, Charles C Chung, David Roberson, Jason Mitchell, Xijun Zhang, Kate M Im, Ji He, Stephen J Chanock, Meredith Yeager, and Michael Dean. The new sequencer on the block: comparison of life technology's proton sequencer to an illumina hiseq for whole-exome sequencing. *Human genetics*, 132(10):1153–1163, 2013.

[5] J Gregory Caporaso, Christian L Lauber, William A Walters, Donna Berg-

Lyons, James Huntley, Noah Fierer, Sarah M Owens, Jason Betley, Louise Fraser, Markus Bauer, et al. Ultra-high-throughput microbial community analysis on the illumina hiseq and miseq platforms. *The ISME journal*, 6(8):1621, 2012.

[6] Lauber Christian L Walters William A Berg-Lyons Donna Huntley James Caporaso, J.Gregory. Ultra-high-throughput microbial community analysis on the illumina hiseq and miseq platforms. *The Isme Journal*, March 2012.

[7] Stephane Le Crom. Commercially-available high throughput sequencers timeline. https://doi.org/10.6084/m9.figshare.5327419.v1. 2017.

[8] Artem E Men, Peter Wilson, Kirby Siemering, and Susan Forrest. Sanger dna sequencing. *Next-Generation Genome Sequencing: Towards Personalized Medicine*, pages 1–11, 2008.

[9] Milanova D. Kerby M.B. Snyder M.P.& Barron-A.E. Niedringhaus, T.P. Landscape of next-generation sequencing technologies. pages 4327–4341, 2011.

[10] Jay Shendure, Shankar Balasubramanian, George M Church, Walter Gilbert, Jane Rogers, Jeffery A Schloss, and Robert H Waterston. Dna sequencing at 40: past, present and future. *Nature*, 550(7676):345, 2017.

[11] Mehdi Kchouk, Jean-François Gibrat, and Mourad Elloumi. Generations of

sequencing technologies: From first to next generation. *Biology and Medicine*, 9(3), 2017.

[12] Cliff Meldrum, Maria A Doyle, and Richard W Tothill. Next-generation sequencing for cancer diagnostics: a practical perspective. *The Clinical Biochemist Reviews*, 32(4):177, 2011.

[13] David Deamer, Mark Akeson, and Daniel Branton. Three decades of nanopore sequencing. *Nature biotechnology*, 34(5):518, 2016.

[14] Clive G Brown and James Clarke. Nanopore development at oxford nanopore. *Nature biotechnology*, 34(8):810, 2016.

[15] Alexander S Mikheyev and Mandy MY Tin. A first look at the oxford nanopore minion sequencer. *Molecular ecology resources*, 14(6):1097–1102, 2014.

[16] James Clarke, Hai-Chen Wu, Lakmal Jayasinghe, Alpesh Patel, Stuart Reid, and Hagan Bayley. Continuous base identification for single-molecule nanopore dna sequencing. *Nature nanotechnology*, 4(4):265, 2009.

[17] Carl W Fuller, Shiv Kumar, Mintu Porel, Minchen Chien, Arek Bibillo, P Benjamin Stranges, Michael Dorwart, Chuanjuan Tao, Zengmin Li, Wenjing Guo, et al. Real-time single-molecule electronic dna sequencing by synthesis using

polymer-tagged nucleotides on a nanopore array. *Proceedings of the National Academy of Sciences*, 113(19):5233–5238, 2016.

[18] Bala Murali Venkatesan and Rashid Bashir. Nanopore sensors for nucleic acid analysis. *Nature nanotechnology*, 6(10):615–624, September 2011.

[19] Miten Jain, Hugh E Olsen, Benedict Paten, and Mark Akeson. The oxford nanopore minion: delivery of nanopore sequencing to the genomics community. *Genome biology*, 17(1):239, 2016.

[20] Ivan Sović, Mile Šikić, Andreas Wilm, Shannon Nicole Fenlon, Swaine Chen, and Niranjan Nagarajan. Fast and sensitive mapping of nanopore sequencing reads with graphmap. *Nature communications*, 7:11307, 2016.

[21] Miten Jain, Ian T Fiddes, Karen H Miga, Hugh E Olsen, Benedict Paten, and Mark Akeson. Improved data analysis for the minion nanopore sequencer. *Nature methods*, 12(4):351, 2015.

[22] Daniel Branton, David W Deamer, Andre Marziali, Hagan Bayley, Steven A Benner, Thomas Butler, Massimiliano Di Ventra, Slaven Garaj, Andrew Hibbs, Xiaohua Huang, et al. The potential and challenges of nanopore sequencing. In *Nanoscience And Technology: A Collection of Reviews from Nature Journals*, pages 261–268. World Scientific, 2010.

[23] M Mitchell Waldrop. The chips are down for moore's law. *Nature News*, 530(7589):144, 2016.

[24] Daniel Golparian, Valentina Dona, Leonor Sanchez-Buso, Sunniva Foerster, Simon Harris, Andrea Endimiani, Nicola Low, and Magnus Unemo. Oxford Nanopore MinION genome sequencer: performance characteristics, optimised analysis workflow, phylogenetic analysis and prediction of antimicrobial resistance in Neisseria gonorrhoeae. *bioRxiv*, page 349316.

[25] Hengyun Lu, Francesca Giordano, and Zemin Ning. Oxford nanopore minion sequencing and genome assembly. *Genomics, proteomics & bioinformatics*, 14(5):265–279, 2016.

[26] Jean-Pierre Deschamps, Gery JA Bioul, and Gustavo D Sutter. *Synthesis of arithmetic circuits: FPGA, ASIC and embedded systems*. John Wiley & Sons, 2006.

[27] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215, 2007.

[28] Srinidhi Kestur, John D Davis, and Oliver Williams. Blas comparison on fpga, cpu and gpu. In *VLSI (ISVLSI), 2010 IEEE computer society annual symposium on*, pages 288–293. IEEE, 2010.

[29] AJ Storm, JH Chen, XS Ling, HW Zandbergen, and C Dekker. Fabrication of solid-state nanopores with single-nanometre precision. *Nature materials*, 2(8):537, 2003.

[30] Robinson Mittmann Sebastian Magierowski Ebrahim Ghafar-Zadeh Zhong-Pan Wu, Karim Hammad and Xiaoyong Zhong. Fpga-based dna basecalling hardware acceleration. 10 2018.

[31] Mathias C Walter, Katrin Zwirglmaier, Philipp Vette, Scott A Holowachuk, Kilian Stoecker, Gelimer H Genzel, and Markus H Antwerpen. Minion as part of a biomedical rapidly deployable laboratory. *Journal of biotechnology*, 250:16–22, 2017.

[32] Christian Ledergerber and Christophe Dessimoz. Base-calling for next-generation sequencing platforms. *Briefings in bioinformatics*, 12(5):489–497, 2011.

[33] Kuo-ching Liang, Xiaodong Wang, and Dimitris Anastassiou. Bayesian basecalling for dna sequence analysis using hidden markov models. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 4(3):430–440, 2007.

[34] Rastislav Rabatin, Broňa Brejová, and Tomáš Vinař. Using sequence en-

sembles for seeding alignments of minion sequencing data. *arXiv preprint arXiv:1606.08719*, 2016.

[35] Petros Boufounos, Sameh El-Difrawy, and Dan Ehrlich. Basecalling using hidden markov models. *Journal of the Franklin Institute*, 341(1-2):23–36, 2004.

[36] Winston Timp, Jeffrey Comer, and Aleksei Aksimentiev. Dna base-calling from a nanopore using a viterbi algorithm. *Biophysical journal*, 102(10):L37–L39, May 2012.

[37] Yiyun Huang, Sebastian Magierowski, Ebrahim Ghafar-Zadeh, and Chengjie Wang. High-speed event detector for embedded nanopore bio-systems. In *Engineering in Medicine and Biology Society (EMBC), 2015 37th Annual International Conference of the IEEE*, pages 2179–2182. IEEE, 2015.

[38] Yiyun Huang, Sebastian Magierowski, Ebrahim Ghafar-Zadeh, and Chengjie Wang. A high-speed real-time nanopore signal detector. In *Computational Intelligence in Bioinformatics and Computational Biology (CIBCB), 2015 IEEE Conference on*, pages 1–8. IEEE, 2015.

[39] Sara Goodwin, James Gurtowski, Scott Ethe-Sayers, Panchajanya Deshpande, Michael C Schatz, and W Richard McCombie. Oxford nanopore sequencing, hybrid error correction, and de novo assembly of a eukaryotic genome. *Genome research*, 2015.

[40] David Brady, Marko Kocic, Arthur W Miller, and Barry L Karger. A maximum-likelihood base caller for dna sequencing. *IEEE transactions on biomedical engineering*, 47(9):1271–1280, 2000.

[41] Alexander V Lukashin and Mark Borodovsky. Genemark. hmm: new solutions for gene finding. *Nucleic acids research*, 26(4):1107–1115, 1998.

[42] Joachim Hagenauer and Peter Hoeher. A viterbi algorithm with soft-decision outputs and its applications. In *Global Telecommunications Conference and Exhibition'Communications Technology for the 1990s and Beyond'(GLOBECOM), 1989. IEEE*, pages 1680–1686. IEEE, 1989.

[43] Matei David, Lewis Jonathan Dursi, Delia Yao, Paul C Boutros, and Jared T Simpson. Nanocall: An Open Source Basecaller for Oxford Nanopore Sequencing Data. *bioRxiv*, page 046086, 03 2016.

[44] P Brown, J Spohrer, P Hochschild, and J Baker. Partial traceback and dynamic programming. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'82.*, volume 7, pages 1629–1632. IEEE, 1982.

[45] Matt Jacbsen Dustin Richmond. Riffa 2.2.2 document. Technical report, University of California San Diego, 8 2016.

[46] Sebastian Magierowski. Nanopore basecaller analysis: Computations, comparisons, costs. *Tech. Rep. York University*, 2016.

[47] SiS Sandra. benchmark suite, sisoftware. https://www.sisoftware.co.uk/, 2018.

[48] Bob Mittmann and Sebastian Magierowski. βcall - c language basecalling for nanopore systems. pages 1–4, 10 2016.

[49] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2):34–46, 2016.

[50] Joseph Wang. Survey and summary: from dna biosensors to gene chips. *Nucleic acids research*, 28(16):3011–3016, 2000.

[51] Adam T Woolley, George F Sensabaugh, and Richard A Mathies. High-speed dna genotyping using microfabricated capillary array electrophoresis chips. *Analytical Chemistry*, 69(11):2181–2186, 1997.

[52] Xilinx Inc. Vc707 evaluation board for the virtex-7 fpga. User Guide, 8 2016. UG885.

[53] Xilinx Inc. Getting started with the virtex-7 fpga vc707 evaluation kit. User Guide, 10 2015. UG848.

[54] Xilinx Inc. 7 series dsp48e1 slice. User Guide, 3 2018. UG479.

[55] Jasmin Ajanovic. Pci express* (pcie*) 3.0 accelerator features. *Intel Corporation*, 10, 2008.

[56] Mike Rose. Fpga pcie bandwidth. *University of California San Diego*, 7, 2010.

[57] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. A quantitative analysis on microarchitectures of modern cpu-fpga platforms. In *Proceedings of the 53rd Annual Design Automation Conference*, page 109. ACM, 2016.

[58] Matthew Hogains Ryan Kastner Matthew Jacobsen, Dustin Richmond. RIFFA 2.1. *ACM Transactions on Reconfigurable Technology and Systems*, 8(4):1 23, 10 2015.

[59] Yann Thoma, Alberto Dassatti, and Daniel Molla. Fpga 2: An open source framework for fpga-gpu pcie communication. In *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*, pages 1–6. IEEE, 2013.

[60] Hossein Kavianipour, Steffen Muschter, and Christian Bohm. High performance fpga-based dma interface for pcie. In *Real Time Conference (RT), 2012 18th IEEE-NPSS*, pages 1–3. IEEE, 2012.

[61] Thomas B Preußer and Rainer G Spallek. Ready pcie data streaming solutions for fpgas. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–4. IEEE, 2014.

[62] Xilinx Virtex. Fpga family, 7.

[63] Xilinx Inc. 7 series fpgas integrated block forpci express v3.3. LogiCORE IP Product Guide, 5 2018. PG054.

[64] Jian Gong, Tao Wang, Jiahua Chen, Haoyang Wu, Fan Ye, Songwu Lu, and Jason Cong. An efficient and flexible host-fpga pcie communication library. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–6. IEEE, 2014.

[65] Xilinx Inc. Vivado design suite. User Guide, 6 2018. UG904.

[66] Xilinx Inc. Xilinx power estimator. User Guide, 6 2017. UG440.

[67] Aditya Kushwaha, Gaurav Verma, and Varun Kumar Kakar. Thermal analysis and modelling of power consumption for fpgas. In *2018 International Conference on Advances in Computing and Communication Engineering (ICACCE)*, pages 419–422. IEEE, 2018.

[68] Chengjie Wang, Zhongpan Wu, Ebrahim Ghafar-Zadeh, and Sebastian Magierowski. Embedded cmos bioinformatics for nanopore sequencers. In

Electrical and Computer Engineering (CCECE), 2017 IEEE 30th Canadian Conference on, pages 1–4. IEEE, 2017.

[69] Vamsi Boppana, Sagheer Ahmad, Ilya Ganusov, Vinod Kathail, Vidya Rajagopalan, and Ralph Wittig. Ultrascale+ mpsoc and fpga families. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*, pages 1–37. IEEE, 2015.

[70] Stephanie Rupprich. The xilinx ultrascale architecture. 2014.

[71] Nick Mehta. Xilinx ultrascale architecture for high-performance, smarter systems. *Xilinx White Paper WP434*, 2013.

[72] Kirk Saban. Xilinx stacked silicon interconnect technology delivers breakthrough fpga capacity, bandwidth, and power efficiency. *Xilinx, White Paper*, 1:1–10, 2011.