Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Sampling as an approach to sequencing MinION data

Bakalárska práca

2017
Matúš Zeleňák

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Sampling as an approach to sequencing MinION data

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	Mgr. Tomáš Vinař PhD.

Bratislava, 2017
Matúš Zeleňák

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname: Matúš Zeleňák
Study programme: Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study: Computer Science, Informatics
Type of Thesis: Bachelor´s thesis
Language of Thesis: English
Secondary language: Slovak

Title: Sampling from MinION Reads

Aim: The MinION DNA sequencing platform produces long reads with high error rates. One of the reasons for high error rates is that electrical signals produced by the sequencing machine need to be first translated into DNA sequences by a process called base calling, and this process is error prone. An alternative way of interpreting these signals is to generate multiple samples from a posterior sequence distributions defined by a hidden Markov model (HMM) representing the properties of the sequencing process. The goal of this thesis is to speed up this sampling by employing GPUs.

Supervisor: doc. Mgr. Tomáš Vinař, PhD.
Department: FMFI.KAI - Department of Applied Informatics
Head of department: prof. Ing. Igor Farkaš, Dr.

Assigned: 27.10.2016

Approved: 31.10.2016 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.... Student

.... Supervisor

Acknowledgement: I would like to thank Mgr. Tomáš Vinař,PhD. and Mgr. Bronislava Brejová, PhD. for their goodwill, guidance and advice. Moreover I am grateful to Mgr. Vladimír Boža for technical help. Special thanks to my family and girlfriend for supporting me. Additional thanks to Shia LaBeouf for providing motivation.

Abstrakt

MinION je platforma sekvenovania DNA, ktorá produkuje dlhé čítania s relatívne vysokým stupňom chybovosti spôsobeným prekladom meraných napätí zo sekvenátora do báz DNA, ktorý je nepresný. Je známe, že predspracovanie týchto meraní pomocou skrytých Markovovových modelov pred prekladom pomáha zmierniť chybovosť. V našej práci sme použili skrytý Markovov model na vzorkovanie z posteriórnej pravdepodobnosti pričom hlavným zameraním bolo zrýchlenie vzorkovacieho algoritmu použitím paralelizmu, ktorý ponúkaju grafické karty. Konkrétne sme adoptovali platformu CUDA na paralelizáciu algoritmov v jazyku C a použili ňou ponúknuté programovacie paradigmy na zrýchlenie kľúčových častí nášho programu. Následne sme porovnali časy vykonávania akcelerovaného algoritmu s verziou, ktorá CUDU nevyužíva a beží na CPU. Výsledky jasne ukazujú, že pre veľkosti vstupov vyskytujúce sa v praxi ponúka náš zrýchlený algoritmus značné zrýchlenie, často o niekoľko rádov v porovnaní s neparalelnou verziou bežiacou na CPU.

Kľúčové slová: MinION, skrytý Markovov model, vzorkovanie, DNA, CUDA

Abstract

MinION DNA sequencing platform produces long reads with relatively high error rates caused by a noisy translation of current measurements into the DNA bases representation. It is known, that preprocessing the current measurements with Hidden Markov models before the translation can help reduce the error. We use the Hidden Markov model to sample from a posterior probability and the main focus of our work is to accelerate the sampling algorithm by using the parallelization potential of graphic cards. Specifically we adopted the CUDA platform used for parallelization of C algorithms and implemented the paradigms it provides into the key parts of our software. Subsequently we compared the running times of both the accelerated algorithm with a version that does not use CUDA and runs on CPU. The results show conclusively that for inputs occurring in real life applications our algorithm offer a significant speedup, ofter in the order of several magnitudes compared with a non-parallel CPU version.

Keywords: MinION, Hidden Markov model, sampling, DNA, CUDA

Contents

Introduction **1**

1 Background **2**

1.1 DNA Sequencing . . . 2
1.2 Nanopore sequencing and MinION platform . . . 3
1.3 Hidden Markov Model . . . 4
1.4 Relationship of the MinION data to HMM . . . 5

2 Sampling **8**

2.1 Theory . . . 8
2.1.1 Motivation for sampling . . . 8
2.1.2 Forward Algorithm . . . 9
2.1.3 Stochaistic tracebacking of sample . . . 10
2.2 Algorithm implementation . . . 10
2.2.1 Sample path decoding . . . 11
2.2.2 Arithmetics in log scale . . . 12
2.2.3 Time and space complexity analysis . . . 12
2.2.4 Used libraries . . . 13
2.2.5 Abandoned attempt at implementation . . . 13

3 Parallelization through GPGPU **14**

3.1 Introduction to GPGPU and example cases . . . 14
3.1.1 GPGPU applications . . . 15
3.2 Used techniques . . . 15
3.3 Parallel implementation of sampling algorithm . . . 16
3.3.1 Parallelization bottleneck identification . . . 16
3.3.2 Theoretical speedup estimation . . . 18

4 Experiment design and results **19**

4.1 Input data . . . 19
4.2 Testing method . . . 19

4.3 Running time comparison 21
4.3.1 Theoretical vs. practical speedup 25

5 Conclusion **27**
5.1 Future work 28

Appendix A **30**

List of Figures

1.1 Nanopore with threaded DNA, table with current measurements [3] . . 3
1.2 Graph representation of HMM 4
1.3 Raw signal from MinION and its segmentation to events [14] 5
1.4 *diff* function example 6
1.5 Example of transition types 7

2.1 Example comparison of Viterbi path and samples 9
2.2 Path decoding example 11

3.1 Simple kernel example 15
3.2 Kernel usage example code 16
3.3 Parallel forward matrix calculation pseudocode 17
3.4 Parallel sampling pseudocode 17
3.5 Parallel decoding pseudocode 17

4.1 Running time of non-paralell calibration algorithm 21
4.2 Running time of forward matrix calculation for varying data size 22
4.3 Running time of sampling phase for varying data size 22
4.4 Running time of sampling phase for variable number of samples 23
4.5 Running time of forward matrix phase for variable k-mer size 23
4.6 Running time of sampling phase for variable k-mer size 24
4.7 Running time for forward matrix phase for variable skip size 24
4.8 Running time for sampling phase for variable skip size 25
4.9 Running time for decoding phase 25

List of Tables

4.1 Parameter defaults 20

Introduction

MinION is a third generation DNA sequencing platform capable of producing long reads. Sequencing is performed by threading a DNA strand through a nanopore and measuring current alterations that the passing DNA fragment induces. Based on these current measurements we are able to determine the nucleotide string of the sequenced DNA strand. Unfortunately, this process of measurement translation is noisy and results in relatively high error rate. It is known that these errors can be mitigated to a degree by first processing the measurement data with Hidden Markov models. For instance, there already exist solutions[16] which use Hidden Markov models for determining the most likely DNA sequence for a set of measurements. In our work, we will focus on sampling which is a process of generating a set of DNA sequences that are close to the most likely one in terms of their probabilities and thus maintain a biological significance. The main focus of our work is on acceleration of the already existing sampling algorithm. We propose parallelization using Nvidia CUDA API for GPUs as a viable way of significantly increasing the speed at which samples can be produced. In the first chapter we provide the reader with theoretical background of our thesis and introduce the model we use for the remainder of our work. Second chapter describes the motivation for sampling and implementation of the algorithm on which we base our software. Third chapter revolves around the potential of graphic cards for scientific use and applies the CUDA paradigms to the sampling algorithm. In the fourth and last chapter we describe the design and results of experiments that rate the worthiness of parallelization for our task.

Chapter 1

Background

This chapter should introduce the reader to relevant topics that will be used throughout the thesis and should serve as a reference point for basic definitions.

1.1 DNA Sequencing

Organisms have most of their working principles encoded into a deoxyribonucleic acid (abbreviated DNA). This molecular structure made of two complementary strands of nucleotides contains information necessary for growth, functioning and reproduction of the carrier organism [1].

From the computer science standpoint, a single strand of DNA can be viewed as a string of finite length over a four symbol alphabet. The alphabet symbols represent the DNA nucleotides (**bases**) which are the building blocks of nucleic acid and are labeled A, C, G, T for adenine, cytosine, thymine and guanine respectively.

Given the importance of DNA, it is understandable that we attempt to retrieve its structure from individual organisms in a process called DNA sequencing [2]. The result of a sequencing run is called **read** – a set of measurements which can be translated into a string representation of the aforementioned nucleotides. In most sequencing technologies there is a limit to the length of a read that can be obtained in a single run. This limit is typically much lower than the length of the a genome we are sequencing and thus individual reads need to be later reassembled using bioinformatics' methods[24].

Throughout history, the sequencing approaches and technologies evolved and can be separated into so-called generations: The first generation consists of modifications of Sanger sequencers which are characteristic by sequencing runs of about 1kbp (kilobase pair), high per-base accuracy of up to 99.999% and a cost in the order of $0.5 per kbp

[23].
The second, or 'Next-generation sequencers' use a wider variety of technologies and compared to the first generation offer shorter reads in the order of tens of base pairs with a higher error rate of about 1-1.5%. The cost is around $2 per megabase pair[23]. The most recent third generation devices are capable of producing reads of lenghts in the order of tens of kilobasepairs per run at about half the cost for second generation at the expense of higher error rate in the ballpark of 13% [22].

1.2 Nanopore sequencing and MinION platform

Definition 1.2.1. ***k*-mer** is a continuous sequence of k nucleotide bases.

Definition 1.2.2. The process of translating sequencing reads into a representation of a corresponding DNA sequence is named **base calling**.

Recently, a third generation platform called **MinION** was released by *Oxford Nanopore technologies.* The core of the platform is a phone-sized device which is connected to a computer via USB. Sequencing is performed by threading a single strand of DNA through a protein nanopore. The nanopore is embedded in a membrane made of synthetic polymers to which an electric current is applied. There are hundreds of such nanopores in one device, each of which generates a stream of data [19].

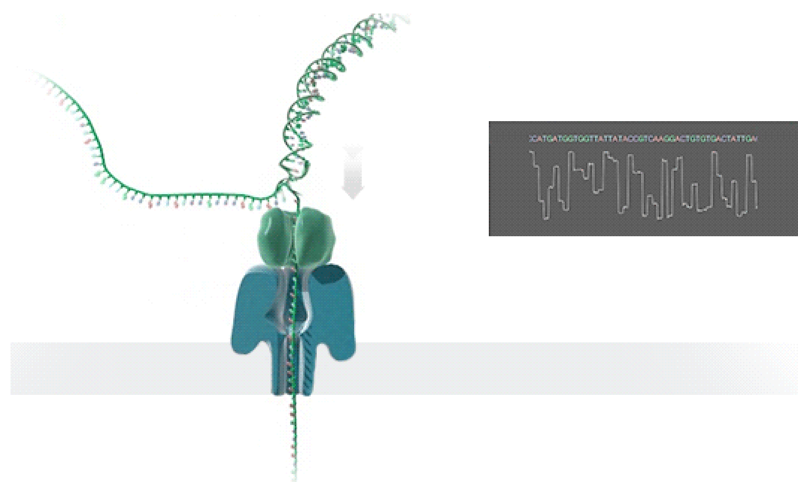

Figure 1.1: Nanopore with threaded DNA, table with current measurements [3]

As a part of the DNA strand passes through the nanopore it induces a characteristic disruptions of the current. In an ideal conditions, the speed at which the strand passes through the nanopore would be constant and there would be a deterministic way to map the disruptions to k-mers that caused them. That is not the case and even the latest version of MinION (the R9.4 of October 2016) has up to **8% error rate**.[18]

1.3 Hidden Markov Model

One way to approach the problem of base calling on MinION data is to design a probabilistic model that characterizes the basic properties of the technology. To this end, we will use a Hidden Markov Model (HMM) [21].

Hidden Markov model is a machine learning concept that has use in many areas: speech and handwriting recognition, image processing and bioinformatics to name a few.

Definition 1.3.1 (Hidden Markov model)**.** Let $H = (Q, E, e, t, s)$ where Q is a non-empty finite set of states, E is a an uncountably infinite set of emissions, $e : Q \times E \to [0, 1]$ is an emission probability function and $t : Q \times Q \to [0, 1]$ is a transition probability function and s is an initial state probability function. H is a Hidden Markov model when the following conditions hold true:

1. $\forall u \in Q : \sum_{v \in Q} t(u, v) = 1$
2. $\forall u \in Q : \int_{x \in E} e(u, x)dx = 1$
3. $\sum_{q \in Q} s(q) = 1$

This model is structurally similar to a finite automaton and can be visually represented as a directed graph. In this graph the states from Q would form vertices and arrows would mark transitions between states u, v that satisfy $t(u, v) > 0$. An example can be seen in Figure 1.3

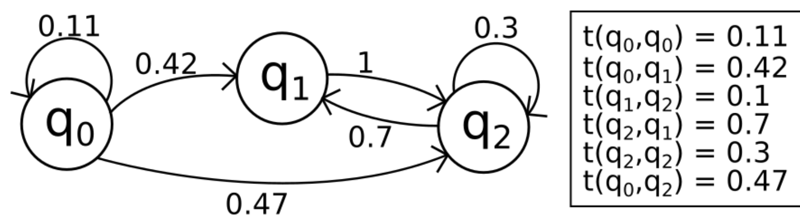

Figure 1.2: Graph representation of HMM

The HMM we use is a generative model. It is used to generate two sequences - a sequence of states and a sequence of observations.

Starting from an arbitrary state q_0 with probability $s(q_0)$ we emit a random observation $o \in E$ with a probability defined by $e(q_0, o)$. Next, we transition to some random state $q_i \in Q$ with a probability defined by $t(q_0, q_i)$. Once in the state q_i we again emit an observation, this time with probability $e(q_i, o)$ and again transition to next state. This chain of transitions can have arbitrary length, even be infinite. For

the sake of our work we are only interested in transitional state paths of finite length.

In this way, HMM defines the joint probability over the pairs $P(o \mid s)$ where o is the sequence of observations and s is a description of the state path the HMM took in order to emit o. This probability can be calculated as

$$P(o \mid s) = (\prod_{i=1}^{n} t(s_{i-1}, s_i)) \cdot (\prod_{i=1}^{n} e(s_i, o_i)) \tag{1.1}$$

where $o = o_1, o_2, \ldots, o_n$ is the sequence of observations and $s = s_0, s_1, s_2, \ldots, s_m$ is the sequence of states.

Definition 1.3.2. Indegree of state $q \in Q$ from HMM (Q,E,e,t,s) is the value

$$I_q = |\{q_f \mid t(q_f, q) > 0, q_f \in Q\}|$$

Definition 1.3.3. Maximal indegree of the HMM (Q,E,e,t,s) is

$$I_H = max(\{I_q \mid q \in Q\})$$

where I_q is indegree of state q.

1.4 Relationship of the MinION data to HMM

Definition 1.4.1. Event is a result of clustering of the raw current measurement data originating from a single nanopore. Event is characterized by its **mean**, **standard deviation** and duration [19].

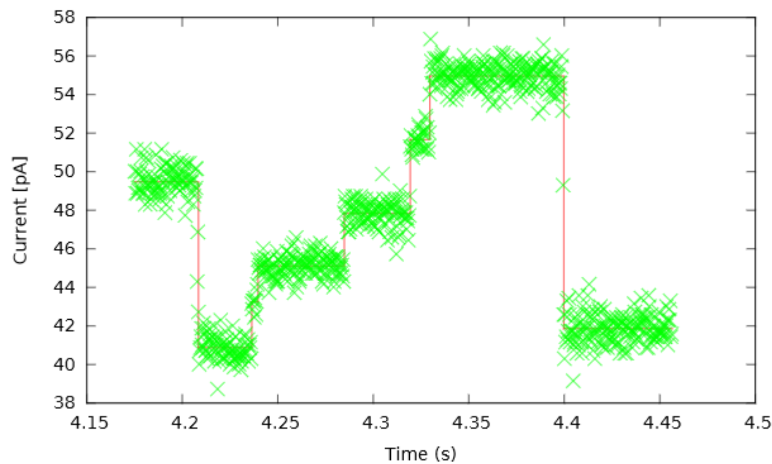

Figure 1.3: Raw signal from MinION and its segmentation to events [14]

In an ideal situation, each consecutive event should correspond to a different DNA context (different part of the sequenced DNA) and consecutive events should represent

DNA contexts which differ in only one nucleotide. In practice the process of segmentation of current measurements into a sequence of events is noisy and can produce consecutive events that represent the same DNA context (stays) or there can be a shift of more than one nucleotide between the consecutive events (skips).
We will use a specific form of an HMM to alleviate this noisiness. Consider a HMM H such that:

- $Q = \{x | x \in \{A, C, G, T\}^k, k \in \mathbb{N}\}$
- $E \subset \mathbb{R}$
- $s(q) = \frac{1}{|Q|} \quad \forall q \in Q$

As can be seen, each state of H represents a different DNA k-mer.

For future use let us create a helper function *diff* operating on pairs of states. The value of $diff(q_u, q_v)$ is $k - l$ where l is the length of the longest suffix of q_u which matches the prefix of q_v of length l and k is the length of a k-mer.

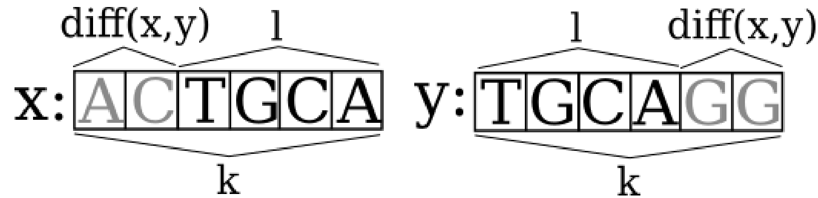

Figure 1.4: *diff* function example

Suppose we have two states q_u and q_v. We will call a transition between two states q_u and q_v that satisfies $diff(q_u, q_v) = 2$ a **skip** transition of one base. We can see an example of such transition as T_3 in Figure 1.4. We define the probability of such transition occurring as $prob_{skip}$. In a similar way we can define a skip transition of up to $k - 1$ bases for states that have $diff \in [2, k]$.
For $diff(q_u, q_v) = 0$ we call the transition a **stay** transition. Figure 1.4 shows this transition as T_2. Again, we define a constant $prob_{stay}$ marking the probability of H making a stay transition.

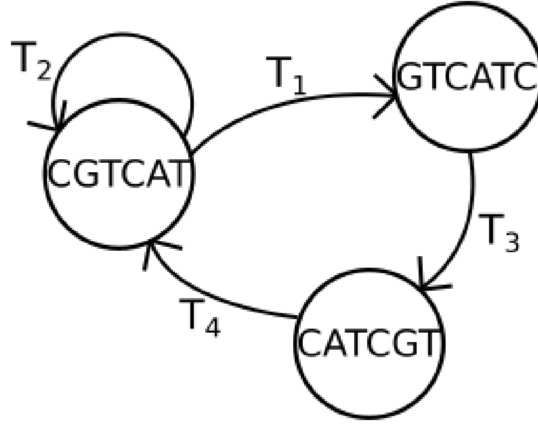

Figure 1.5: Example of transition types

Additionally we have a variable m defining the maximum allowed number of bases H can skip in transition.

The transitional probability $t(q_m, q_n)$ for a value $m \in [1, k-1]$ can now be derived from the following conditions:

- for $\mathit{diff}(q_m, q_n) = 0 : t(q_m, q_n) = prob_{stay}$
- for $\mathit{diff}(q_m, q_n) = s \in [2, m] : t(q_m, q_n) = \frac{(prob_{skip})^{s-1}}{|\{q_x \mid diff(q_m, q_x) = s\}|}$
- for $\mathit{diff}(q_m, q_n) = 1 : t(q_m, q_n) = \frac{1 - prob_{stay} - \sum_{q_o \in \{q \mid \mathit{diff}(q_m, q) > 1\}} t(q_m, q_o)}{|\{q_x \mid diff(q_m, q_x) = 1\}|}$
- for $\mathit{diff}(q_m, q_n) > m : t(q_m, q_n) = 0$

For a given observation x the probability of a state emitting x is calculated from a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ where both μ and σ are parameters associated with a particular state.

Imagine that we have a sequence of events produced by MinION and we would like to find the most probable DNA sequence that caused these events. We can perceive the events as an observations sequence o emitted by H. In such case, our problem of finding the DNA sequence is reduced to finding a state path s_{max} such that $s_{max} = \arg\max_s P(o|s)$.

In real-life application we count on using a particular instance of H provided by Nanopore Technologies [8]. Our algorithms however accepts any HMM that is structurally identical to H.

Chapter 2

Sampling

This chapter should persuade the reader about the viability of sampling for biological tasks, explain how sampling can be done on HMMs and describe the concrete algorithm we use.

2.1 Theory

2.1.1 Motivation for sampling

Hidden Markov models are most usually employed for finding the optimal solution to a problem, for example by running a Viterbi algorithm [12] to find the most probable state path for a given observation sequence.

In the case of Nanopore model running Viterbi algorithm and decoding the state path would yield us the most probable DNA sequence corresponding to the given observation sequence of events read from MinION.

Suppose that instead of the optimal solution we are interested in a set of solutions that have a chance of being close to optimal. The process of obtaining such set is called **sampling**.

Regarding our example with Nanopore model, by sampling from the HMM we could instead get several state sequences instead of only the most probable one. Note that we know the observation sequence beforehand and thus we are sampling from a posterior probability [5].

Practice shows that searching for such samples is meaningful as the resulting sequences retain significance from a biological point of view [15].

In figure 2.1.1 we can see how exploring the randomized solution space through samples can help us obtain useful information about a DNA sequence. We can see that the most probable path found with Viterbi algorithm does not bear similarity to majority of the samples on some of its parts. Based on this observation we could infer

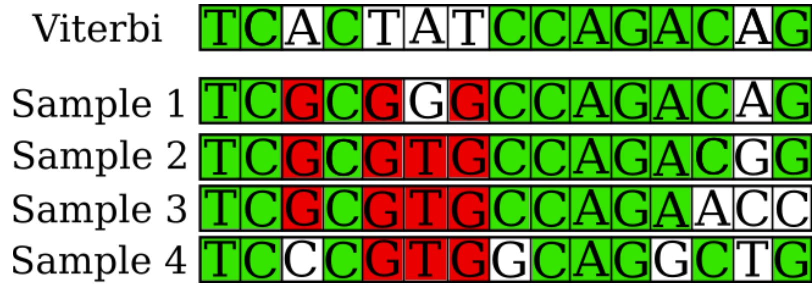

Figure 2.1: Example comparison of Viterbi path and samples

that these particular parts in the Viterbi path are an artifact since the alternate part is supported by many suboptimal solutions.

2.1.2 Forward Algorithm

Suppose we want to obtain a probability that the HMM generated an observation sequence o. The probability of a particular path $\pi = q_1, q_2, \dots q_n$ emitting an observation sequence $o = o_1, o_2, \dots o_n$ was already shown in the equation 1.1. In order to find out the probability of the HMM generating o we would need to sum up the probabilities of path π emitting e over all the paths.

$$P(o) = \sum_{\pi} P(o|\pi)$$

There can be exponentially many paths (in relation to the number of states) that generate the observation sequence o. Doing a brute-force search would thus be very inefficient for even small models. Fortunately, a polynomial time dynamic programming algorithm exists for this task.

The algorithm works by iteratively computing rows of matrix FW where $FW[i, k]$ expresses the probability that for an emitted observation sequence $o_1, o_2 \dots o_i$ the HMM ended up in state k.

The matrix of values $FW[i, k]$ for an observation sequence $o = o_1, o_2, \dots o_n$ can be computed recursively by formula

$$FW[1, l] = \frac{1}{|Q|} \cdot e(l, o_1) \tag{2.1}$$

$$FW[i + 1, l] = e(l, o_{i+1}) \cdot \sum_{k \in Q} FW[i, k] \cdot t(k, l) \tag{2.2}$$

In the end, we can obtain the final probability of HMM generating the $o = o_1, o_2, \dots o_n$ by calculating $\sum_{q \in Q} FW[n, q]$

2.1.3 Stochaistic tracebacking of sample

Suppose we have an observation sequence $o = o_1, o_2 \dots o_n$ We want to sample a state sequence $s = s_1, s_2 \dots s_n$ from a distribution described by probability mass function $f(s) = P(s|o)$ where $P(s|o)$ is the probability of our HMM moving through state path s and emitting o.

We will start constructing the sampled state sequence from the end. Having the FW matrix precomputed, we can sample the last state s_n of the sample with a probability

$$\frac{FW[n, s_n]}{\sum_{q \in Q} FW[n, q]}$$

Afterwards we sample for each next state s_i with probability

$$\frac{FW[i, s_i] \cdot t(s_i, s_{i+1}) \cdot e(s_{i+1}, o_{i+1})}{FW[i+1, s_{i+1}]}$$

We can repeat the above step until we run out of observations.

At the end, we have a sample state sequence $s_1, s_2, \dots, s_n$.

2.2 Algorithm implementation

The input for our algorithm consists of:

1. HMM model (in format as seen in Nanopore [8]) along with parameters such as the $prob_{stay}, prob_{skip}$ or maximum allowed skip distance
2. Sequence of events – An array labeled *events*

First, we load the HMM and compute the transitions according to the conditions formulated in section 1.4.

Next, we compute the FW matrix with m rows and n columns where m is the length of the event sequence and n is the number of states for our HMM. Matrix calculation can be implemented using a nested loop where the outer loop iterates over the increasing prefix lengths p_i of the observation sequence, while the inner loop goes through the states q_{to} of the HMM. Inside this inner loop is yet another loop which iterates all of the states q_{from} such that $t(q_{from}, q_{to}) > 0$ and calculates the sum (refer to 2.2) that ends up as a result for cell $FW[p_i, q_{to}]$.

Afterwards we traceback a sample sequence. We start by computing $S = \sum_{q \in Q} FW[n, q]$ and normalizing the last row of FW by S. We compute the prefix sum array p of this last row and then choose a random number r from uniform distribution on interval $[0, 1]$. Using binary search we find an index i of p such that $p[i-1] \leq r \leq p[i]$. Index i now refers to the position of a sampled state in the last row of FW. We mark the last

state as *prev* (previously seen state).
Having found the last state, we start a traceback routine consisting of a loop iterating over the decreasing prefix lengths s_i of the observation sequence. Inside it, we calculate

$$T = e(prev, events[s_i]) \cdot \sum_{q \in Q, t(q,prev)>0} FW[s_i, q] \cdot t(q, prev)$$

. We create an array w and fill it with values

$$w[q] = FW[s_i, q] \cdot t(q, prev) \cdot e(prev, events[s_i]) \quad \forall q \in Q$$

Again, we normalize w by T and calculate its prefix sum array w_p. We use the same process involving random number r and binary search to sample the next state (index) of w_p. The final step of the loop is to set the value of *prev* to the state that we have just sampled.

We repeat the traceback routine as many times as is the number of samples we need.

2.2.1 Sample path decoding

After we have obtained a sample state sequence $q = q_1, q_2, \dots q_m$ we need to decode it into the DNA sequence it represents. Recall that every state of the HMM is a different k-mer. Consider now a pair of states q_i, q_{i+1} that are next to each other in the state sequence. In the decoding process we look at the k-mers that the two states describe and find the longest prefix of q_{i+1} that matches the suffix of q_i – we call this substring an overlap. The result of decoding is the suffix of q_{i+1} that immediately follows the found overlap. In case of q_i being the first state of the sequence we prepend it to the suffix. Example of this process can be seen in figure 2.2.

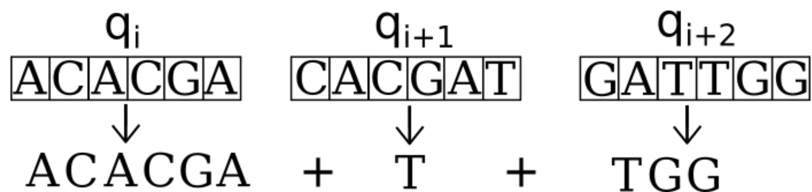

Figure 2.2: Path decoding example

The path decoding can be implemented via loop that iterates the state sequence from the start, calculates the length of the overlap for q_i, q_{i+1} in a nested loop and appends the non-overlapping part of q_{i+1} to the resulting string.

We repeat the above procedure starting from the sequence start for all pairs of consecutive states and concatenate the results into a single string which represents the sample DNA sequence.

2.2.2 Arithmetics in log scale

When calculating products of probabilities, such as in the Forward algorithm, we find out that after only a few steps of computation the resulting probabilities have a value that is too small to store in the limited space for floating point numbers in computers with classical architecture. For example, the most precise floating type in C language is the 64-bit *double* type which is capable of storing only numbers that have a value between 10^{-308} and 10^{308}. To overcome this problem we store logarithms of numbers instead. For instance, instead of storing 10^{-4247} (which we can not do with *double* type) we store $\ln(10^{-4247}) \sim -9779$ which fits in the *double* without problems.

Such modification requires us to handle the basic arithmetic operations differently. In case of number multiplication, we have to do addition since:

$$\ln(x \cdot y) = \ln(x) + \ln(y)$$

Similarly, division becomes subtraction:

$$\ln(\frac{x}{y}) = \ln(x) - \ln(y)$$

Addition is a bit more tricky:

$$\ln(x + y) = x + \ln(1 + e^{(y-x)}) \quad x > y$$

$$\ln(x + y) = y + \ln(1 + e^{(x-y)}) \quad y > x$$

When implementing these operations one must not forget to handle the problems arising from the use of $\ln(0)$ (which in most languages results in some form of minus infinity).

2.2.3 Time and space complexity analysis

The time complexity of the forward algorithm is $\mathcal{O}(nmd)$ where n is the number of HMM states, m is the length of the event sequence and d is the maximum indegree of all the HMM states. The space complexity is $\mathcal{O}(nm)$ since we just store the resulting matrix.

The time complexity of a single sample traceback is $\mathcal{O}(md)$ and space complexity is $\mathcal{O}(n)$ since we only store the resulting sample. The total space complexity for storing k samples is inevitably $\mathcal{O}(km)$ and the time complexity is $\mathcal{O}(kmd)$

As for the path-decoding phase, the time complexity for one sample is $\mathcal{O}(mk^2)$ where k is the length of a $k-mer$. The k^2 factor arises from the fact that we need to find the overlap by iterating the length of a $k-mer$ and comparing the resulting substrings for equality each time. Space complexity is $\mathcal{O}(km)$.

2.2.4 Used libraries

In order to parse FAST5 files that are produced by MinION we used the **fast5** library by Matei David.
For command line argument parsing and logging **Boost** library was used.

2.2.5 Abandoned attempt at implementation

During the tracebacking phase we are computing an array with length that of the indegree of HMM, normalizing it, and making prefix sums. Originally, we tried to precompute all of these prefix sum arrays during the forward matrix calculation. The reasoning was that for queries with large enough number of samples this would make the tracebacking phase faster as it would only need to binary search on an already existing array. In practice however, we found that the additional memory burden of an indegree factor far outweighed the advantage which was only present for a small set of inputs. Moreover, this memory overhead limited the test inputs for experiments to only a few thousands events and HMMs with small indegrees. For these reasons we abandoned this version of our algorithm, though it is still present in the code and can be triggered with a special flag.

Chapter 3

Parallelization through GPGPU

In this chapter we aim to educate the reader on the topic of graphic card usage for scientific purposes, provide examples of algorithm parallelization with concrete technology and describe how we transformed the sampling algorithm from the previous chapter into a parallel version along with an estimation of the performance gain that this conversion could provide.

3.1 Introduction to GPGPU and example cases

In their early years, graphic cards (often referred to as **GPU**s) were supporting merely specific fixed-function pipelines that served the purpose of creating image output [13]. Throughout the years however, GPUs have been becoming increasingly programmable and the focus split to utilizing their potential for uses other than just computer graphic computation. Their parallel nature proves to be valuable for scientific purposes, even more so because GPUs excel at fast floating point arithmetics.

The key difference between a standard CPU and GPU is in the chip structure. CPU is made to be as universal as possible and consists of only a few cores that focus on sequential processing while GPU has thousands of smaller, more efficient cores designed to run tasks in parallel. [13]

The term for the idea of GPU usage for tasks other than just image output has been coined as **General-purpose computing on graphics processing units**, abbreviated GPGPU.

Nowadays, the most widespread instance of GPGPU support is CUDA by Nvidia corporation. CUDA is a parallel computing platform that effectively provides a layer of abstraction between the instruction set of the graphic card and a high level programming language, such as C,C++ or Python. [6]

3.1.1 GPGPU applications

GPGPU has found uses in many areas : machine learning, physics, quantum chemistry and many more [11]. With closer focus on bioinformatics, there are solutions implementing CUDA for sequence alignment, motif discovery or evolutionary reconstruction [11].

3.2 Used techniques

The core concept of CUDA programming in C/C++ is separation of parallelized code into so called **kernels**. Kernels to GPU are more-or-less what threads are to CPU. From the programmer's perspective, the kernel is just a block of C code.
A simple kernel that doubles an array value can look like this:

```
__global__ void array_sum(int *d_a){
  unsigned int kernel_id = threadIdx.x;
  d_a[kernel_id] *= 2;
}
```

Figure 3.1: Simple kernel example

Many kernels can be run at once, asynchronously of the CPU which can continue code execution and be notified once the kernels terminate in order to collect the results.

Typical work flow in C/C++ when using CUDA consists of:

1. Copying input data from host memory into GPU memory
2. Launching kernels that process the input data and save the output into GPU memory
3. Copying output back into host memory

Let us demonstrate how the already shown kernel can be used to double every value of an array using parallel kernel execution.

```
__global__ void array_sum(int *d_a){
  unsigned int kernel_id = threadIdx.x;
  d_a[kernel_id] *= 2;
}

int a[5] = {1,3,2,7,5};
int *d_a;
cudaMalloc((void**)&d_a, 5*sizeof(int));  //allocate memory on GPU
cudaMemcpy(d_a,&a,5*sizeof(int),cudaMemcpyHostToDevice); //copy to GPU
array_sum<<<1,5>>>(d_a);  //run 5 kernels in parallel
cudaDeviceSynchronize();  //block CPU until all kernels finish
cudaMemcpy(&a,d_a,5*sizeof(int),cudaMemcpyDeviceToHost);
//output to RAM
//c now contains [2,6,4,14,10]
cudaFree(d_a);
```

Figure 3.2: Kernel usage example code

3.3 Parallel implementation of sampling algorithm

3.3.1 Parallelization bottleneck identification

Let us now explore how we can use the CUDA parallelization to accelerate the two essential algorithm phases:

- Forward matrix computation
- Tracebacking of the samples

First, consider the formula for Forward algorithm that calculates the matrix in 2.2.

As can be seen, in order to calculate the value of $FW[i+1, l]$ we need to have values $FW[i, x]\forall x$ computed beforehand, therefore it is impossible to split the calculation of matrix rows into independent parallel kernels and still satisfy the correctness of formula for the algorithm.
On the other hand, it can be seen that the calculation of $FW[i, l]$) does not depend on the value of $FW[i, m]$ for $m \neq n$. Therefore we can run n kernels $k_1, k_2, \ldots k_n$ where each kernel k_j computes the value of $FW[i, j]$ independently. In order to satisfy the execution order governed by the formula we have to wait until all the kernels finish running before launching them again for the calculation of the next matrix row.
Pseudocode that implements this idea through CUDA kernels resembles the one in figure 3.3.1

```
__global__ void fw_matrix_cell(double *fwm, double observation){
  //cell value computed here
}

for (int p = 1; p < observations.length; p++){
  fw_matrix_cell<<<n_of_states/threads, threads>>>(&fwm, observations[p]);
  cudaDeviceSynchronize(); //wait for all kernels to finish
}
```

Figure 3.3: Parallel forward matrix calculation pseudocode

Considering the sample tracebacking stage, we can safely conclude that there can be arbitrary number of instances where each instance tracebacks one sample. This is possible due to the fact that this stage does not write into any memory that is shared between the instances (only reads from the forward matrix).

The implementation is therefore straightforward as can be seen in figure 3.3.1

```
__global__ void traceback(double *fwmatrix, &samples){
  //one sample constructed here
}

for (int i = 0; i < num_of_samples; i++){
  traceback<<<num_of_samples/threads, threads>>>(&fwmatrix, &samples);
}
cudaDeviceSynchronoze(); //wait for all kernels to finish
```

Figure 3.4: Parallel sampling pseudocode

In case of path decoding phase we are allowed much greater freedom – decoding two consecutive states is an operation entirely independent of the decoding of any other part of the sample. Therefore, we can split each sample into blocks (possibly as small as a single state pair) and decode every block in parallel.

```
__global__ void decode_block(int *samples, int *results, int blocksize){
  //one sample block decoded here
}
//'samples' contains
decode_block<<<total_threads/threads_per_block,
               threads_per_block>>>(samples, results, blocksize);
cudaDeviceSynchronoze();
```

Figure 3.5: Parallel decoding pseudocode

3.3.2 Theoretical speedup estimation

In the previous chapter we concluded that the time complexity of the forward algorithm is $\mathcal{O}(nmd)$. By implementing the parallelization as seen in 3.3.1 and assuming perfect conditions where all n kernels are launched and terminated at the same time we expect to shrink the practical complexity to $\mathcal{O}(md)$.

Similarly, for the sampling stage we start with a time complexity of $\mathcal{O}(kmd)$ for k samples. By computing all k samples in parallel with an assumption of the same perfect conditions we should shrink the time complexity to $\mathcal{O}(md)$.

Lastly, the path decoding phase normally has $\mathcal{O}(nmk^2)$ time complexity for n samples of length m with k-mer size k. Decoding a pair of states takes $\mathcal{O}(k^2)$. If we choose a block size b and assume we have enough kernels to decode all of the blocks of all samples we should lessen the complexity for n samples to $\mathcal{O}(bk^2)$.

Chapter 4

Experiment design and results

4.1 Input data

The primary objective of our work is a speedup of sample generation and not the quality of samples in terms of metrics relevant for biology. For this reason we can afford to operate on random input which in turn is processed by a HMM which has some of its aspects chosen randomly as well.

We will be feeding our algorithm observation sequences consisting of event means generated randomly from an uniform distribution on interval $[0, 1]$

Because our algorithm works with HMMs of a form described in the Background chapter, we are limited to HMMs that structurally differ from the Nanopore model only in the k-mer size (and consequently the number of states) and maximal allowed skip. The means and standard deviations for the HMM states are both random numbers chosen from a uniform distribution on interval $[0, 1]$

Both in the case of event data and HMM state parameters the actual values should not affect computation speed since they are all stored as double precision numbers.

4.2 Testing method

We have created a set of tests all of which were run on both CPU and GPU version of the sampling algorithm. Each test is parametrized by following variables:

- k-mer size
- input data size (in number of events)
- number of generated samples
- maximum amount of bases the HMM can skip on transition

k-mer size	6
max skip	1
data size	5000
samples	100

Table 4.1: Parameter defaults

We have run test batches where one or two of the parameters were changing while the rest were fixed to defaults.

All of the computation done for the purpose of this thesis was performed on a server cluster equipped with Nvidia Tesla K40c and Intel Xeon CPU E5-2670 [4][7].
In order to not interfere with other running jobs we have set an artificial limit of 8GB for both RAM and GPU RAM as well as time limit of 15 minutes for every test. Test runs which would require more than stated memory limits were not executed and the results of those that exceeded the time limit were ignored.

We should note that the k-mers of the HMM for all the tests are strings over the alphabet $\{A, C, G, T\}$. Our algorithm accepts k-mers over arbitrary alphabet.

For each test we recorded the running time for following computation stages:

- Forward matrix calculation
- Traceback to find samples
- Sample path decoding

Each test was repeated 3 times and the final result is an average of the three measurements.

4.3 Running time comparison

In order to objectively judge the speedup of the non-parallel CPU algorithm and its parallelized GPU version on a particular hardware, we first need to know the performance of both the used CPU and GPU for a task which does not make use of the parallelism.
In our case, we ran a simple algorithm consisting of a loop which manipulated an array of double values. The reasoning behind this choice is that these operations are most common in our sampling algorithm. We recorded the total running time of this algorithm for variable array sizes.

```
__global__ void kernel(double *arr, int len){
double prev = 0.0;
for (int i = 0; i < len; i++){
        if (i % 2){
                arr[i] = prev - i;
        }
        else{
                arr[i] = prev + i;
        }
        prev = arr[i];
}
```

The result of this test can be observed in 4.1.

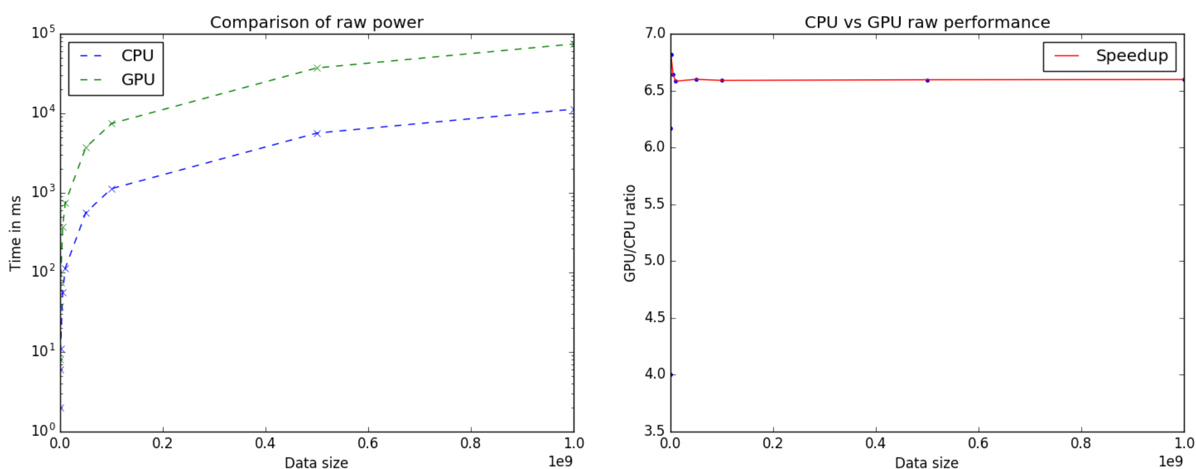

Figure 4.1: Running time of non-paralell calibration algorithm

We can see that the GPU is 7 times **slower** compared to the CPU for all input sizes. This can be easily explained by looking at the device architecture – the Xeon processor has very few cores that are more powerful than any of the Tesla's 2880 smaller cores. For better brevity, we will refer to this factor of 7 as **CPU/GPU factor**. Based on

this factor we can better estimate just how big of an impact the parallelization had. Knowing this, we can move on to the tests for the sampling algorithm.

In the graphs we compare the running times of the parallel sampling algorithm on the GPU and its non-parallel CPU version.

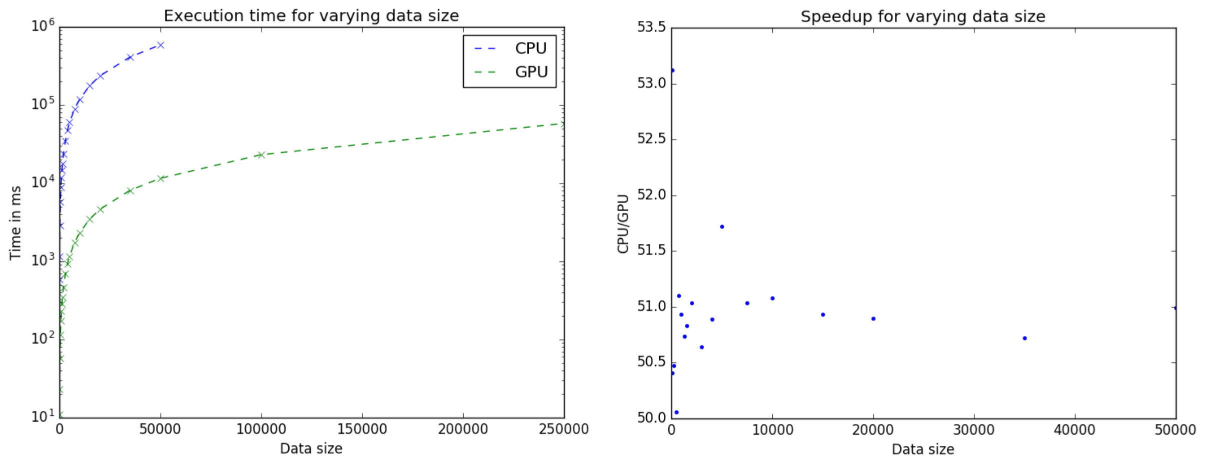

Figure 4.2: Running time of forward matrix calculation for varying data size

In case of forward matrix calculation for varying data sizes we can observe a uniform speedup of factor **∼50**. Since the rows of the forward matrix need to be computed sequentially and the number of states remained constant, such a result was predictable. Taking the CPU/GPU factor in the account, we can estimate a 350-fold increase of effectiveness on the GPU.

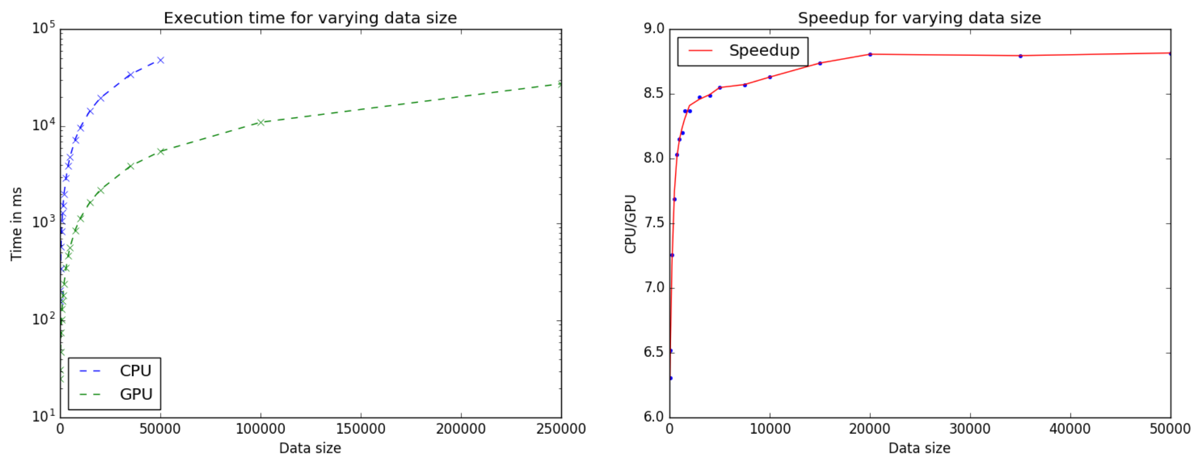

Figure 4.3: Running time of sampling phase for varying data size

Since the data size does not affect the number of kernels launched for sample trace-backing, the speedup is unsurprisingly constant.

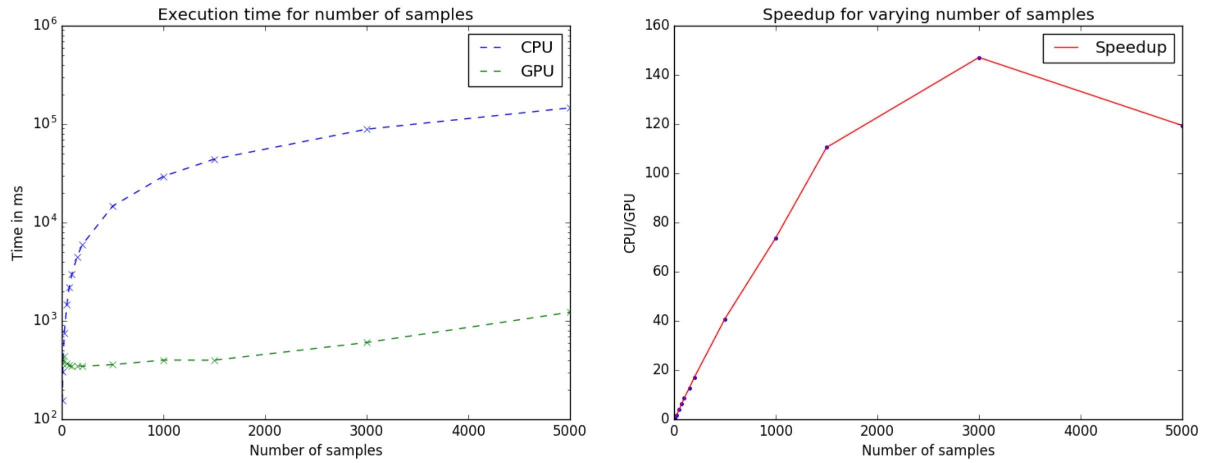

Figure 4.4: Running time of sampling phase for variable number of samples

In figure 4.4 we can see that the speedup grows larger with the increasing number of samples up until a certain point, which can be explained by the fact that the GPU can run only a limited number of kernels at the same time. We assume that after the GPU exhausts this limit, the scheduling overhead takes a toll on the speed. We do not include the results for matrix calculation as the number of samples has no effect on them due to constant data size and number of states. The obtained speedup reached a factor of **∼150** at its peak.
Considering the CPU/GPU factor, the effectiveness of GPU is 1050 times better with parallelization.

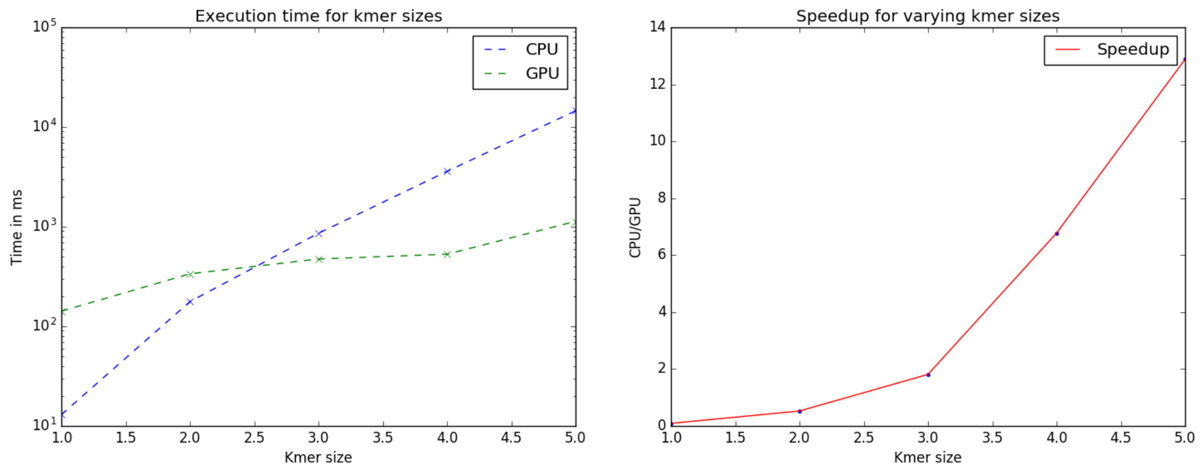

Figure 4.5: Running time of forward matrix phase for variable k-mer size

Changing the k-mer size and therefore the number of states results into more kernels used for forward matrix cell calculation. Due to this we can see an increasing speedup of up to **∼13x** as we increase the k-mer size.

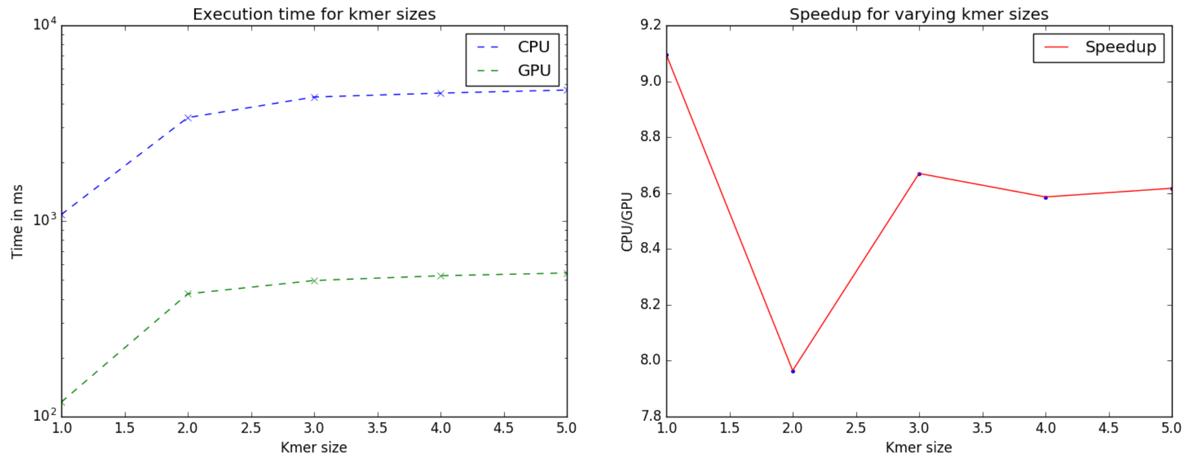

Figure 4.6: Running time of sampling phase for variable k-mer size

Similarly to results in Figure 4.3, changing the number of states does not change the constant speedup of **∼8.6x** for sampling phase.

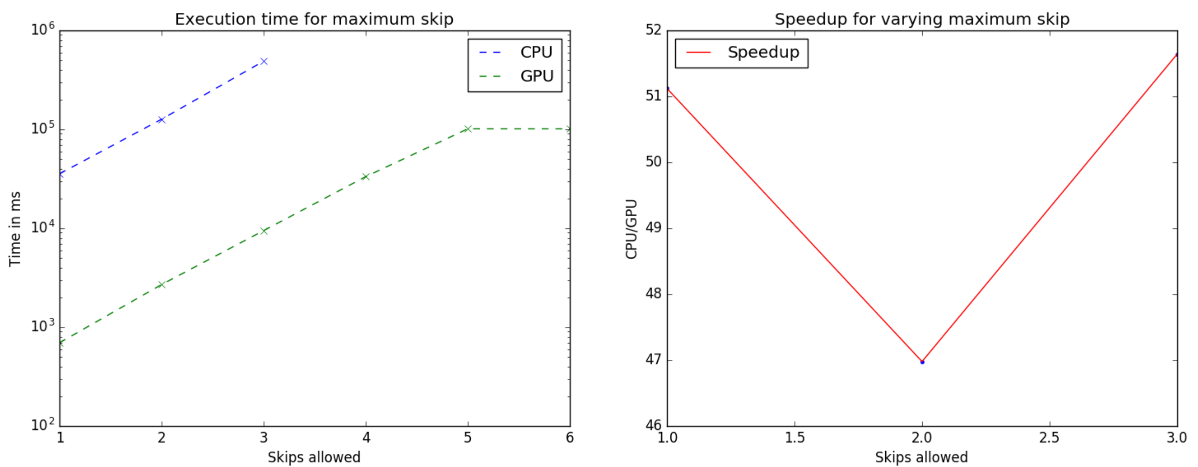

Figure 4.7: Running time for forward matrix phase for variable skip size

We can only hypothesize what caused the sudden speedup drop for skip size 2 in forward matrix calculation on GPU. Since the skip size determines the maximal indegree of states and therefore the execution time of each kernel it might have to do with how Cuda internally assigns the kernels into execution queues. The speedup again averages at **∼50x**.

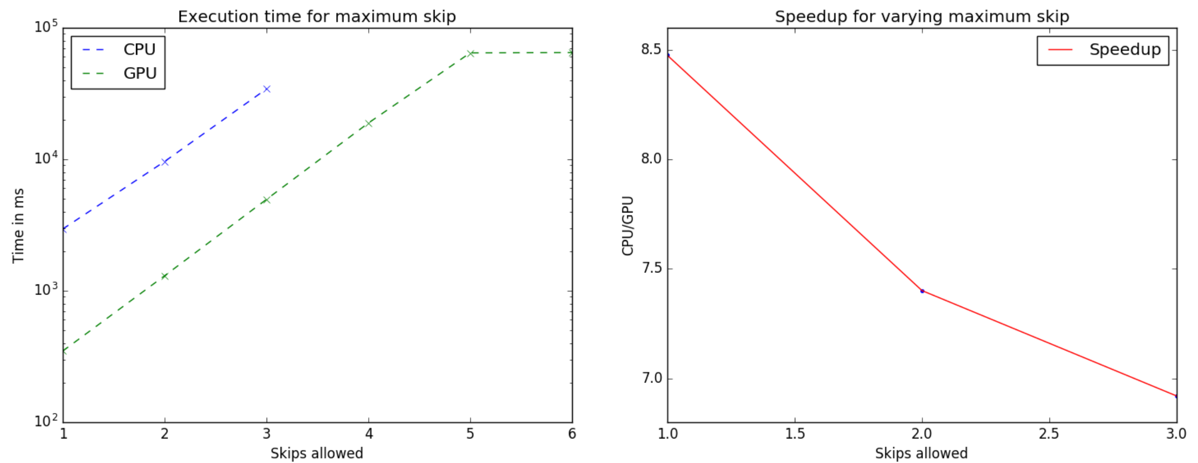

Figure 4.8: Running time for sampling phase for variable skip size

The speedup drop off here could be explained by contextualizing the CPU/GPU factor. The most time consuming part of sample tracebacking is the construction of prefix array that has a length of the maximal indegree and binary searching the state from it. Even though the CPU constructs samples sequentially, it should overtake the GPU in terms of a single prefix array calculation. As a result, the more we increase the maximum indegree of our HMM, the less advantage the GPU has.

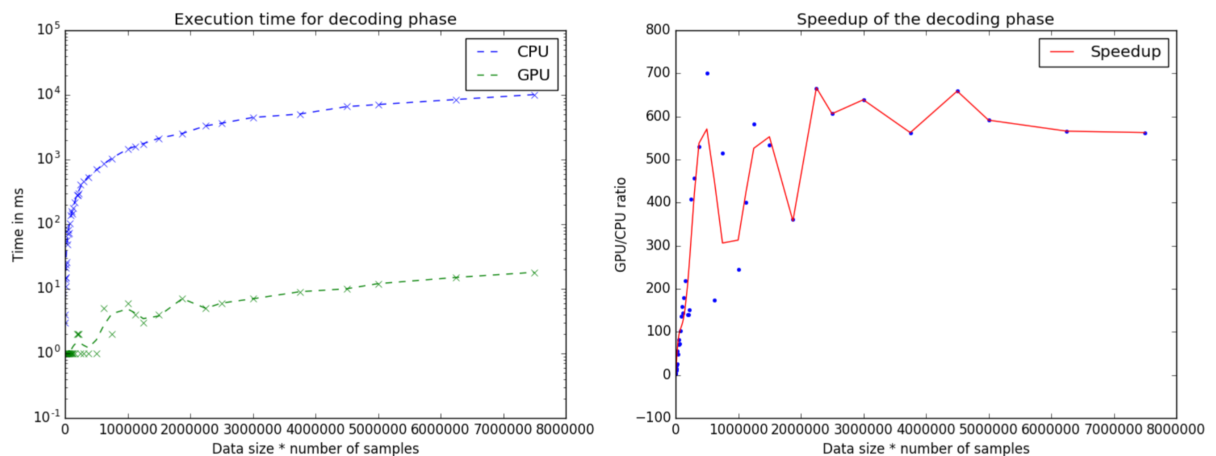

Figure 4.9: Running time for decoding phase

With path decoding phase we can once more observe a rising speedup of up to **650x** until a threshold where the GPU exhausted all available kernel instances. This means a 4550 times better effectiveness due to parallelization.

4.3.1 Theoretical vs. practical speedup

At the end of the previous chapter we set an optimistic expectation of the algorithm speedup where we assumed an ideal condition of GPU being able to run as many

kernels as we need with no additional overhead. As can be judged from the results these estimates were far off. For a HMM with 4096 states we have observed a 350 times better effectives compared to a hypothetical algorithm that does not use parallelism for the forward matrix calculation phase. While less than 10 times the ideal result , this can still be considered impressive. When it comes to the sampling phase we have seen as much as 1050 times better effectiveness for 3000 samples which is only worse from the theoretical maximum by a factor of 3. As for the path decoding phase, we obtained an impressive 4550x improvement for input value of 2250000 total states.

Chapter 5

Conclusion

In our work we were exploring the ways to perform sampling from posterior probability on a Hidden Markov model in an effective way by exploiting the parallel power of graphic cards. The goal was to design, implement and test an algorithm that would offer a speed advantage over a naive approach that only utilizes CPU and at the same time would be able to process the read data from the MinION sequencing platform.

In the first chapter we introduced the reader to the problematics of DNA sequencing and shown how the behaviour of the MinION platform can be modeled with Hidden Markov models. A particular form of Hidden Markov model was chosen for the rest of the work and described in greater detail.

The second chapter justified sampling as a viable process of inferring information about an event sequence produced by MinION as well as describing how the sampling can be implemented along with time and space complexity estimation.

In the next chapter we explored the topic of GPGPU, looked at example uses in other areas, and demonstrated how we can use CUDA API to create parallel versions of algorithms. In addition, we reasoned about the ways we can use parallelization for specific parts of our algorithm as well as about the limits that disqualify the use of such technique. Moreover, we took a closer look on how the parallelization could improve the already analyzed time complexity from previous chapter.

In the fourth and final chapter we described the testing methods, environment and input data and examined the results of running time for our parallel implementation compared to the naive CPU approach.

The tests proved rather conclusively that the parallel implementation offers significant speedup. Compared to a hypothetical GPU algorithm without parallelization we experienced a 350x speedup in forward matrix calculation, as much as 1050x speedup for multiple samples tracebacking and as high as 4550x speedup for path decoding phase. In comparison with the CPU performance the forward matrix calculation was sped up 50 times, sampling phase has seen as much as a 150x increase in speed while

the path decoding reached a 650x running time. These factors are especially noticeable for large datasets and HMMs where the difference between the CPU and GPU implementation is drastic enough to render the former hardly usable (running time of hours/days). while the latter still runs in reasonably quickly (running time of a few minutes). Even more importantly, these speedups occur for inputs that either resemble the real-life data (e.g. tens of thousands of events and HMM with k-mer size 6) or far exceed then, offering a reserve for future use.

We can conclude that in the case of sampling, parallelization is not only possible, but also a viable option that can very positively affect the performance of our algorithms.

5.1 Future work

While we successfully accomplished the task of producing large amounts of long samples in short time, there remains a question of how to further process these. It would definitely be useful to have a algorithm that can align [10] many sequences quickly, where again a GPU accelerated algorithms could come in useful.

Moreover, since the functionality of the HMM in our software is rather crude, we would like to implement features like parameter training to better prepare the software for real-life use.

Bibliography

[1] Dna. https://en.wikipedia.org/wiki/DNA. Accessed: 15.5.2017.

[2] Dna sequencing. https://www.genome.gov/10001177/dna-sequencing-fact-sheet/. Accessed: 15.5.2017.

[3] How does nanopore dna/rna sequencing work? https://nanoporetech.com/how-it-works. Accessed: 14.5.2017.

[4] Intel® xeon® processor e5-2670. https://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI. Accessed: 15.5.2017.

[5] Mcmc sampling for dummies. http://twiecki.github.io/blog/2015/11/10/mcmc-sampling/. Accessed: 15.5.2017.

[6] Nvidia's cuda: The end of the cpu? http://www.tomshardware.com/reviews/nvidia-cuda-gpu,1954-6.html. Accessed: 11.5.2017.

[7] Nvidia® tesla™ k40c gpu. http://www.thinkmate.com/product/nvidia/900-22081-2250-000. Accessed: 15.5.2017.

[8] Predictive kmer models for development use. https://github.com/nanoporetech/kmer_models. Accessed: 15.5.2017.

[9] Scalable, real-time biological analysis technology. https://nanoporetech.com/products. Accessed: 14.5.2017.

[10] Sequence alignment. https://en.wikipedia.org/wiki/Sequence_alignment. Accessed: 15.5.2017.

[11] Transforming computational research and engineering. http://www.nvidia.com/object/gpu-applications.html. Accessed: 11.5.2017.

[12] Viterbi algorithm. https://en.wikipedia.org/wiki/Viterbi_algorithm. Accessed: 15.5.2017.

[13] What is gpu-accelerated computing? `http://www.nvidia.com/object/what-is-gpu-computing.html`. Accessed: 11.5.2017.

[14] V. Boža, B. Brejová, and T. Vinař. DeepNano: Deep Recurrent Neural Networks for Base Calling in MinION Nanopore Reads. *ArXiv e-prints*, March 2016.

[15] Simon L Cawley and Lior Pachter. HMM sampling and applications to gene finding and alternative splicing. *Bioinformatics*, 19(suppl 2):ii36–ii41, 2003.

[16] Matei David, Lewis Jonathan Dursi, Delia Yao, Paul C Boutros, and Jared T Simpson. Nanocall: an open source basecaller for oxford nanopore sequencing data. *Bioinformatics*, page btw569, 2016.

[17] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[18] 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.

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

[20] Rastislav Rabatin. Alignment of nanopore sequencing reads, 2016. bachelor thesis at Comenius University.

[21] Lawrence Rabiner and B Juang. An introduction to hidden markov models. *ieee assp magazine*, 3(1):4–16, 1986.

[22] Eric E. Schadt, Steve Turner, and Andrew Kasarskis. A window into third-generation sequencing. *Human Molecular Genetics*, 19(R2):R227, 2010.

[23] Jay Shendure and Hanlee Ji. Next-generation dna sequencing. *Nature biotechnology*, 26(10):1135–1145, 2008.

[24] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.

Title of Appendix A

Attached to this thesis is a CD with the source code and test results in JSON format along with graphs. Source code can also be found on Github.