EXPLORATION OF GPU ACCELERATION FOR PAIR-HMM ALGORITHM AND ITS APPLICATION IN THE DNA ALIGNMENT PROBLEM

BY

ENLIANG LI

SENIOR THESIS

Submitted in partial fulfillment of the requirements
for the degree of bachelor of science in Electrical Engineering
in the Grainger College of Engineering of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Professor Deming Chen

ABSTRACT

The hidden Markov model, known as HMM, is an important type of statistical model with extensive application in estimating hidden parameters and decoding observed Markov Chains.

On top of the HMM, the Pair-HMM Algorithm with HalotypeCaller is developed as a popular solution for the DNA alignment problem. For two aligned sequences of DNA observations, one named as reference, and the other one named as read, there are only three possible hidden states, i.e. *match* (A , A), *insertion*(- , A), and *deletion*(A , -). However, what we could observe by DNA sequencing in real-life is the summation of the possibilities for *match*, *insertion*, and *deletion* as macrostates. In order to determine the alignment with maximum probability, we need to score each possible pairwise alignment and which leads to a computationally intensive problem that usually contributes to the most latency in a variant calling with the GATK HaplotypeCaller.

In the CPU implementation of a proper Pair-HMM forward algorithm, there are 7 multiply-accumulate operations for each (i , j) location on the read-reference matrix. Moreover, since *transitions* and *emission* matrices are fixed throughout a single alignment process, a CUDA implementation with single-precision floating-point is proposed to accelerate the Pair-HMM forward algorithm.

CUDA implementation with minibatch and states-parallelization, along with the use of float32, gives us an around 22.6x speedup compared to the CPU implementation. While it comes with a price, using single-precision instead of double-precision floating-point introduces a more serious underflow problem at the beginning of the alignment scoring process. A normalization technique is used to help fix this problem.

Subject Keywords: Hardware Acceleration; DNA Alignment; Pair-HMM; Forward Algorithm; CUDA implementation

ACKNOWLEDGMENTS

I would like to express my special thanks to my advisor, Professor Deming Chen, as well as my colleague, Anand Ramachandran, for their assistance and enthusiasm.

I am also grateful for Ashutosh Dhar's effort for providing technical support while using the GCAD server.

This senior thesis would have been impossible without their help.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Hidden Markov Model 1
1.2 Pair-Hidden Markov Model 2
1.3 DNA Alignment Problem 3

CHAPTER 2 LITERATURE REVIEW 4
2.1 GPU Related Work 4
2.2 FPGA Related Work 5

CHAPTER 3 GPU ACCELERATION IMPLEMENTATION 6
3.1 Data Structures 6
3.2 Basic Parallelization 7
3.3 Use of Single-Precision Float 9
3.4 Other GPU optimizations 10

CHAPTER 4 RESEARCH RESULTS 11
4.1 Compared with CPU Baseline 11
4.2 Compared with Existing Work 12

CHAPTER 5 CONCLUSION 14

REFERENCES 15

CHAPTER 1

INTRODUCTION

In a typical DNA alignment problem, the Pair-HMM (Pair Hidden Markov Model) forward algorithm is most commonly used to evaluate the overall likelihood of any possible alignments between two DNA sequences. At the same time, the Pair-HMM forward algorithm could easily occupy 70% of the total execution time of the GATK HaplotypeCaller. In this thesis, we implement a proper Pair-HMM forward algorithm on GPU with CUDA to reduce the execution time of each forward function call to improve the performance of a GATK flow. It could, also be adapted for other application, such as the HOLMES alignment algorithm

1.1 Hidden Markov Model

As we're discussing a hidden Markov model(HMM), it could be separated into two sides: one is the observable part with sequences emitted by the other side of the model, a hidden state machine. It is worth pointing out, the hidden state machine needs to meet the criterion of a Markov process, namely, describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event.

In a traditional HMM, there are a few parameters used to describe the system, they are Visible Alphabets, Set of States, Transition Probabilities between States, Start Probabilities, and Emission Probability for each Visible Alphabets per State.

1.1.1 Mathematical Definition [1]

Visible Alphabets could be represented by,

$$\sum = \{b_1, b_2, ..., b_M\}$$

Set of States in the Model could be represented by,

$$Q = \{1, ..., K\}$$

The Transition Probability from state i to state j could be represented by a_{ij},and $a_{i1} + ... + a_{iK} = 1$ for all states $i = 1...K$.

The Start Probabilities a_{0i} for all states $i = 1...K$. And their sum has to be 1.

The Emission Probability for each state $e_i(b) = P(x_i|\pi_i = k)($ for all states $i = 1...K$. And their sum has to be 1.

1.1.2 Forward Algorithm

The Forward Algorithm(FA) is an efficient way to solve a HMM decode problem by calculating the joint probability of $P(x_t|b_{1:t})$, where x_t represents the hidden states and $b_{1:t}$ represents all observations from the beginning through position t.

And a proper FA could be written as [1],

```
for $i = 1$ to $t$:
    $f_k(i) = P(x_1...x_i, \pi_i = k)$
    $= \sum_{\pi_1...\pi_{i-1}} P(x_1...x_{i-1}, \pi_1, ..., \pi_{i-1}, \pi_i = k)e_k(x_i)$
    $= \sum_l \sum_{\pi_1...\pi_{i-2}} P(x_1...x_{i-1}, \pi_1, ..., \pi_{i-2}, \pi_{i-1} = l)a_{lk}e_k(x_i)$
    $= \sum_l P(x_1...x_{i-1}, \pi_{i-1} = l)a_{lk}e_k(x_i)$
    $= a_{lk} \sum_l f_l(i-1)e_k(x_i)$
end for
```

Figure 1.1: Forward Algorithm

1.2 Pair-Hidden Markov Model

The only difference between a proper Pair-Hidden Markov Model (Pair-HMM) and HMM should be now there are two visible sequences instead of one. In most cases, the Pair-Hidden Markov Model could be characterized by the same way as we do in HMM, with each state emitting two Visible Alphabet b_t instead of one each time.

With the information stated above, a modified Emission Probability for Pair-HMM should be sufficient, and it could now be represented by,

$$e_i(b_{t1}, b_{t2}) = P(x_i|\pi_i = k).$$

1.3 DNA Alignment Problem

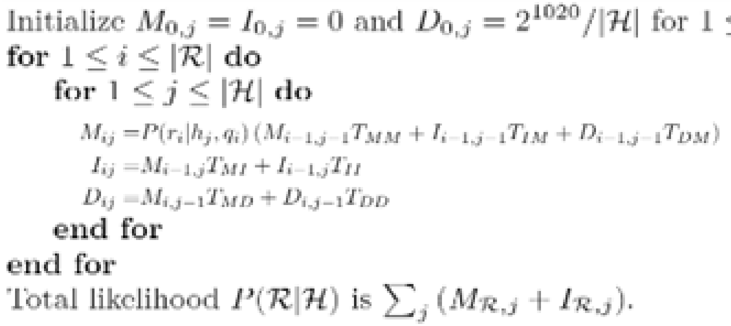
In the DNA alignment problem, the structure of the Pair-HMM is fixed with three states, they are *match*(M), *insert*(I) and *delete*(D). Given the following two aligned sequences,

```
Read : A  C  T  C  G  -
Ref :  A  C  -  -  G  T
```

We have the matching states to be M, M, I, I, M, D. Thus, the score P of this specific alignment could be calculated as [2],

$$e(A,A) * a_{MM} * e(C,C) * a_{MI} * e(T,-) * a_{II} * e(C,-) * a_{IM} * e(G,G) * a_{MD} * e(-,T)$$

And this could be done efficiently by the Pair-HMM forward algorithm, which will be the focus of this paper. Furthermore, the pseudocode of the Pair-HMM forward algorithm could be written as [3],



```
Initialize $M_{0,j} = I_{0,j} = 0$ and $D_{0,j} = 2^{1020}/|\mathcal{H}|$ for $1 \le j \le |\mathcal{H}|$.
for $1 \le i \le |\mathcal{R}|$ do
    for $1 \le j \le |\mathcal{H}|$ do
        $M_{ij} = P(r_i|h_j, q_i)\,(M_{i-1,j-1}T_{MM} + I_{i-1,j-1}T_{IM} + D_{i-1,j-1}T_{DM})$
        $I_{ij} = M_{i-1,j}T_{MI} + I_{i-1,j}T_{II}$
        $D_{ij} = M_{i,j-1}T_{MD} + D_{i,j-1}T_{DD}$
    end for
end for
Total likelihood $P(\mathcal{R}|\mathcal{H})$ is $\sum_j (M_{\mathcal{R},j} + I_{\mathcal{R},j})$.
```

Figure 1.2: Pair-HMM Forward Algorithm

CHAPTER 2

LITERATURE REVIEW

The hardware acceleration of the HMM and Pair-HMM algorithm has been a popular research topic for a while, and there are considerable amount of existing works before this paper. In this chapter, we're going to briefly review the optimization attempts from other works, and some of those techniques do motivate our implementation as well.

2.1 GPU Related Work

Inter-task and Intra-task parallelization are proposed [4].

2.1.1 Proposed implementation

Inter-task parallelization has each thread with independent implementation, and thus it allows many copies of the algorithm running in parallel. Instead of keep track of the whole $forward\_matrix$, each thread, which corresponds to a cell in the matrix, it only records the values of the direct top, left, and top-left neighbors for computing the current cell.

Intra-task parallelization is less intuitive and more complicated to implement. Because the calculation of each cell in the $forward\_matrix$ only depends on its top, left and top-left neighbors, there is a well-known wave-front pattern propagates through the anti-diagonal of the whole $forward\_matrix$ during the scoring process. The intra-task parallelization calculates cells along the anti-diagonal in parallel.

They also make use of the tile-based implementation, which is also a very common optimization technique on GPU to reduce the global memory access.

However, since they implement each thread independently, the control divergence leads to a serious waste of computation power as well as lack of flexibility. If the size of the $forward\_matrix$ changes dramatically, i.e. the ratio of length of DNA read to reference varies, the performance could suffer a lot.

2.1.2 Results

The performance of the proposed implementation mentioned above has a speed up effect of 154x on a **10s** dataset comes from a Whole Genome Sequence (WGS) dataset [5].

The corresponding running time is 70ms, given that the one on original Java on CPU is 10800 ms.

2.2 FPGA Related Work

A processing element(PE) ring structure is proposed to accelerate the Pair-HMM algorithm [6].

2.2.1 Proposed implementation

Thanks to the scalability of field-programmable gate array (FPGA), fixed length PEs propagate along the anti-diagonal of the $forward\_matrix$ similar to the intra-task parallelization in GPU, but could cooperate with each other through internal buffer.

The FPGA implementation is also benefit from its lightweight heading nature, so in most situations, it beats GPU implementation in *Throughput per watt*[7].

2.2.2 Results

The performance of the proposed implementation mentioned above has a speed up effect of 2038x on a **10s** dataset comes from a Whole Genome Sequence (WGS) dataset [5] on Stratix V platform, and 4154x if the same algorithm is implemented on the Arria 10.

The corresponding running time is 5.3 ms (Stratix V) and 2.6 ms (Arria 10), given the one on original Java on CPU is 10800 ms.

CHAPTER 3

GPU ACCELERATION IMPLEMENTATION

The optimized GPU Acceleration is implemented with CUDA developed by Nvidia Corporation. In this section, we will first introduce a convenient way of organizing the data and then the optimization techniques used in our implementation. Specifically, in section 3.3, a new way of mitigating the potential drawback from replacing 64-bit floating point with the 32-bit one will also be discussed.

3.1 Data Structures

To better understand how to design a straightforward and GPU-friendly data structures, we should first visualize a Pair-HMM forward work flow by mapping the whole scoring process into a 2-dimension matrix as shown in Figure 3.1,

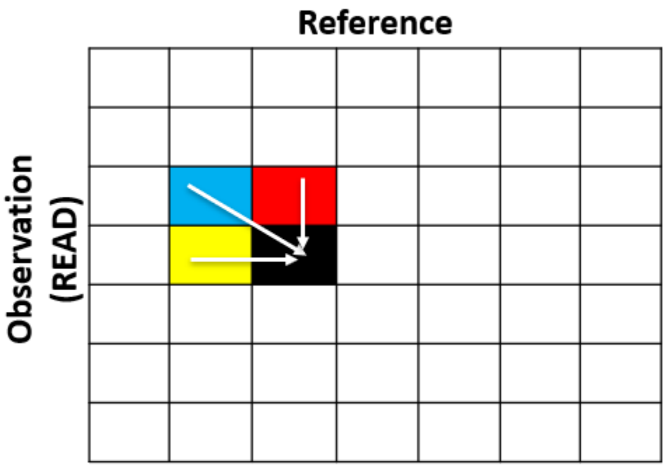


Figure 3.1: Forward Matrix

In Figure 3.1, the number of cells in each row and each column refer to $(x_{dim} + 1)$ and $(y_{dim} + 1)$ of the whole system. As the evaluating process proceed, the black cell is the current work, the blue cell refers to the *match*, the red cell refers to the *deletion*, and yellow the *insert*.

Based on the given information above, we put the scoring results in a matrix named *forward_matrix* with the dimension of $[x_{dim}+1][y_{dim}+1][batch][states]$. In this setup, the data is organized in a GPU-friendly way that allows flexibility for parallelization.

As we mention in section1.1 and 1.2, to characterize a Pair-HMM, we also need the following matrices *transitions*, *emissions* and *start_transitions*. Following the same idea, they are structured as,

$$\begin{gathered}\text{transitions}[x_{dim}+1][batch][states-1][states]\\ \text{emissions}[x_{dim}+1][y_{dim}+1][batch][states]\\ \text{start\_transitions}[batch][states-1]\end{gathered}$$

In the next section, we will introduce how to fill in the data and process them.

3.2 Basic Parallelization

In the previous section, we introduced the data structure used in this CUDA implementation. Now we discuss the parallelization across the mini-batch and three different states: *match*, *insert*, and *deletion*.

3.2.1 Minibatch Parallelization

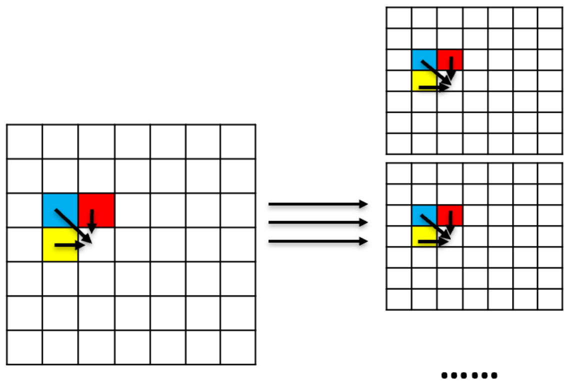


Figure 3.2: Minibatch Parallelization

In a supervised learning or a GATK variant call, usually only segments of DNA sequences will be examined and evaluated. They are discontinued with respect to their locations in their parent sequences. The scoring work needs to be done individually between two distinct pairs of observations versus references.

In Figure 3.2, by introducing the *batch* dimension parallelization, the program could evaluate the likelihood of multiple pairs of DNA sequences simultaneously. Even the $x_{dim}+1$ or $y_{dim}+1$ are not the same across *batch*, we just need to leave the *emissions* to be zeros for those unused spaces in the $forward\_matrix$.

As preprocessing the input data, we will pick up the following highlighted dimension,

$$\begin{gathered}\text{transitions}[x_{dim}+1][\mathbf{batch}][states-1][states]\\ \text{emissions}[x_{dim}+1][y_{dim}+1][\mathbf{batch}][states]\\ \text{start\_transitions}[\mathbf{batch}][states-1]\\ \text{forward\_matrix}[x_{dim}+1][y_{dim}+1][\mathbf{batch}][states]\end{gathered}$$

3.2.2 Across States Parallelization

After a careful observation of the pseudocode of the Pair-HMM algorithm, we should find that the calculations of Match, Insert and Deletion are individually at each cell.

In Figure 3.3, by introducing the *states* dimension, the program could evaluate the different *states* within a cell at the same time.

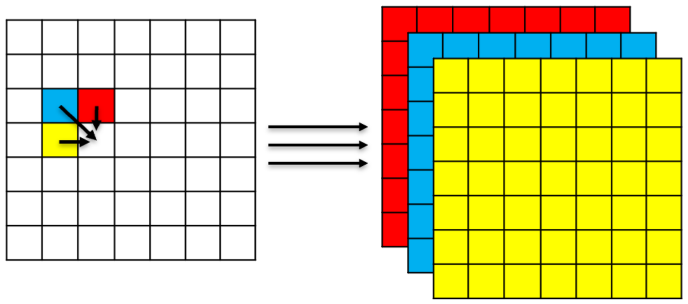


Figure 3.3: Across States Parallelization

As preprocessing the input data, we will pick up the following highlighted dimension,

$$\text{transitions}[x_{dim}+1][batch][\mathbf{states-1}][\mathbf{states}]$$

$$\text{emissions}[x_{dim}+1][y_{dim}+1][batch][\textbf{states}]$$
$$\text{start\_transitions}[batch][\textbf{states}-\textbf{1}]$$
$$\text{forward\_matrix}[x_{dim}+1][y_{dim}+1][batch][\textbf{states}]$$

It is worth mentioning, as we modify the depth of the *states* dimension, we give it the capability of adapting different scenarios. For example, in the case of the HOLMES alignment algorithm, there will be 27 states instead of 3 states for each cell in the forward matrix.

3.3 Use of Single-Precision Float

The idea of using 32-bit float instead of 64-bit double to optimize the implementation comes from the fact that in modern computing, a significant index of performance for a hardware is floating point operations per second (FLOPS).

In the case of GPU, FLOPS is calculated as the number of multiply-accumulate (i.e. $y = a * x + b$) operations possible per second. And for example, on the platform of TitanXp, theoretical FP32(single-precision) FLOPS is 12.15 TFLOPS[8], where FP64(double-precision) FLOPS is 379.7 GFLOPS[8], around 1/32 of the FP32.

3.3.1 Normalization technique [9]

As we mention in the header of chapter3, using float instead of double will cause more serious underflow problem at the beginning of the alignment process, thus a new way is introduced to the Pair-HMM forward algorithm.

To reduce the underflow brought by using float, we define a pair of functions. The **normalization(value)** function preprocesses the input data by,

$$rep = value/normalization\_factor$$

And then when we want to denormalize it, we call the function **denormalization(rep)**,

$$value = log(rep) + log(normalization\_factor)$$

3.4 Other GPU optimizations

In our CUDA implementation of the GPU acceleration, some trivial but traditional optimization techniques are also applied.

For emissions and transitions matrices that may be frequently reused along the calculation, a shared memory copy is made from the global memory to reduce the times of relatively slower global read-writes.

Also, motivated by an existing work [4], we make use of CUDA intrinsic instruction $\_\_fmaf\_rn()$ to computes the value of $a * x + b$, which reduces the execution time by preventing any possible redundant operations.

CHAPTER 4

RESEARCH RESULTS

In this chapter, we will present the comparison of the results with the C++ implementation on a CPU and also those existing work presented in Chapter2.

4.1 Compared with CPU Baseline

For the comparison with our CPU baseline, we have three different setups with random input data to simulate the computation complexity of corresponding Pair-HMM forward calculations.

The running time is measured in milliseconds, estimated by 100 consecutive function calls as shown in Table 4.1,

	Read length : 18 Ref length : 15 Num of Batch : 1	Read length : 18 Ref length : 15 Num of Batch : 3	Read length : 148 Ref length : 158 Num of Batch : 1
"forward_cpu.cpp" (i7-6650U CPU @ 2.20GHz) [double]	**21.0816 ms**	**N/A**	**1638.26 ms**
"forward_gpu.cu" (TITAN Xp) [double]	**13.0121 ms**	**13.7454 ms**	**72.4838 ms**
"forward_gpu.cu" (TITAN Xp) [float]	**12.6679 ms**	**12.9385 ms**	**69.3239 ms**

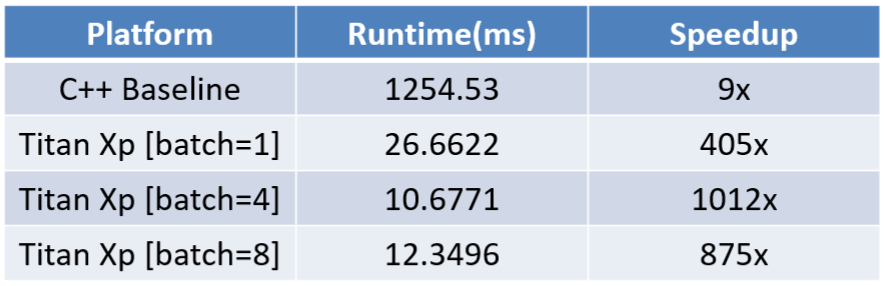
Table 4.1: CPU Comparison

In Table 4.1, by comparing the first and the second column, we find the CUDA implementation on Titan Xp could process three batches within almost the same amount of time as only working with single batch, which is equivalent to a 3X speed up for the number of batch to be 3 compared to sequential computation.

Since the global memory on GPU is very precious resources, and for Titan Xp, the global memory is 12 gigabytes, which could barely hold a *forward_matrix* with the size of [159 × 149], single batch. And the speed up for this specific setup is around 1638.26/72.4838 = 22.60x for double precision and 1638.26/69.3239 = 23.63x in average.

4.2 Compared with Existing Work

To compare our work with the existing ones, we can use the **10s** dataset comes from a Whole Genome Sequence (WGS) dataset [5], and the performance should look like Table 4.2,



Platform	Runtime(ms)	Speedup
C++ Baseline	1254.53	9x
Titan Xp [batch=1]	26.6622	405x
Titan Xp [batch=4]	10.6771	1012x
Titan Xp [batch=8]	12.3496	875x

Table 4.2: Performance based on 10s dataset

Where the performance of the existing works could be found in the following Table 4.3 [6],

Platform	Runtime(ms)	Speedup
Java on CPU	10800	1×
C++ Baseline	1267	9×
Intel Xeon AVX Single Core	138	78×
NVidia K40 GPU	70	154×
Intel Xeon 24 Cores	15	720×
Altera OpenCL (Stratix V)	8.3	1301×
PE Ring (Stratix V)	**5.3**	**2038**×
Altera OpenCL (Arria 10)	2.8	3857×
PE Ring (Arria 10)	**2.6**	**4154**×

Table 4.3: Performance comparison across various implementation

The best speedup from our work is at around 1012x, with *batch* equal 4s and from Table 4.3 we can see those better performance implementations are achieved by FPGA implementation.

Compared to the GPU implementation on Nvidia K40 GPU [4], the speedup is around 1012/154 = 6.57x. And the two biggest contributions should be the parallelization across batch, and we're running our code on a more powerful GPU based on the new generation Pascal architecture.

CHAPTER 5

CONCLUSION

In this thesis, with the simple parallelization and use of single-precision floating-point format (Float32), we achieve a speedup of 22.6x with a size of *forward_matrix* with $[158 \times 148]$ compared to the C++ CPU implementation.

Although the throughput of our CUDA implementation is not as competitive as those with FPGA, its flexibility could prove the significance. If working with Python, to include the CUDA implementation in a GATK variant call or any related application, we just need to use the pyCUDA package to instantiate the Pair-HMM call, where pyCUDA is a light-weight Python wrapper for CUDA API, supporting Numpy array as inputs.

The project is available at GitHub:

`https://github.com/lienliang/Pair_HMM_forward_GPU`

REFERENCES

[1] Victoria Popic and Serafim Batzoglou and John Louie, "CS262 Lecture6: Hidden Markov Models Continued," Jan 2015. [Online]. Available: https://web.stanford.edu/class/cs262/archives/notes/lecture6.pdf

[2] Rishi Bedi and Serafim Batzoglou, "CS262 Lecture8: Pair Hidden Markov Models," Jan 2015. [Online]. Available: https://web.stanford.edu/class/cs262/archives/notes/lecture8.pdf

[3] Broad Institute, David Benjamin, "Pair HMM probabilistic realignment in HaplotypeCaller and Mutect," Aug 2017. [Online]. Available: https://github.com/broadinstitute/gatk/blob/master/docs/pair_hmm.pdf

[4] Ren, Shanshan and Bertel, Koen and Al-Ars, Zaid, "Exploration of alternative GPU implementations of the pair-HMMs forward algorithm," *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 2016.

[5] "Pair-HMM test data." [Online]. Available: https://github.com/MauricioCarneiro/PairHMM/tree/master/test_data

[6] S. Huang, G. J. Manikandan, A. Ramachandran, K. Rupnow, W.-m. W. Hwu, and D. Chen, "Hardware acceleration of the pair-hmm algorithm for dna variant calling," pp. 275–284, 2017. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021749

[7] S. S. Banerjee, M. El-Hadedy, C. Y. Tan, Z. T. Kalbarczyk, S. S. Lumetta, and R. K. Iyer, "On accelerating Pair-HMM computations in programmable hardware," *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, 2017.

[8] techpowerup.com, "Nvidia TITAN Xp Tech Specs," Apr 2017. [Online]. Available: https://www.techpowerup.com/gpu-specs/titan-xp.c2948

[9] A. R. University of Illinois at Urbana-Champaign and D. C. Coordinated Science Laboratory, "Development code base with the ESCAD group."