IMPROVED GPU IMPLEMENTATIONS OF THE PAIR-HMM FORWARD ALGORITHM FOR DNA SEQUENCE ALIGNMENT

BY

ENLIANG LI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Adviser:

Professor Deming Chen

ABSTRACT

With the rise of Next-Generation Sequencing (NGS), the clinical sequencing services become more accessible but also facing new challenges. As we discovered closed connection between key DeoxyriboNucleic Acid (DNA) mutation spots and major diseases or conditions, the need for computational genomics has increased significantly. The surging demand motivates developments of more efficient algorithms for genome assembly, error correction, k-mer counting etc. In this thesis, we focus on DNA sequencing analysis, one of the fastest growing markets in NGS, and its related alignment problems.

In recent years, many new hardware technologies and algorithms have been researched for their potential applications in massive parallel sequencing. The emerging hardware includes GPU, FPGA and other ASICs providing parallel processing resources. In this thesis, we choose GPU as our computation platform for its massive parallel processing capabilities.

The Forward Algorithm (FA) still remains one of the most commonly used methods in solving sequences alignment problems modeled as Pair-Hidden Markov Model (HMM). The Pair-HMM Forward Algorithm (FA) is not only a computation but data intensive algorithm. Multiple previous works have been done in efforts to accelerate the computation of the FA by applying massive parallelization on the workload, and in this thesis, we bring more optimizations not only by improving the computation concurrency of both initialization process and Pair-HMM FA but also by tackling the communications overhead between the host and devices. We will discuss the general principles of optimizing the Forward Algorithm on GPU and present an improved implementation of the Pair-HMM FA with native CUDA C++.

Our design has shown a speedup of 25.10x over the C++ baseline on the GATK HaplotypeCaller Pair-HMM workload with a portion of the real dataset from human genome database, *NA12878*. This is a major improvement that beats the state-of-the-art implementation with a margin of 60%.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to express my special thanks to my thesis advisor, Professor Deming Chen, as well as my colleagues, Subho Sankar Banerjee and Anand Ramachandran, for their generous assistance and innovative ideas when I was facing challenges.

I am also grateful to Professor Ravishankar K Iyer for providing access to the CSL Symphony server, which enabled me to conduct experiments and collect results needed for completing this thesis.

This thesis contributes to a funded project by Mayo Clinic, one of the leaders in Clinical Next-Generation Sequencing services. I would like to thank Mayo Clinic for inspiring the motivation for this work.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

ALU	Arithmetic-Logic Unit
ASIC	Application-Specific Integrated Circuit
CUDA	Compute Unified Device Architecture
DNA	DeoxyriboNucleic Acid
FA	Forward Algorithm
FPGA	Field-Programmable Gate Array
GATK	Genome Analysis ToolKit (by Broad Institute)
GPU	Graphics Processing Unit
HMM	Hidden Markov Model
indels	(DNA) insertions and deletions
LoC	Line of Code
NGS	Next-Generation Sequencing
PHMM	Pair-Hidden Markov Model
PU	Processing Unit
SM	Streaming Multiprocessor (a GPU component)
SNP	Single Nucleotide Polymorphism

CHAPTER 1

INTRODUCTION

After the millennium, with the emergence of new hardware components and the explosive grow of computation power, many sequencing or statistical problems that took days or weeks to complete are now solvable with effective algorithms in hours. In the research area of computational genomics, scientists now see the light at the end of the tunnel after decades of trials to identify unexpected key DNA mutations for early diagnosis of cancer or other serious human diseases at the clinic [1], [2].

Next-Generation Sequencing (NGS) technology has the features of low-cost and high-performance. To further improve the efficiency and continue providing affordable analysis resources in response of the increasing availability of genome data through public databases, researchers have started investigating the possibility of deploying new algorithms on GPU [3], [4] and FPGA [5], [6], [7], [8], [9] given the fact that bioinformatics workloads could be executed in parallel given their batch processing nature.

In a typical DNA alignment problem, the Pair-HMM forward algorithm (FA) is used for evaluating the overall likelihood of any possible forms of alignments between two DNA sequences. Multiple read sequences are issued with respect to a reference sequence, and we are interested in how likely any of these two sequences could be aligned with each other considering all types of mutations. To interpret the raw biological data and translate them into meaningful bioinformatics workloads, several tools have been developed, including the Genome Analysis ToolKit (GATK).

GATK is a popular and widely used toolkit for DNA analysis both in research and in the clinic. Mayo Clinic has been one of the leaders applying whole-exome sequencing technology in personal healthcare and investing in GATK related research. It has become more feasible now for clinical practices to identify the early stage of serious disease by DNA variant calling.

During per-read likelihoods determination, the FA could easily occupy

Table 1.1: Execution Time Breakdown

Stage	Absolute Time (hr)	Percentage Time
Plausible Haplotypes Assembly	2.87	13.8
Pair-HMM FA	14.78	71.1
Realign and Others	3.13	15.1

more than 70% of the total execution time of the GATK on a portion of the popular *NA12878* sample from the GIAM consortium [10] while we are searching for mutation spots between the read and reference sequences as listed in Table 1.1.

The main contributions of this thesis include the following:

- We identify the bottleneck procedures of GTAK HaplotypeCaller and re-explore the source code provided by official GATK and implement the GPU version of many key auxiliary functions used by the Pair-HMM FA, including *emissions* and *transitions* matrices initialization.
- We discuss principles for GPU acceleration and propose a comprehensive method including parallel data flow, tasks pipelining on GPU and shared memory technique to improve the Pair-HMM FA on GPU for higher throughput to speed up the DNA alignment process.
- We implement the proposed method with native CUDA C++ and achieved the best speedup of $25.10X$ on a real GATK HaplotypeCaller workload.
- Our implementation is easy to be maintained and deployed. An integrated version with the GATK HaplotypeCaller is provided through the modified library loader for the Intel Genomics Kernel Library (GKL) and Java Native Interface (JNI).

In Chapter 2, we first explain the motivation of this work and background of the Pair-HMM FA. Chapter 3 summarizes previous works on improving the Pair-HMM FA and how our implementation mitigates the drawbacks in previous works. With the observations made in Chapter 3, we propose our design in Chapter 4 and present the results of comprehensive evaluations with our implementations in Chapter 5.

CHAPTER 2

BACKGROUND AND MOTIVATION

The market for NGS technologies has grown dramatically in recent years. Within all NGS services, the reproductive health NGS test is the fastest growing one [1] and its worldwide market size is expected to reach $3.3 billion [2] by 2022. Traditional clinical NGS services include diagnostics, risk prediction in cancer and other diseases (e.g. cardiovascular), therapy selection and monitoring, and screening.

As the market continues to grow, research on new software and effective algorithms dedicated for bioinformatics workload will become more significant and necessary.

2.1 GATK and Its HaplotypeCaller

Genome Analysis Toolkit [11], GATK, is a toolkit developed by the Broad Institute as part of its Data Sciences Platform. The GATK provides a wide range of tools with a major focus on variant discovery and genotyping. It not only supports NGS technologies with high-performance computing features but also becomes the industry standard in terms of single nucleotide polymorphisms (SNPs) and DNA insertions and deletions (indels).

From isolating DNA all the way to the results for SNP and Indel, we need to take every step carefully following a well-documented protocol, named *Best Practices* for SNP and Indel [12], [13] to optimize yield and ensure reproducibility. And the GATK provides the whole pipeline following the standard *Best Practices* as shown in Figure 2.1 [11], [14].

With the pipline, bioscientists or laboratory personnel could now digitalized their DNA read sequences at the earliest possible step and pass the rest of the workload to the GATK or any other similar toolkits. NGS technologies minimize human participation and unnecessary repeated workload for SNPs

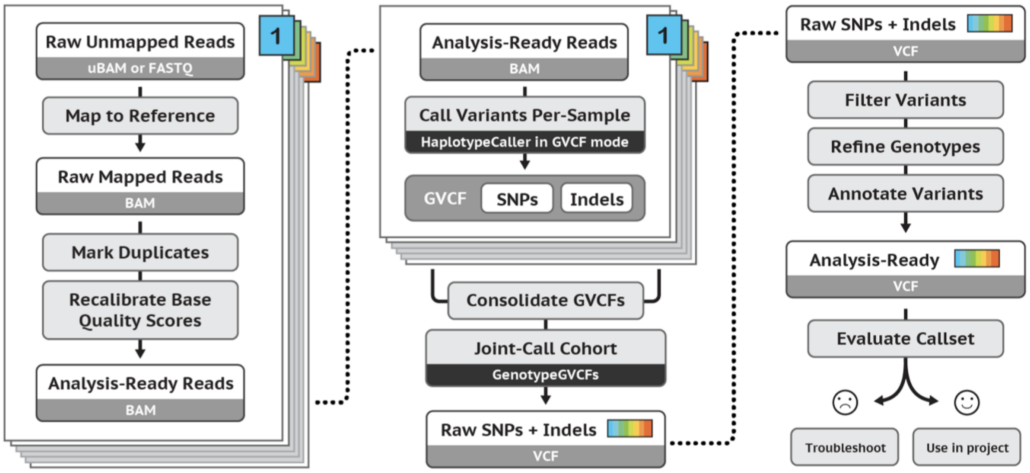

Figure 2.1: GATK Best Practices for SNPs and Indels Discovery
Source: Facts adapted from [14]

and Indels discovery, which makes the DNA sequencing results more reliable and accessible.

Many other commercial solutions or toolkits have been seen in the market, including the *Sequel IIe* system from the Pacific Biosciences (PacBio) as well as the *Clara Parabricks* from Nvidia. However, they are either not open-source or require modifications to the existing industry-standard *Best Practices* pipline.

Within the *Best Practices* pipline, the *HaplotypeCaller* is the tool that occupies most of the wall time [15], and thus optimizing the *HaplotypeCaller* could bring potentially the most reduction of the total runtime of the pipeline.

As we begin to optimize the GATK *HaplotypeCaller*, we further identify its underneath algorithm, Pair-HMM, as the most time-consuming algorithm according to Table 1.1. In order to accelerate the GATK *HaplotypeCaller*, we will need to develop an efficient implementation for the Pair-HMM.

2.2 Hidden Markov Model (HMM)

DNA variation during evolution could be modeled by Hidden Markov Model (HMM), and the HMM is following a Markov process, which must be stochastic and satisfying the property of memorylessness [16]. The HMM could be separated into two components: one is the observable side (*Observation* $\mathcal{O}_T$ at time step T) with sequences emitted by the hidden side of the

system, which is a hidden Markov process; the other is not directly visible to us and includes *States* with independent *Transition* rates over time, which represents current status of the process.

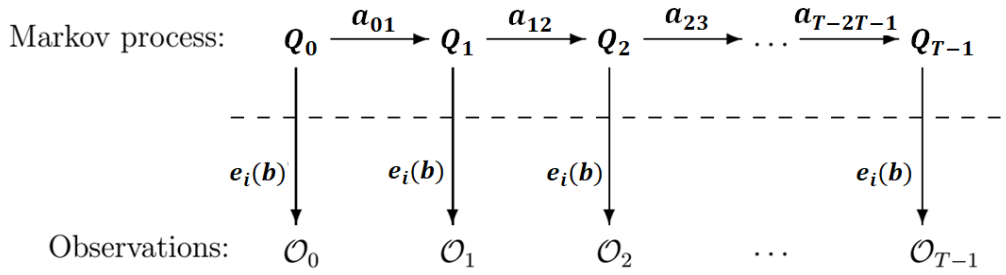

Figure 2.2: An Example of HMM

In an HMM, there are a few parameters used for describing the system: *Visible Alphabets, Set of States, Transition Probabilities between States, Start Probabilities* and *Emission Probability* for *Visible Alphabets* per *State*.

The mathematical definition for a formal HMM:

1. *Visible Alphabets* could be represented by

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

   Observations must fall into the set of *Visible Alphabets*: $\mathcal{O}_T \in \sum$.

2. *Set of States* in the model could be represented by

$$Q = \{1, ..., K\} \text{ in Figure 2.2}$$

3. The *Transition Probability* from *State* i to *State* j is represented by

$$a_{ij} \text{ in Figure 2.2}$$

   where $a_{i1} + ... + a_{iK} = 1$ for all *States* $i = 1...K$. It is the possibility of each *State* goes to another *State* (or stays at itself). The sum of *Start Probabilities* a_{0i} for all *States* $i = 1...K$ has to be 1.

4. The *Emission Probability* for each *State* is represented by

$$e_i(b) \text{ in Figure 2.2}$$

where $e_i(b) = P(b|Q_i = k)$ for all *States* $i = 1...K$. And their sum has to be 1. It is the possibility of emitting a *Visible Alphabet* b given current *State* Q_i.

2.3 Pair-Hidden Markov Model (Pair-HMM)

The Pair-Hidden Markov Model (Pair-HMM) inherited most of the properties from HMM with the only difference that now we need to deal with two visible sequences instead of one. Thus, the Pair-HMM could be characterized by a similar way compared to HMM, with each hidden *State* Q_i emitting two *observations* $\mathcal{O}_{T,1}$ and $\mathcal{O}_{T,2}$ instead of one at each time step T as shown in Figure 2.3.

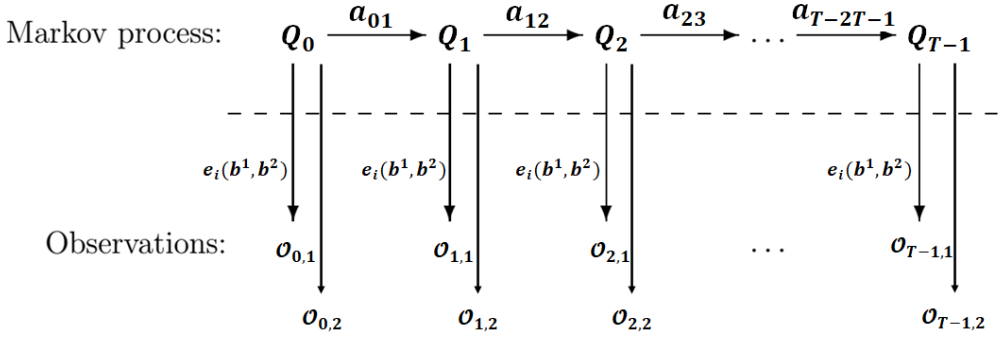

Figure 2.3: An Example of Pair-HMM

The mathematical definition for a formal Pair-HMM needs a modified *Emission Probability*, which could now be represented by

$$e_i(b^1, b^2) = P(b^1 : b^2|Q_i = k)$$

for all *States* $i = 1...K$. And their sum has to be 1. Here, b^1 and b^2 represent the pair of *Visible Alphabets* emitted by current *State* Q_i.

2.4 DNA Alignment Problem and Forward Algorithm (FA)

We could model the DNA alignment problem with Pair-HMM with three hidden states: *match* (M), *insert* (I), *delete* (D); and five *Visible Alphabets*: *adenine* (A), *cytosine* (C), *guanine* (G), *thymine* (T) and *placeholder* (-).

Table 2.1: An Example of Alignment Attempt

read seq :	A	C	T	C	G	-
ref seq :	A	C	-	-	G	T

Consider two aligned sequences as in Table 2.1. For this specific alignment attempt, we could infer the Pair-HMM hidden states to be M, M, I, I, M, D as shown in Figure 2.4.

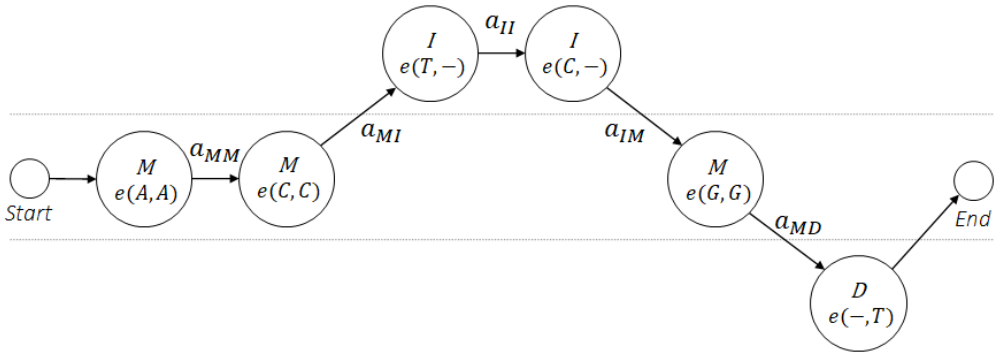

Figure 2.4: Equivalent Pair-HMM Hidden States Evolution

Thus, the likelihood of this alignment happens given this pair of *read-ref* sequences could be evaluated by likelihood score P for the possibility of its equivalent Pair-HMM hidden states evolution.

The likelihood score P could be formulated as

$$P = e(A,A) \times a_{MM} \times e(C,C) \times a_{MI} \times e(T,-) \times a_{II} \times e(C,-) \times a_{IM} \times e(G,G) \times a_{MD} \times e(-,T)$$

The forward algorithm (FA) is an efficient way to solve the Pair-HMM decode problem by calculating the joint probability of $P(x_t|b_{1:t})$, where x_t represents the hidden states and $b_{1:t}$ represent all pairs of observations from the beginning through position t. A native implementation of the Pair-HMM FA used by GATK HaloptypeCaller is listed in Algorithm 1.

A *Haplotype* is a group of genes within an organism that was inherited together from a single parent. In the step of Pair-HMM alignment, each read is aligned to many candidate *Haplotype* sequences with likelihood score calculation [17]. We use these candidate *Haplotype* sequences as reference sequences in the scope of Pair-HMM [18].

Algorithm 1 Pseudocode for Pair-HMM Forward Algorithm

1: **procedure** FORWARD ALGORITHM(R, H, a, e)
▷ R is the Read and H is the Haplotype* sequence
2: **Initialize:**
$M_{0,j} = I_{0,j} = 0$ and $D_{0,j} = 1/|H|$ $1 \leq j \leq |H|$
3: **for** $i = 1$ to $|R|$ **do**
4: **for** $j = 1$ to $|H|$ **do**
5: $M_{i,j} \leftarrow e(R_{i,j}, H_{i,j})(M_{i-1,j-1}T_{MM} + I_{i-1,j-1}T_{IM} + D_{i-1,j-1}T_{DM})$
6: $I_{i,j} \leftarrow M_{i-1,j}T_{MI} + I_{i-1,j}T_{II}$
7: $D_{i,j} \leftarrow M_{i,j-1}T_{MD} + D_{i,j-1}T_{DD}$
8: **end for**
9: **end for**
10: **end procedure**

Total likelihood $P(R|H)$ is
$\sum_j (M_{|R|,j} + I_{|R|,j})$ for $1 \leq j \leq |H|$
$*$: **Haplotype is used as reference sequence in PHMM**.

The Pair-HMM FA iterates through all possible alignments between the read and reference sequences by the method of *update-and-proceed*. In *line3* and *line4* of the algorithm, the alignment problem is divided into smaller alignment problems with subsequences of the read sequence and subsequences of the reference sequence. In *lines 5 to 7*, the total likelihood of aligning the read subsequence with length from one to *ReadLength* against the reference subsequence with length from one to *HaplotypeLength* is updated with the infer hidden states of match (M), insert (I) and delete (D). Finally, in *line 10*, we will be able to know the total likelihood for all possible alignment patterns by summing up the outstanding likelihood from previous steps.

In the actual implementation of the Pair-HMM FA, match matrix M, insertion matrix I and deletion matrix D could be stacked into a three-dimensional array named $forward\_matrix$. The $forward\_matrix$ keeps track of intermediate results when we are stepping forward during the execution and it should have a dimension of $ReadLength \times HaplotypeLength \times NumberofStates$.

The Pair-HMM FA shown in Algorithm 1 is computational intensive because its time complexity on CPU is $\mathcal{O}(mn)$ where m is the length of DNA read sequence and n is the length of DNA haplotype (reference) sequence.

The algorithm requires an extra space for $forward\_matrix$ and thus its space complexity is $\mathcal{O}(mn)$ as well.

CHAPTER 3

RELATED WORK AND OBSERVATIONS

Previous works have tried to improve the Pair-HMM FA from different perspectives. S. Ren et al. [3], [4] utilize GPU to parallel the workload and accelerate the Pair-HMM FA by increasing throughput; S. S. Banerjee [5], S. Huang [6] and L. van Dam [7] implement Pair-HMM FA on Field-Programmable Gate Array (FPGA), with optimized data forwarding strategies and special computation routines.

We analyze these existing works and gain insights that guide us through the implementation of this work.

3.1 GPU Related Work

Two independent optimization methods for the Pair-HMM FA are proposed in a previous GPU work in [3], [4]: *Inter-task* and *Intra-task* parallelization. We use this work as one of the baselines for evaluation.

1. *Inter-task* parallelization makes GPU thread to work on each independent pair of alignment, and thus it allows many copies of the algorithm running in parallel. Instead of keeping track of the whole $forward\_matrix$, each GPU thread only records values of its direct top, left, and top-left neighboring cells for updating the current cell.

   The obvious drawback of this naive *Inter-task* parallelization is that it does not effectively reduce the computational complexity visible to a GPU thread of the Pair-HMM FA since there is still a nested *for Loop* in the implementation which leads to $O(mn)$ time complexity.

2. *Intra-task* parallelization is less intuitive and more complicated to implement. Because the update of each cell in the $forward\_matrix$ only depends on its direct top, left and top-left neighboring cells, we can

parallel the processing of cells in anti-diagonal during Pair-HMM FA execution.

However, for *Intra-task* parallelization, the data structure becomes a lot more complicated since the indexing of all cells in the $forward\_matrix$ is no long linear. Compared to the *Inter-task* parallelization, the occupancy of GPU computational resources will also suffer since the length of anti-diagonal is not fixed while we are propagating from the top-left to the bottom-right of the $forward\_matrix$. Threads processing along the anti-diagonal could be in idle and causing control divergence in the Streaming Multiprocessor (SMs), wasting computational resources.

3.2 FPGA Related Work

Modern FPGA is comprised of configurable integrated circuits with millions of logic gates. Its outstanding flexibility allows researchers and programmers design and deploy hard-wired logic to accomplish dedicated tasks in efficient ways.

S. S. Banerjee [5] and S. Huang [6] proposed a similar Processing Element (PE) ring structure to accelerate the Pair-HMM FA by hardware routines supporting arithmetic operations within the Pair-HMM FA and data forwarding paths to pass data between PEs.

The PE ring structure exploits the anti-diagonal calculation pattern similar to *Intra-task* parallelization on GPU.

With the scalability of FPGA, fixed length PEs that propagate along the anti-diagonal of the $forward\_matrix$ could cooperate with each other through internal buffers, i.e., intermediate results from one pair of DNA sequences could be captured and reused while calculating another pair of DNA sequences nearby, and thus further improve the efficiency of FA. This is one of the advantages of FPGA implementations since GPU executes instructions in a lock-step style which makes internal sharing expensive.

The work in [7] approaches the Pair-HMM improvement by using posit number format. The posit number is a different way to represent floating point numbers as IEEE754 format.

By calculating Pair-HMM FA with 32-bit (float) posit arithmetic, authors state the decimal accuracy increases by two decimals of accuracy on average

compared to the IEEE754 format. The decimal accuracy is an evaluation metric proposed by J. Gustafson [19] as

$$\text{decimal accuracy} = -log_{10} \mid log_{10}(\frac{\widetilde{X}}{X}) \mid$$

where $\widetilde{X}$ is the computed and X is the exact floating point value.

The FPGA implementation of posit-based Pair-HMM is 10^5 to 10^6 times [7] faster than its software implementation with improved decimal accuracy on implementations based on the IEEE754 format.

However, optimizations for implementation on FPGA are usually device-specific and require extra efforts to be deployed across different devices. In general, GPUs are relatively easier to program as well compared to FPGAs.

3.3 Observations

GPU is generally more powerful than FPGA in terms of higher cores frequency, more dedicated ALUs and larger memory bandwidth. FPGA has its own advantages in terms of lower power consumption and customized computing. We decided to use GPU as our hardware platform given its better potential for further Pair-HMM FA improvements and higher portability of CUDA implementations on different GPUs.

In previous GPU implementations, authors evaluate their Pair-HMM FA implementation with synthetic data and measure the end-to-end speedup for integrated GATK *HaplotypeCaller* [4], [5]. However, they do not cover the following:

1. the memory transfer overheads between host and GPU, which drags the overall performance of Pair-HMM FA on GPU down if not handled well

2. overhead for generating *transmission* and *emission* probabilities matrices on CPU

3. potential improvements with concurrent processing of multiple DNA sequences

4. allocation of computational resources according to different lengths of read and reference sequences

In Chapter 4, we exploit these opportunities for Pair-HMM FA improvements and present our design of Pair-HMM FA on GPU.

CHAPTER 4

DESIGN AND IMPLEMENTATION

The improved GPU accelerator for the Pair-HMM Forward Algorithm (FA) is implemented with CUDA. In this chapter, we discuss principles for optimizing Pair-HMM FA on GPU and introduce details on GPU implementations, including the applications of some common tricks or practices that could help improving the overall performance of GPU algorithms.

4.1 Principles and Mythologies

Our proposed implementation is based on the following observations and insights:

1. In a real application of the DNA alignment problem, there are usually multiple read sequences aligned against single reference sequence in a batch [1]. And the alignment calculation of each pair of *read-reference* sequences is independent.

2. To generate and initialize *transitions* and *emissions* matrices, we need per-based substitution (including match) Quality scores *readQuals*, insertion Gap Opening Penalties *insertionGOP*, deletion Gap Opening Penalties *deletionGOP* and Gap Continuation Penalties *overallGCP* *etc*. The *transitions* and *emissions* matrices only need to be initialized once if multiple pairs of *read-reference* sequences share the same initialization parameters.

3. Initialization of each cell within *transitions* and *emissions* matrices does not depend on each other and thus we could generate them in parallel using GPU.

4. GPU supports multiple streams and thus operations including kernels

invocations, host-to-device and device-to-host memory transfers could overlap with each other to hide the latency.

4.1.1 GPU and CUDA

Graphics Processing Unit (GPU) refers to a kind of special design computer hardware used for graphics rendering. Different manufacturers may follow various principles when designing their GPUs products, but they should all share the same methodology of processing pixels in a large scale of parallelization [20]. In the early years of GPUs, their graphics pipelines are immutable due to the fixed logic implemented by circuits. However, with the raising concept of General Purpose Graphics Processing Unit (GPGPU), researchers started looking for possibilities of using GPUs for accelerating existing algorithms with high parallelism. For Nvidia GPUs, the basic execution unit is Warp: Warps will be captured and executed by Streaming Multiprocessors (SMs). A modern GPU could have as many as 84 SMs (Tesla V100), each of which contains thousands of threads.

Compared to FPGA, GPU is generally running on a faster clock. A much faster and stable clocking with better portability makes GPU a smarter choice than FPGA in terms of implementation for commonly used algorithms like Pair-HMM FA. The main drawback of GPU is its lockstep execution for all threads within the same SM, and thus consecutive branching in a program could cause serious control divergence on the GPU that will eventually compromise overall performance. However, in recent years, GPUs manufacturers like Nvidia have noticed this flaw in design and some solutions have been proposed and implemented in their new architecture. For example, the Volta Architecture has introduced a new feature of Independent Thread Scheduling which enables intra-warp synchronization patterns and a finer grid so programmers have more freedom.

CUDA (Compute Unified Device Architecture) is a High Performance Computation (HPC) platform developed and maintained by Nvidia; it could be deployed on any devices that support CUDA. In CUDA programming [21], the basic unit is *thread*, and *threads* will be grouped into *blocks*, where *blocks* could be further grouped into *grids*. Nevertheless, all *threads* will become parts of *warps* and SMs after compilation before execution. So we need to

be careful of the granularity issue to guarantee computational resources on GPUs are properly distributed and maximally utilized.

Our implementation is written in C++/CUDA that contains 1k LoC.

4.1.2 Data Structures and Pair-HMM FA Dependencies

To better understand how to design a straightforward and GPU-friendly data structure, we should first visualize a part of the Pair-HMM FA workflow with a two-dimension matrix as shown in Figure 4.1,

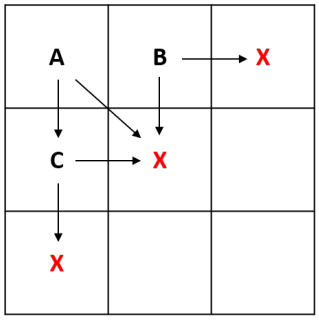

Figure 4.1: Pair-HMM FA Data Dependencies

The cells with letter A, B and C are completed ones, and those marked with red crosses are cells ready to be updated. Black arrows indicate the data dependencies between each cell and we will utilize this dependencies patterns in next subsection to parallelize execution. In the Pair-HMM FA, total numbers of cells in each row and each column are represented by $(x_{dim}+1)$ and $(y_{dim}+1)$, which correspond to *ReadLength* and *HaplotypeLength*.

Based on this observation, we put intermediate results in an array named $forward\_matrix$ with the size of $(batch, states, x_{dim}+1, y_{dim}+1)$. But we actually do not need to explicitly keep this $forward\_matrix$ during implementation because it could be replaced by a two-row shared memory discussed in Section 4.4.

As we mention in Section 2.4, to calculate the total likelihood P, we also need matrices *transitions* and *emissions*. They could by organized as,

$$\text{transitions}[batches][transitions][x_{dim}+1]$$
$$\text{emissions}[batches][states][x_{dim}+1][y_{dim}+1]$$

In this case, we can easily translate the indices according to current *threadIdx* and *blockIdx* in $\mathcal{O}(1)$ time. To keep memory accesses coalesced, we need

to make sure the order of dimensions follow the memory accesses patterns during calculation since elements in high-dimensional matrices are stored sequentially in GPU memory. For this reason, *batch_id* should come as the first dimension, then *state_id* and finally the positions of a single cell, indexed by *read_pos* (read position) and *hap_pos* (haplotype position).

4.2 Computation Parallelization

By observing the DNA sequencing data, we discover the lack of dependencies between cells updating in three levels. (1) In a pair of sequences, updates of cells in the direction parallel to anti-diagonal (from the lower left corner to the upper right corner of a matrix) do not depend on each other. (2) In a pair of sequences, updates of $MatchMatrix$, $InsertionMatrix$ and $DeletionMatrix$ do not depend on each other. (3) Across all pairs of sequences to be processed, total likelihoods of each pair of sequences do not depend on each other.

Across-Diagonal Parallelization: We design a basic execution unit, Processing Unit (PU) as shown in Figure 4.2, to carry out the Pair-HMM FA computation. A PU is composed of multiple GPU *threads* with shared memory access, which allows efficient cooperation between these GPU *threads*. The length of a PU is always integral multiple of the *warp* size in GPU for friendly *warps* execution scheduling.

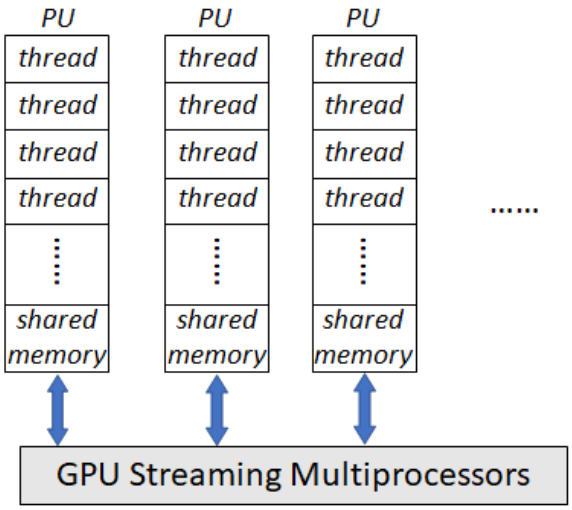

Figure 4.2: Architecture of Processing Unit

PUs will be used to process cells across the direction parallel to the anti-diagonal of the matrix and propagate forward until the whole $forward\_matrix$ has been updated.

An example of across-diagonal parallelization is given in Figure 4.3, assuming the length of a single PU is 4 GPU *threads*. Blue cells indicate those have been completed, yellow ones represent cells the PU is working on, and green ones will be the cells processed next in parallel by the PU.

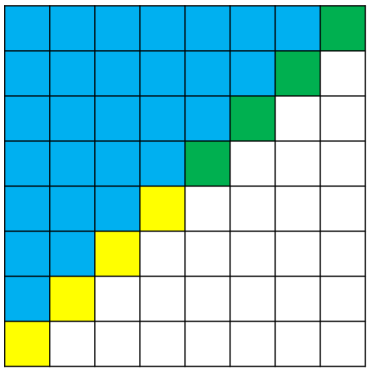

Figure 4.3: Pair-HMM FA Across-Diagonal Parallelization

We find out that a longer PU does not necessarily bring better performance. A single PU with too many GPU *threads* could potentially reduce SM occupancy because when Pair-HMM FA updating cells at the top-left and bottom-right boundaries, lots of GPU *threads* will be in idle.

Across-State parallelization: In Figure 4.4, where blue refers to $Match$, red refers to $Deletion$ and yellow refers to $Insertion$. The updates of three *States* M, I, D do not depend on each other and thus could be executed in parallel by distributing the workload into different *threads*. In our implementation, GPU identifies which *State* each *thread* is working on by its `block_idx.y` value.

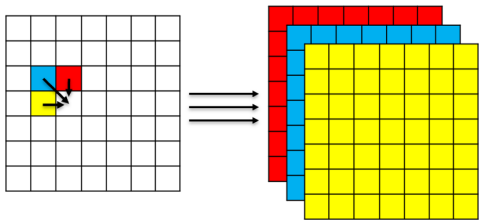

Figure 4.4: Pair-HMM FA Across-State Parallelization

The parallelization across pairs of sequences could also be viewed as Batch-parallelization. The batch level parallelization fully utilizes GPU power through the Single Instruction Multiple Threads (SIMT) computational model. On GPU, we are paying extra overheads like data padding, data transfer between host and device as well as kernel invocation for high parallelism in return.

As shown in Algorithm 2, the time complexity for the Pair-HMM FA has now reduced from $\mathcal{O}\ (mn)$ to $\mathcal{O}\ (m+n)$ since the nested *for Loop* in the basic Pair-HMM shown in Algorithm 1 has become two *for Loops* at the same level. Part of the major component of this implementation is shown in Figure 4.5.

Algorithm 2 Pseudocode for Across-Diagonal Parallelization

```
1:  procedure PAIR HMM DIAGONAL(R, H)
2:      diagonal_id ← threadIdx.x
3:      final_sum_probabilities ← 0
4:      for i = 1 to |R| do
5:          read_pos ← i − diagonal_id
6:          haplo_pos ← 1 + diagonal_id
7:          if read_pos and haplo_pos within matrix then
8:              update M, I, D
9:          end if
10:     end for
11:     for j = 1 to |H| do
12:         read_pos ← (|R| − 1) − diagonal_id
13:         haplo_pos ← j + diagonal_id
14:         if read_pos and haplo_pos within matrix then
15:             update M, I, D
16:         end if
17:         if read_pos is |R| − 1 then
18:             final_sum_probabilities
                  += (M_{x_pos,y_pos} + I_{x_pos,y_pos})
19:         end if
20:     end for
21: end procedure
```

Take Titan RTX as example, it has up to 72 Streaming Multiprocessor (SMs) available and each SM could take up to 16 active *blocks* or 64 active *warps*. And thus, to make sure we could put as much *warps* into a SM as possible but at the same time reduce the percentage of *threads* in idle, there

need at least four *warps* in each *block*. In our implementation, we design PU to include 64 GPU *threads*.

There are 32 *threads* in each *warp* [20] and we have three *States* for Pair-HMM FA. We need

$$(64\ threads/32\ threads\ per\ warp) * 3\ states = 6\ warps$$

for each pair of alignment and we maximize the possible pairs of alignment that could be deployed simultaneously by using

$$BlockDim = (64, 3, 1)$$

The *GridDim* could be adjusted according to the actual need, for Titan RTX, we can allocate up to

$$72\ SMs * floor(64\ warps/6\ warps\ per\ pair) = 720 \text{ pairs of alignment}$$

on the GPU simultaneously.

```
// Across-Diagonal Parallelization
for (int x_offset = 1; x_offset < curBlock_configuration[0]; x_offset++) {
    for (int offset_factor = 0; offset_factor * unit_length < x_offset; offset_factor++) {
        x_pos = (x_offset - offset_factor * unit_length) - diagonal_id;
        y_pos = (1 + offset_factor * unit_length) + diagonal_id;
        if (x_pos > 0 && y_pos < curBlock_configuration[1]) {
            //Across-State parallelization
            if (states_id == 0) {
                forward_4d(x_pos, y_pos, batch_id, 0) = (prior_3d((x_pos-1), (y_pos-1), batch_id) * (forward_4d((x_pos-1), (y_pos-1),
                batch_id, 0) * transitions_3d(batch_id, matchToMatch, x_pos) +
                            forward_4d((x_pos-1), (y_pos-1), batch_id, 1) * transitions_3d(batch_id, indelToMatch, x_pos) +
                            forward_4d((x_pos-1), (y_pos-1), batch_id, 2) * transitions_3d(batch_id, indelToMatch, x_pos)));
            } else if (states_id == 1) {
                forward_4d(x_pos, y_pos, batch_id, 1) = (forward_4d((x_pos-1), y_pos, batch_id, 0) * transitions_3d(batch_id,
                matchToInsertion, x_pos) + forward_4d((x_pos-1), y_pos, batch_id, 1) * transitions_3d(batch_id, insertionToInsertion,
                x_pos));
            } else {
                forward_4d(x_pos, y_pos, batch_id, 2) = (forward_4d(x_pos, (y_pos-1), batch_id, 0) * transitions_3d(batch_id,
                matchToDeletion, x_pos) + forward_4d(x_pos, (y_pos-1), batch_id, 2) * transitions_3d(batch_id, deletionToDeletion,
                x_pos));
            }
        }
    }
}
```

Figure 4.5: Main Implementation for Computation Parallelization

4.3 Matrices Initialization on GPU

Pairs of DNA *(read-reference)* sequences from the same active region share the same *insertionGOP*, *deletionGOP*, *overallGCP* for generating *transitions* and the same *haplotypeBases*, *readBases*, *readQuals* for *emissions* matrices. We do not want to waste valuable computational resources and power on generating same matrices multiple times. We implement the GPU version of initialization functions for *emissions* and *transitions* matrices in a separate GPU kernel and invoke the initialization kernel only at the first time these pairs of DNA sequences are received by the Pair-HMM FA.

Compared to previous GPU implementations, our Pair-HMM FA flow with matrices initialization on GPUs has two benefits:

1. By transferring a much smaller amount of data from the host to GPU and generating large matrices on GPU, we could significantly reduce the memory traffic. In previous GPU implementations [4], [3], *transitions* and *emissions* matrices with the size of $ReadLength * HaplotypeLength$ and $6 * ReadLength$ correspondingly need to be copied from host to GPU.

   We move the generation of these two matrices onto GPU and only need to copy two Gap Opening Penalties (GOP) arrays, one Gap Continuation Penalties (GCP) array, $readBases$, $readQuals$ and $haplotypeBases$ from host to GPU. These source matrices have the total size of $(5 * ReadLength + HaplotypeLength)$.

   Compared to previous GPU implementations, our implementations reduce the memory traffic by the ratio of

   $$\begin{gathered}\text{(Size of } transitions \text{ + Size of } emissions\text{) / (Size of } GOP/GCP + \\ \text{Size of } Bases/Quals\text{) =} \\ (ReadLength * HaplotypeLength + 6 * ReadLength) \ / \\ (5 * ReadLength + HaplotypeLength)\end{gathered}$$

2. We speed up the matrices initialization process by distributing the workload to lots of GPU threads and each thread is responsible for initializing one cell within the *transitions* and *emissions* matrices.

The initialization process includes three steps: prepare a look-up table named *matchToMatchProb* that holds the pre-calculated *State* match to *State* match transition probability values, generate the full *transitions* matrix and finally initialize the *emissions* matrix.

4.4 Tasks Pipeline with CUDA Streams

CUDA treats kernels invocations and API calls asynchronously and thus it allows the use of multiple GPU streams to overlap workload. These GPU streams virtualize hardware and computational resources on GPU. Programmers do not need to dwell on exactly when a task starts or completes or how to distribute workloads across a GPU, but do need to focus on resolving logic and data dependencies.

In our Pair-HMM FA, we have **four** types of GPU tasks:

- **HostToDeviceMemCpy(H2D)**: Copy arrays of input data from host to GPU
- **Initialization Kernel**: Generate and initialize *transitions* and *emissions* matrices on GPU global memory
- **Computation Kernel**: Computation for total likelihood using Pair-HMM FA
- **DeviceToHostMemCpy(D2H)**: Copy arrays of results from GPU back to host

With careful design of GPU streams, we could hide the latency for *memory copy* and *initialization* by tasks overlapping shown in Figure 4.6.

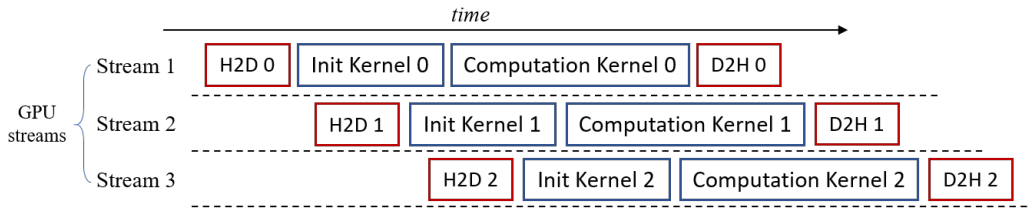

Figure 4.6: Taks Pipeline with CUDA Streams

CUDA provides powerful tools for overlapping GPU activities, including kernel invocations and memory transfers. In CUDA, kernel invocations are

non-blocking by nature. Control is returned to the host CPU once a kernel is configured and starts running on the GPU. For memory transfers, we need to use the `cudaMemcpyAsync` function to copy data asynchronously. GPU streams are created by the `cudaStreamCreate` function and dependencies within a GPU *stream* is guaranteed by the `cudaStreamSynchronize` function.

In our implementation, we utilize aggressive multiple streams strategy, i.e., there will be the same number of alive GPU streams running as the number of batches the GPU is working on. An idle GPU stream will not be allocated with computational resources since all tasks on it have been completed and we trust the CUDA runtime system for properly handling the idle GPU stream.

The number right after each kernel invocation or API call is the batch number GPU is currently working on. The pseudocode for multiple streams Pair-HMM FA is attached in the Appendix: Algorithm 5.

4.5 Memory Strategies for GPU

In our implementation, we use shared memory to store the intermediate anti-diagonal data to avoid frequent and expensive read or write operations from the global memory. On Nvidia GPU, shared memory is much closer to the threads and the operation on shared memory is usually considered as 100x faster than that on global memory.

As discussed, we use PUs to update cells along the anti-diagonal of the *forward_matrix* but we are only interested in the total likelihood at the end of the Pair-HMM forward algorithm (FA). Thus for each pair of sequences to be processed, GPU only needs to memorize the cells along the anti-diagonal right before ones it is working on.

In Figure 4.7, yellow cells have been processed and green ones are those GPU is working on. The updates of green cells only depend on those yellow ones, and thus, GPU *threads* can directly read them out from the shared memory. An example of the dependency is given in Figure 4.7, where a green cell is updated from three yellow cells labeled with number 1, 2 and 3. The dependencies are marked with red arrows.

When the GPU is done with one column of green cells and stepping for-

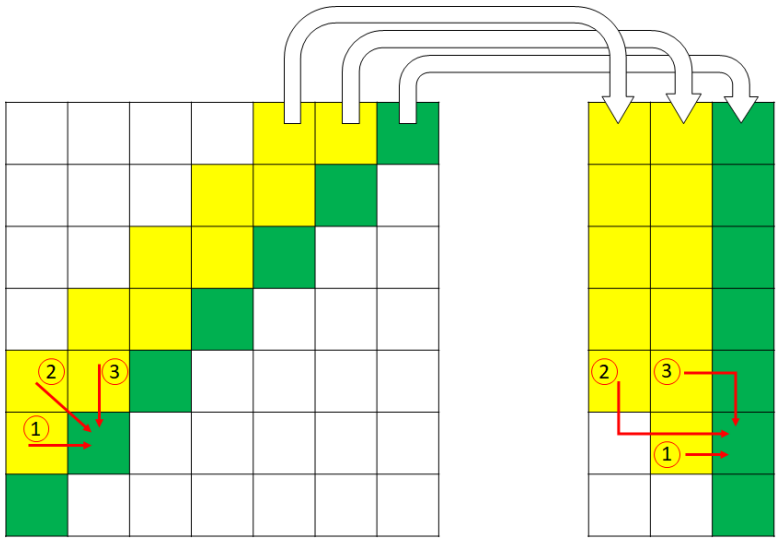

Figure 4.7: Shared Memory Strategy

ward, the shared memory pointer moves to next column and now green cells become the inputs. As the pointer to the shared memory reaches the last column, it will move back to the first column next time. The GPU keeps doing this until the bottom-right cell is updated.

In our case, we just need to allocate a shared memory space of 3×320 for each GPU block to fit the longest anti-diagonal.

4.6 Workload-based Resources Allocation

In a real human genome datasets, the length of haplotype could vary as shown in Figure 4.8, which represents the cumulative probability for different sizes of active region for a fragment of human genome *NA12878*.

As a response to the concerns of wasting memory or computational resources, our implementation includes an optimization on the host side. Two different kernels with sizes in terms of read and haplotype ($\mathbf{320} \times \mathbf{320}$) and ($\mathbf{320} \times \mathbf{640}$) are implemented. An incoming pair of sequences will be handled by the proper kernel that could contain it.

We assume the GATK HaplotypeCaller will not generate alignment with a read sequence longer than 320 and haplotype sequence longer than 640.

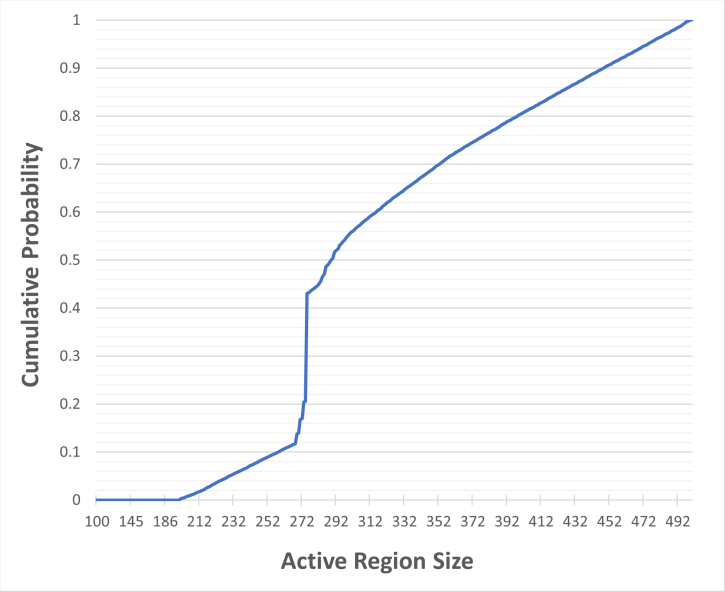

Figure 4.8: Active Region Sizes Distribution

4.7 Other GPU Optimizations

According to the *CUDA C++ Best Practices Guide* [21], we include GPU optimizations recommended for general GPU workload to further improve its performance and robustness as follows:

1. We define constant variables used across GPU kernels by constant memory or compile-time variables to reduce runtime workload.

2. We use shared memory for storing configuration information such as the number of batches, read and reference sequences lengths in shared memory for each GPU block to shorten query time.

3. Matrices data is copied from GPU global to shared memory with coalesced memory access pattern. All matrices and threads are organized in row-major format for two-dimensional matrices.

4. If matrices need to be accessed randomly and violate the memory coalescing, we always copy them from GPU global memory to shared memory first.

5. We fine-tuned the *grids* and *blocks* dimensions for GPU kernels to find out the best combinations for the device we use to deploy our implementation.

CHAPTER 5

EVALUATION AND RESULTS

5.1 Experimental Setup

We set up three different experiments to evaluate our GPU implementations.

1. Generate synthetic data simulating DNA sequences with different sizes. The platform we use for collecting experimental results is Intel Xeon E5-2680 and Nvidia Titan RTX.

2. Measure the performance with "10s" dataset [22] available with the original Java implementation. The platform we use for collecting experimental results is Intel Xeon E5-2695 and Nvidia Tesla V100.

3. Integrate our GPU implementation for Pair-HMM FA into the GATK HaplotypeCaller and run a standard DNA alignment flow with *NA12878* and human genome reference *v37*. We use the same platform as the second experiment.

5.2 Synthetic Dataset

We generate random sequences as inputs with *ReadLength* of 128 and *HaplotypeLength* of 256. By repeating the experiments, we simulate the workload of 10^3, 10^4, 10^5, 10^6 and 10^8 pairs of DNA sequences alignment and compare the results.

The method we used for generating random sequences is shown in Algorithm 3.

To assure a fair comparison is made, we deploy both our implementation and another GPU accelerator from previous work on the same GPU.

Algorithm 3 Pseudocode for Generating Random Sequences

```
1: procedure GENERATE RANDOM SEQUENCE(Length)
2:     Initialize:
       srand(time)
       Create a new char array random with Length
3:     for i = 1 to Length do
4:         random[i] ← (char)(rand() % 256)
5:     end for
6:     return random
7: end procedure
```

Titan RTX is a NVIDIA GPU and built on Turing architecture. It is running on a clock of 1350 MHZ, containing 72 SMs and 4608 single precision CUDA Cores. Its Thermal Design Power (TDP) is 230 Watts.

The throughput performance for the DNA alignment problem is measured by the GCUPS (Giga Cell Updates Per Second), formulated as

$$GCUPS = \frac{\sum_i m_i \times n_i}{t \times 10^6}$$

where t is time of execution measured in milliseconds, m_i and n_i are the length of ith read and haplotype sequences correspondingly. Then we sum the number of cells contributed by all pairs of sequences processed during the execution by Algorithm 4.

Algorithm 4 Pseudocode for Experiments

```
1: for i = 1 to no_of_pairs / 100 do
2:     Declare a < Vector > input for storing batch input
3:     for j = 1 to 100 do
4:         Create a newInput structure
5:         Filled in readQuals, insertionGOP,
       deletionGOP, overallGCP, haplotypeBases
       and readBases with random sequences generated by procedure
       GENERATE RANDOM SEQUENCE.
6:         input.push_back(newInput)
7:     end for
8:     run implementation 1 with input
9:     run implementation 2 (this paper) with input
10: end for
```

We use the *high_resolution_clock* class provided by the C++ standard library to compute the time elapsed. The results are shown in Table 5.1.

Table 5.1: Synthetic Dataset Results Compared with Previous GPU Work

Number of Pairs	10^3	10^4	10^5	10^6	10^8
This thesis (ms)	1.04776	6.70163	65.7450	636.645	66035.5
Previous work (ms)	1.33900	8.34423	81.6554	834.273	85311.9

After calculating the GCUPS according to the time spent on each dataset, we find our GPU implementation achieved the highest throughput of 51.47 GCUPS with 10^6 pairs of DNA sequences.

The GPU implementation from previous work [3] has the performance of 39.28 GCUPS running the same synthetic dataset on the Titan RTX platform. We achieve a 1.31x speedup compared to the previous GPU implementation [3], [4] for Pair-HMM FA running on the synthetic dataset.

5.3 Real Dataset

To evaluate the performance of our GPU implementation on the Pair-HMM FA, we utilize a dataset [22] extracted from a real DNA *read-reference* alignment. The dataset is commonly used for evaluation in previous works and has been given the name of "10s"dataset since its Pair-HMM FA takes about 10 seconds to complete on the original Java baseline.

We present the results of our implementation running on the "10s" dataset [22] and compare it with other implementations [23], [24] as in Table 5.2.

Table 5.2: Performance Comparison with "10s" Dataset

Platform	**Runtime (ms)**	**Speedup**
Java on CPU	10800	1x
C++ Baseline	1267	9x
Intel Xeon AVX Single Core	138	78x
Nvidia K40 GPU	70	154x
Nvidia Tesla V100 GPU (our work)	**16.3**	**663x**
Intel Xeon 24 Cores	15	720x
Altera OpenCL (Stratix V)	8.3	1301x
PE Ring structure (Stratix V)	5.3	2038x

Our GPU implementation shows significant performance improved from previous GPU work [3], [4], but does not beat FPGA implementation in this

small dataset. FPGA implementations win in this "10s" dataset because of its low overhead and short latency on arithmetic operations.

In the next experiment, we will show that GPU implementations outrun FPGA ones while processing large dataset batches.

5.4 Integration into GATK HaplotypeCaller

In this experiment, we modify the Pair-HMM FA implementation within GATK 4.2 by replacing the original Java or C++ baseline with our GPU implementation.

The integration requires us to deep dive into the source code of GATK HaplotypeCaller and the Intel Genomics Kernel Library (GKL) [25]. We first modify the flag `isAvxSupported` within Intel GKL to switch the runtime program entry for the Pair-HMM FA from Java or C++ baseline to our GPU implementation. We also need to compile our implementation into C++ shared object file (*.so) and dynamic library file (*.dylib) that could be loaded by the GATK HapolotypeCaller. The last step is the compilation of the GATK 4.2 from its source files [26] and the replacement of modified files inside the GATK build.

For testing data, we download the digitalized *NA12878* dataset released by the National Library of Medicine (NLM), part of the National Institutes of Health (NIH) and the digitalized reference file *v37* provided by the human 1000 genome project [27].

We then down-sample the original 300x to 60x coverage read groups to reduce input workload from 500 GB to around 100 GB and measure the end-to-end runtime for the GATK HaplotypeCaller to finish the alignment for input read groups (*.bam) and reference (*.fasta) and generate the Genomic Variant Call Format File (*.GVCF) that could be used for following steps in the GATK Best Practices shown in Figure 2.1.

The results of Pair-HMM execution time within the GATK HaplotypeCaller are shown in Table 5.3, our PHMM FA implementation beat all previous FPGA implementations and the GPU work.

Table 5.3: Performance Comparison of our PHMM Implementation Integrated into GATK HaplotypeCaller

Work	Platform	Speedup	Speedup / Power
C++Baseline [28]	(IBM) Power8 CPU	1x	1x
[29]	Intel Xeon Single Core	0.91x	1.33x
[8]	Power8 & Xilinx KU060	2.35x	33.24x
[6]	Arria 10	6.32x	60.04x (TDP=20W)
[3] [4]	Power8 & Nvidia K40	15.51x	12.80x (TDP=235W)
[5]	Power8 & Xilinx XC7VX	14.85x	147.49x
This Work	Power8 & Tesla V100	25.10x	16.32x (TDP=300W)

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this work, we identified the significance of Pair-HMM forward algorithm (FA) and discussed trade-offs between different previous efforts on improving the FA with hardware accelerator in terms of their architectures and implementations. We then proposed a comprehensive solution on GPU that produced the best performance so far.

In Chapter 4, we not only introduced our design and implementation of GPU accelerator for Pair-HMM FA, but also share thoughts as well as common practices that could be used for accelerating algorithms on GPUs in general.

Our work deployed and optimized for Nvidia Titan RTX shows its potentials with the best throughput of 51.47 GCUPS on synthetic datasets. As we integrated our work into the GATK *HaplotypeCaller* and deployed it on Nvidia Tesla V100, we observed a speedup of 25.10x compared to the C++ baseline, which is 60% faster than the state-of-the-art GPU implementation.

We are expecting to open source our work by releasing our implementation as a docker image that could be adopted and deployed on different devices with Nvidia GPUs support. In the future, we would like to explore distributing workload across multiple GPUs to further improve the performance.

REFERENCES

[1] S. Caspar, N. Dubacher, A. Kopps, J. Meienberg, C. Henggeler, and G. Matyas, "Clinical sequencing: From raw data to diagnosis with lifetime value," in *Clinical Genetics*, vol. 93, no. 3, 2018, pp. 508–519.

[2] K. A. Phillips and M. P. Douglas, "The global market for next-generation sequencing tests continues its torrid pace," *The Journal of Precision Medicine*, vol. 4, Oct 2018. [Online]. Available: https://www.thejournalofprecisionmedicine.com/wp-content/uploads/2018/11/Phillips-Online.pdf

[3] S. Ren, K. Bertel, and Z. Al-Ars, "Exploration of alternative GPU implementations of the Pair-HMMs Forward Algorithm," in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 2016, pp. 902–909.

[4] S. Ren, K. Bertels, and Z. Al-Ars, "Efficient acceleration of the Pair-HMMs Forward Algorithm for GATK HaplotypeCaller on graphics processing units," *Evolutionary Bioinformatics*, vol. 14, p. 117693431876054, 2018.

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

[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," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[7] L. van Dam, J. Peltenburg, Z. Al-Ars, and H. P. Hofstee, "An accelerator for posit arithmetic targeting posit level 1 BLAS routines and Pair-HMM," *Proceedings of the Conference for Next Generation Arithmetic 2019*, 2019.

[8] M. Ito and M. Ohara, "A power-efficient FPGA accelerator: Systolic array with cache-coherent interface for Pair-HMM algorithm," in *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX)*, 2016, pp. 1–3.

[9] J. Peltenburg, S. Ren, and Z. Al-Ars, "Maximizing systolic array efficiency to accelerate the Pair-HMM Forward Algorithm," in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 2016, pp. 758–762.

[10] J. M. Zook and et al., "Extensive sequencing of seven human genomes to characterize benchmark reference materials," *Scientific Data*, vol. 3, no. 1, p. 160025, Jun 2016. [Online]. Available: https://doi.org/10.1038/sdata.2016.25

[11] A. G. v. der and B. D. O'Connor, *Genomics in the Cloud: Using Docker, GATK, and WDL in Terra*. O'Reilly Media, 2020.

[12] N. D. Olson, S. P. Lund, R. E. Colman, J. T. Foster, J. W. Sahl, J. M. Schupp, P. Keim, J. B. Morrow, M. L. Salit, J. M. Zook, and et al., "Best practices for evaluating single nucleotide variant calling methods for microbial genomics," *Frontiers in Genetics*, vol. 6, 2015.

[13] D. C. Koboldt, "Best practices for variant calling in clinical sequencing," *Genome Medicine*, vol. 12, no. 1, p. 91, Oct 2020. [Online]. Available: https://doi.org/10.1186/s13073-020-00791-w

[14] GATK Team, "Germline short variant discovery (snps + indels)." [Online]. Available: https://gatk.broadinstitute.org/hc/en-us/articles/360035535932-Germline-short-variant-discovery-SNPs-Indels

[15] J. R. Heldenbrand, S. Baheti, M. A. Bockol, T. M. Drucker, S. N. Hart, M. E. Hudson, R. K. Iyer, M. T. Kalmbach, K. I. Kendig, E. W. Klee, N. R. Mattson, E. D. Wieben, M. Wiepert, D. E. Wildman, and L. S. Mainzer, "Recommendations for performance optimizations when using GATK3.8 and GATK4," *BMC Bioinformatics*, vol. 20, no. 1, p. 557, Nov 2019. [Online]. Available: https://doi.org/10.1186/s12859-019-3169-7

[16] J. Felsenstein and G. A. Churchill, "A Hidden Markov Model approach to variation among sites in rate of evolution." *Molecular Biology and Evolution*, vol. 13, no. 1, pp. 93–104, 01 1996. [Online]. Available: https://doi.org/10.1093/oxfordjournals.molbev.a025575

[17] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[18] M. Vasimuddin, S. Misra, and S. Aluru, "Identification of significant computational building blocks through comprehensive investigation of ngs secondary analysis methods," *bioRxiv*, 2018. [Online]. Available: https://www.biorxiv.org/content/early/2018/07/25/301903

[19] J. L. Gustafson, *The End of Error: Unum Computing.* CRC Press, 2017.

[20] J. Nicholls and D. Kirk, "Appendix c: Graphics and computing GPUs." [Online]. Available: https://booksite.elsevier.com/9780124077263/downloads/advance_contents_and_appendices/appendix_C.pdf

[21] Nvidia Corporation, "CUDA C++ Best Practices Guide." [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

[22] M. Carneiro, "GitHub: Optimization of a Haplotype Pair-HMM class for GPU/FPGA and AVX processing." [Online]. Available: https://github.com/MauricioCarneiro/PairHMM/tree/master/test_data

[23] C. Rauer and N. Finamore, "Accelerating Genomics Research with OpenCL and FPGAs." [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01262-accelerating-genomics-research-with-opencl-and-fpgas.pdf

[24] R. Wertenbroek and Y. Thoma, "Acceleration of the Pair-HMM forward algorithm on FPGA with cloud integration for GATK," in *2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 2019, pp. 534–541.

[25] Intel Corporation, "GitHub: Intel Genomics Kernel Library (GKL)." [Online]. Available: https://github.com/Intel-HLS/GKL

[26] Broad Institute, "GitHub: Genome Analysis Toolkit (GATK) 4." [Online]. Available: https://github.com/broadinstitute/gatk

[27] A. Auton and The 1000 Genomes Project Consortium, "A global reference for human genetic variation," *Nature*, vol. 526, no. 7571, pp. 68–74, Oct 2015. [Online]. Available: https://doi.org/10.1038/nature15393

[28] Y. Cheng and T. Ogasawara, "Speed up HaplotypeCaller on IBM POWER8 systems." [Online]. Available: https://discussions4562.rssing.com/chan-67237868/article1480.html

[29] Intel Corporation, "Replication Recipe for HaplotypeCaller Tool Runtimes." [Online]. Available: https://www.intel.ca/content/dam/www/public/us/en/documents/pdf/10380-2%20Recipe-GATK_021615.pdf

APPENDIX

Algorithm 5 Pseudocode for Multiple Streams Pipeline

```
1: procedure GPU PAIR HMM(batch_size, inputs, outputs)
2:     cudaStream ← 1
3:     HostToDeviceAsync < cudaStream > (inputs[cudaStream])
4:     cudaStreamSync(cudaStream)
5:     init_kernel < cudaStream > ()
6:     while cudaStream <= batch_size do
7:         cudaStreamSync(cudaStream)
8:         compute_kernel < cudaStream > ()
9:         cudaStreamSync(cudaStream)
10:        DeviceToHostAsync < cudaStream > (outputs[cudaStream])
11:        cudaStream + = 1
12:        HostToDeviceAsync < cudaStream > (inputs[cudaStream])
13:        cudaStreamSync(cudaStream)
14:        init_kernel < cudaStream > ()
15:    end while
16:    cudaStreamSync(cudaStream)
17:    compute_kernel < cudaStream > ()
18:    cudaStreamSync(cudaStream)
19:    DeviceToHostAsync < cudaStream > (outputs[cudaStream])
20: end procedure
```