Accelerating DNA Variant Calling Algorithms on High Performance Computing Systems

Ren, Shanshan

DOI
10.4233/uuid:1752b8ce-631b-4127-91c9-92538e34a13b

Publication date
2018

Document Version
Final published version

Citation (APA)
Ren, S. (2018). *Accelerating DNA Variant Calling Algorithms on High Performance Computing Systems*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:1752b8ce-631b-4127-91c9-92538e34a13b

ACCELERATING DNA VARIANT CALLING ALGORITHMS ON HIGH PERFORMANCE COMPUTING SYSTEMS

ACCELERATING DNA VARIANT CALLING ALGORITHMS ON HIGH PERFORMANCE COMPUTING SYSTEMS

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus prof.dr.ir. T.H.J.J. van der Hagen
chair of the Board for Doctorates
to be defended publicly on
Monday 17 December 2018 at 10:00 o'clock

by

Shanshan REN

Master of Science in Computer Science and Technology,
National University of Defense Technology, China
born in Sichuan, China

This dissertation has been approved by the promotors:
Dr. ir. Z. Al-Ars
Prof. dr. ir. K.L.M. Bertels

Composition of the doctoral committee:

Rector Magnificus,	chairman
Dr. ir. Z. Al-Ars,	Delft University of Technology, promotor
Prof. dr. ir. K.L.M. Bertels,	Delft University of Technology, promotor

Independent members:

Prof. dr. F. Baas,	Leiden University Medical Center
Prof. dr. Y. Dou,	National University of Defense Technology, China
Dr. ir. J. de Ridder,	University Medical Center Utrecht
Prof. dr. ir. C. Vuik,	Delft University of Technology
Prof. dr. ir. M.J.T. Reinders,	Delft University of Technology

The research described in this thesis was performed in the Quantum Computer Engineering Department. This work was supported by the China Scholarship Council (CSC) and Delft University of Technology (TUDelft).

ISBN 978-94-028-1318-0

Keywords: Pair-HMMs forward, sequence alignment with traceback, de Brujin graph construction, GPU acceleration, FPGA acceleration

Printed in the Netherlands

Dedicated to my parents and husband

ACKNOWLEDGEMENTS

I would like to thank all the nice people I encountered in the past five years. As we all know, it is not easy to be a PhD student, especially at an overseas university. I still remember how nervous and shy I was on the first day I arrived in the Netherlands. Thank you all for your company, making my life in the Netherlands full of happiness. Without your support and help, I would not have been able to finish this thesis.

First, I would like to thank my promotor Prof. Koen Bertels. Dear Koen, thank you so much for providing me an opportunity to start my PhD in the Netherlands. On my first day in the Netherlands, you invited me to attend your inaugural speech, which made me feel so warm and less nervous of my new life. At the beginning of my PhD, I was still unsure if what I wanted to do. Big data was such a new and broad field for me. You helped me to quickly get started and find an interesting topic to focus on. Thank you very much for your encouragement and guidance in my research. As the head of our group, you held many interesting activities almost every month, such as the Christmas party, carting, barbecues, bowling, pasta dinners, girls events and so on, making me feel relaxed and close to other members in our group. The amazing thing was that you are good at these activities and always got the first prize in the games. You not only guided me on how to do research, but also taught me how to enjoy life. Whenever I saw you, I felt happy and positive. Thank you very much for the support and help you gave during the five years.

I would also like to give my thanks to my promotor Dr. Zaid Al-Ars. Dear Zaid, words cannot express my gratitude for your supervision in the five years. When I started my research in bioinformatics, I was totally a newbie since I had no background in genomics. You used an example to let me practise and gave me the opportunity to take courses in Leiden University. Thank you for your patience on my poor oral English. My oral English needed much improvement in the first year, but you could understand my words in our daily discussions. You were always passionate about research, which influenced me so much. When I got stuck, you always encouraged me and helped me find solutions. During our daily discussions, you always gave me good suggestions, taught me how to do research and how to write papers. After discussions with you, I always felt confident again and had the power to continue fighting. Thank you for having faith in me all the time. Thank you for spending so much time revising my papers. Every time I sent my papers to you, you always finished the revision asap. I learned a lot of writing skills from your revision. In addition, you always encouraged me to speak my mind and try new things. I really learned a lot from you.

I want to give my thanks to my office mates Imran Ashraf, Nauman Ahmed, Hamid Mushtaq and Hani Al-Ers. Imran, thank you for introducing me to every member on my first day in the group. You are very friendly and obliging. Nauman and Hamid, thank you for all the discussions we had and all the suggestions you gave me in my research. Hani, thank you for helping me practicing my presentation skills.

I would also like to thank Johan Peltenburg, Vlad-Mihai Sima, Lidwina Tromp, Joyce van Velzen and Erik de Vries. Vlad, thank you for helping me understanding the framework of the Convey machine and the support you provided for my first paper. Johan, thank you for helping me in my research and sharing spicy food with me. Lidwina and Joyce, thank you for giving me help whenever I turned to you. Erik, thank you for keeping my computer without problems. Finally, I would like to thank my fellow PhD students and friends, especially Jintao, Anh, Lei, Mottaqiallah, Mihai, George, Joost, Mahroo, Xiang, Lingling, Yande, Lizhou, Baozhou, He.

I would like to express my sincere thanks to my friends. Special thanks are given to Xiaoqin Ou. I enjoyed so much the time we spent together. Ling Xia, I really miss the month when we lived in the same apartment. You taught me how to enjoy life while working hard. I really admire your courage to face life's trials. Jing Liu, thank you for keeping in touch with me and helped me handle annoying things in China in the five years. Xu Huang, Xu Xie and Yu Xin, thank you for all your support and help. I also want to express my thanks to Yue Zhao, Guanliang Chen, Hou Zhe, Mingjuan Zhao, Tao Lv, Yang Qu, Jie Shen, Jianbing Fang, Yong Guo, Yazhou Yang and many others. You made my life in the Netherlands full of fun.

I am extremely grateful for my parents. You trusted in me and gave support to every plan I had in the last thirty years. Your optimistic attitude to life inflected me, making me bravely face challenges in life. Thank you very much for your endless love.

Last but not least, I would like to say thank you to my husband. Without your love and support, I would not have been able to complete my PhD. Everyday we had a call around thirty minutes in the five years. It is really hard to imagine. I guess this is the only habit I kept in the five years. You raise me up to more than I can be. Thank you a lot.

Shanshan Ren
Changsha, August 2018

SUMMARY

Next generation sequencing (NGS) technologies have transformed the landscape of genomic research. With the significant advances in NGS technologies, DNA sequencing is more affordable and accessible than ever before. Meanwhile, many DNA sequence analysis tools have been developed to derive useful information from the raw sequencing data produced by NGS platforms. However, the massive amount of generated sequencing data poses a great computational challenge, thereby shifting the bottleneck towards the efficiency of the DNA sequence analysis tools. Due to the high computational needs, high performance systems are playing an important role for DNA sequence analysis. Moreover, dedicated hardware, including graphics processing units (GPUs) and field programmable gate arrays (FPGAs), have become important computational resources in many high performance systems.

In this thesis, we use GPUs and FPGAs to accelerate a number of important bioinformatics algorithms. These represent the most computationally intensive algorithms of the GATK HaplotypeCaller (HC), which we use to improve its performance. GATK HC is a widely used DNA sequence analysis tool. By investigating GATK HC, three computationally intensive algorithms are selected, including the de Buijn graph (DBG) construction algorithm for micro-assembly, the pair-HMMs forward algorithm and the semi-global pairwise alignment algorithm. We first propose a novel GPU-based implementation of the DBG construction algorithm for micro-assembly. Compared with the software-only implementation, it achieves a speedup of up to 3x using synthetic datasets and a speedup of up to 2.66x using human genome datasets. We then propose a systolic array design to accelerate the pair-HMMs forward algorithm on FPGAs. Experimental results show that the FPGA-based implementation is up to 67x faster than the software-only implementation. In order to fully utilize the computing resources on FPGAs, we present a model to describe the performance characteristics of the systolic array design. Based on the analysis, we propose a novel architecture to better utilize the computing resources on FPGAs. The implementation achieves up to 90% of the theoretical throughput for a real dataset. Next, we propose several GPU-based implementations of the pair-HMMs forward algorithm. Experimental results show that the GPU-based implementations of the pair-HMMs forward algorithm achieve a speedup of up to 5.47x over existing GPU-based implementations. Finally, we propose to accelerate the semi-global pairwise sequence alignment algorithm with traceback to obtain the optimal alignment on GPUs. Experimental results show that the GPU-based implementation is up to 14.14x faster than the software-only implementation.

After accelerating these algorithms on GPUs and FPGAs, we integrate two GPU-based implementations into GATK HC. We first integrate the GPU-based implementation of the pair-HMMs forward algorithm into GATK HC. In single-threaded mode, the GPU-based GATK HC implementation is 1.71x faster than the baseline GATK HC implementation. For multi-process mode, a load-balanced multi-process optimization is proposed

to ensure a more equal distribution of computation load between different processes. The GPU-based GATK HC implementation achieves up to 2.04x in load-balanced multi-process mode over the baseline GATK HC implementation in non-load-balanced multi-process mode. Next, we additionally integrated the GPU-based implementation of the semi-global alignment algorithm into the GATK HC. Experimental results shown that this implementation is 2.3x faster than the baseline GATK HC implementation in single-thread mode.

SAMENVATTING

Nieuwe generatie sequencing (NGS) technologieën hebben het landschap van genomisch onderzoek getransformeerd. De aanzienlijke vooruitgang in NGS-technologieën heeft DNA-sequencing goedkoper en toegankelijker gemaakt. Tegelijkertijd zijn veel programma's voor DNA-sequentieanalyse ontwikkeld om bruikbare informatie af te leiden uit de sequentiegegevens die zijn geproduceerd door NGS-systemen. De enorme hoeveelheid gegenereerde sequentiegegevens vormt echter een grote rekenkracht uitdaging, waardoor het knelpunt wordt verschoven naar de efficiëntie van de DNA-sequentieanalyse programma's. Vanwege de hoge reken behoeften spelen computersystemen met hoge rekenkracht een belangrijke rol bij DNA-sequentieanalyse. Bovendien worden speciale hardware systemen, waaronder grafische kaarten (GPU's) en veldprogrammeerbare poortmatrixen (FPGA's), vaak toegepast om de berekeningen te versnellen.

In dit proefschrift gebruiken we GPU's en FPGA's om een aantal belangrijke bioinformatica-algoritmen te versnellen. Dit zijn de meest rekenintensieve algoritmen van de GATK HaplotypeCaller (HC), een veel gebruikt analyse-programma voor DNA-sequenties. We hebben drie rekenintensieve algoritmen van GATK HC geselecteerd, waaronder de Buijn-diagram (DBG) constructie-algoritme voor micro-assembly, pair-HMMs voorwaarts-algoritme en semi-globale paarsgewijze alignment algoritme. We stellen eerst een nieuwe GPU-gebaseerde implementatie voor van het DBG constructie-algoritme voor micro-assembly. Vergeleken met de originele implementatie, bereikt onze GPU-implementatie een snelheid van maximaal 3x voor synthetische datasets en een snelheid van maximaal 2,66x voor DNA-gegevens van menselijk genoom. Vervolgens stellen we een systolisch array ontwerp voor om het pair-HMMs voorwaarts-algoritme op FPGA's te versnellen. Experimentele resultaten laten zien dat de FPGA-implementatie tot 67x sneller is dan de originele implementatie. Om de FPGA's volledig te benutten, presenteren we een model om de prestatiekenmerken van het systolische array ontwerp te beschrijven. Op basis hiervan stellen we een nieuwe architectuur voor om de FPGA's beter te benutten. Deze implementatie behaalt tot 90% van de theoretische prestatie voor een echte dataset. Vervolgens stellen we verschillende GPU-gebaseerde implementaties voor van het pair-HMMs voorwaarts-algoritme. Experimentele resultaten tonen aan dat de GPU implementaties van het pair-HMMs voorwaarts-algoritme een versnelling van 5.47x behalen ten opzichte van bestaande GPU implementaties. Ten slotte stellen we voor om het semi-globale paarsgewijze alignment algoritme te versnellen om de optimale alignment op GPU's te verkrijgen. Experimentele resultaten laten zien dat de GPU implementatie tot 14.14x sneller is dan de originele implementatie.

Na het versnellen van deze algoritmen op GPU's en FPGA's, integreren we twee GPU-gebaseerde implementaties in GATK HC. We integreren eerst de GPU implementatie van het pair-HMMs voorwaarts-algoritme in GATK HC. Deze GPU GATK HC implementa-

tie is 1,71x sneller dan de originele GATK HC. Daarna wordt deze implementatie verder geoptimaliseerd met een gebalanceerde multi-procesmethode om een gelijke verdeling van de rekenkracht tussen de verschillende processen te waarborgen. Deze GPU GATK HC implementatie haalt maximaal 2,04x ten opzichte van de originele GATK HC. Vervolgens hebben we de GPU implementatie van het semi-globale paarsgewijze alignment algoritme in GATK HC geïntegreerd. Experimentele resultaten laten zien dat deze implementatie 2,3x sneller is dan de originele GATK HC.

CONTENTS

1

INTRODUCTION

Next generation sequencing technologies (NGS) provide a high-throughput and cost-effective sequencing method, bringing about a revolution in genomics research and creating vast opportunities for profound understanding of genetics of many species, especially the human genome.

The data generated by NGS platforms, which consists of billions of short DNA fragments, cannot be directly used by biologists. Complex statistical models and sophisticated genomic analysis tools are proposed to turn raw sequencing data into biologically meaningful information. However, the massive amount of sequencing data poses increasing pressure on the computationally intensive algorithms used in genomic analysis tools. In this thesis, we provide solutions to optimize these computationally intensive algorithms on high performance computing systems.

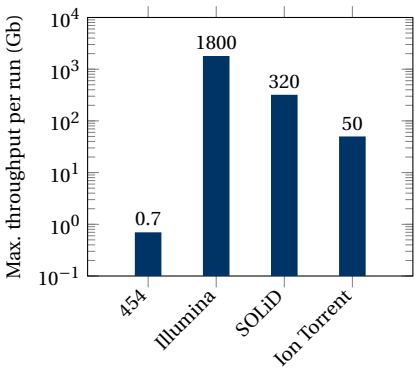
In this chapter, Section 1.1 discusses the background and related work. Section 1.2 introduces the challenges and limitations of accelerating these algorithms. Section 1.3 defines our contributions. Finally, Section 1.4 describes the thesis organization.

1.1. BACKGROUND AND RELATED WORK

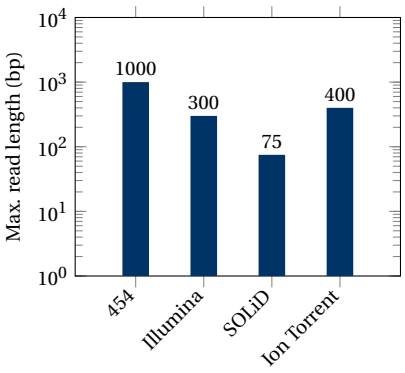
1.1.1. NEXT GENERATION SEQUENCING

DNA sequencing is the process of determining the order of nucleotides in a DNA molecule. There are in total four kinds of nucleotides in a DNA molecule, which can be distinguished using the type of base contained in each nucleotide. Thus, the objective of DNA sequencing is to determine the order of these four nucleotides base types: adenine (A), cytosine (C), guanine (G) and thymine (T) in a DNA molecule.

In the mid-1970s, first generation sequencing technologies referring to two distinct DNA sequencing approaches were discovered by Maxma and Gilbert [1] and Sanger and colleagues [2]. Due to its accuracy, robustness and ease of use, the approach proposed by Sanger and colleagues (the Sanger sequencing approach) became more commonly used to sequence DNA than the approach developed by Maxma and Gilbert (the Maxma-Gilbert approach) [3]. In the next 30 years, the Sanger sequencing approach has been



(a) Maximum throughput of NGS platforms .



(b) Maximum read length of NGS platforms

Figure 1.1: Maximum throughput and read length of NGS platforms

improved by multiple techniques [4–7].

Since 2005, NGS platforms have been released by several manufactures and replaced the Sanger sequencing platforms in many sequencing projects. 454 Life Sciences (now Roche), Solexa/Illumina and Applied Biosystems (now Thermo Fisher Scientific) released their first NGS platform in 2005 [8], 2006 and 2007 [9], respectively. In the following three years, several new NGS platforms were developed by the three manufactures. Moreover, Ion Torrent (now Thermo Fisher Scientific) introduced its first NSG platform (Personal Genome Machine, PGM) in 2010 and launched the Ion Proton in 2012.

Different NSG platforms exploited different sequencing technologies, which include Roche 454 sequencing, SOLiD sequencing, Illumina sequencing, and Ion torrent Proton/PGM sequencing. However, compared with the traditional Sanger sequencing technology, these NGS technologies share three major characteristics.

1. All NGS platforms perform sequencing of billions of short DNA fragments (referred to as reads) in parallel, leading to a dramatic increase of throughput of each instrument run. Fig. 1.1a shows the maximum throughput of different NGS platforms, which are significantly bigger than that of the Sanger sequencing platforms. In addition, due to the process of parallel sequencing data, NGS is also referred to as massively parallel sequencing.

2. The length of reads produced by NGS platforms is short. Fig. 1.1b shows the maximum length of reads produced by different NGS platforms, which is shorter than that of the Sanger sequencing platforms (1000~1200 bp).

3. The sequencing cost of NGS technologies is much lower than that of the Sanger technology. As shown in Fig. 1.2, the cost of sequencing a human genome decreased greatly in 2008 due to the replacement of the Sanger technology with NGS technologies in sequencing centers.

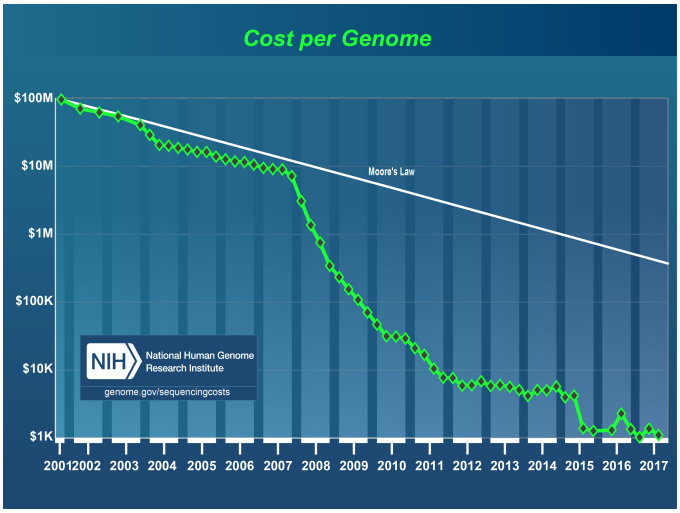


Figure 1.2: Changes in cost per genome from 2001 to 2017 [10]. The green line presents the nature of the reductions in DNA sequencing costs, while the white line presents hypothetical data reflecting Moore's Law

In contrast to traditional Sanger technology, NGS technologies are capable of generating massive DNA sequencing data in a short time and at a low cost, making DNA sequencing more affordable and accessible than ever before. The total amount of sequencing data has doubled approximately every seven months over the past decade reaching a worldwide sequencing capacity of 35 petabases per year in 2015. This capacity is estimated to reach one zettabase per year by 2025 if the growth continues at its current rate [11].

NGS technologies open a new era for genomic research. In particular, NGS technologies provide great opportunities to understand human genetics and investigate the influence of genotypes on diseases, which leads to new discoveries in disease diagnosis and realizes the promise of personalized therapies. In order to derive useful information from the raw sequencing data produced by NGS platforms, many applications for DNA sequence analysis have been developing at an unprecedented rate in the last decade, such as sequence alignment, variant discovery, de novo assembly of new genomes, DNA methylation analysis and so on.

1.1.2. HIGH PERFORMANCE COMPUTING SYSTEMS AND APPLICATIONS

The analysis of the huge amounts of DNA sequencing data is a computational challenge. For example, the sequencing data generated per run by a NGS platform, which is up to 6 billion reads, requires more than 4 days to be processed by an alignment tool on a single 16-core machine [12]. Typical laptop or desktop computers do not satisfy the performance requirements of DNA sequence analysis. In order to address this computational challenge, high performance computing systems are used to perform DNA sequence analysis [13].

High performance computing systems, including server-class machines, supercom-

puters, clusters and cloud computing environments, are known for their high processing capacity. Traditionally, genomics applications mainly depended on CPUs as the main computing resource responsible to process data. With the development of hardware technologies and parallel programming languages, dedicated hardware, such as graphics processing units (GPUs) and field programmable gate arrays (FPGAs), have also been adopted as important computing resources in many high performance computing systems.

GPUs are originally designed to process graphics and images in computers. Due to their highly parallel structure containing thousands of small cores on a single chip, modern GPUs are more efficient in processing highly-parallel computationally intensive algorithms than the conventional CPUs. With the development of parallel programming languages, such as CUDA, modern GPUs are widely used as accelerators to perform general purposed computing instead of only processing computer graphics, leading to dramatic performance improvement. In 2011, the Tianhe-1A became the fastest supercomputer in the world as a result of including up to 7168 NVIDIA Tesla M2050 GPUs [14]. Since then, many supercomputers use GPUs to improve their performance. In 2018, five of the world's seven fastest systems were powered by GPU [15]. Besides, clusters and cloud computing environments (in which the computing nodes are equipped with GPUs) have attracted increasing attentions from scientific computing in multiple domains [16, 17].

FPGAs are semiconductor devices designed to be configured after manufacturing. They contain a matrix of millions of configurable logic blocks (CLBs) connected via configurable interconnects. Since all the logic blocks can be programmed to run in parallel, modern FPGAs can offer massive parallelism to perform computationally intensive algorithms, resulting in the deployment of FPGAs in multiple high performance computing systems. Convey HC-1 [18], a hybrid core server that couples a standard multi-core Intel Xeon processor with a FPGA-based reconfigurable co-processor, was proposed in 2010. In 2015, SRC Computers announced the Saturn 1 Server, a dynamically reconfigurable server relying extensively on FPGAs [19]. Moreover, both IBM and CRAY released systems accelerated both by GPUs and FPGAs [20, 21]. Furthermore, clusters and cloud computing environments powered by FPGAs are a new trend for scientific computing [22, 23].

Due to high performance computing system architectures including CPUs, GPUs and FPGAs, there are two ways to improve the performance of DNA sequence analysis. One way is to distribute workloads across many compute nodes with big data technologies including Spark [24], Hadoop Map-Reduce [25], MPI [26], OpenMP [27] and so on. [28] presents a number of bioinformatics applications running on clusters and in cloud computing environments, such as SparkSeq [29] and CloudBurst [30]. The second way to improve the performance of DAN analysis is to accelerate the time-consuming algorithms of sequence analysis tools on GPUs and FPGAs to achieve a large speedup. [31] gives an overview of about twenty GPU-based sequence alignment tools, such as SW# [32] and SOAP3 [33]. [34] shows multiple tools accelerated by FPGAs. Further examples include FHAST [35] and Tera-Blast [36].

In this thesis, we use GPUs and FPGAs to accelerate the computationally intensive algorithms of GATK HaplotypeCaller (HC).

1.1.3. GATK HAPLOTYPECALLER

Genome Analysis Toolkit (GATK) is developed at the Broad Institute, which supplies a wide variety of tools with a primary focus on variant discovery and genotyping [37]. GATK HC is one of these tools, which is widely used in many large-scale sequencing projects.

Variant discovery is a significant application of DNA sequence analysis, which is used to identify the variants from a given sample genome. The most commonly used approach to identify variants is the alignment-based variant discovery approach. It first aligns the sequencing dataset of a sample genome produced by a NGS platform to a reference genome using alignment tools, such as BWA [38]. It then compares the aligned dataset to the reference genome and extracts the genome positions where the sample genome differs from the reference genome using variant callers, such as GATK HC. The variants found through this approach include single nucleotide variations (snvs), small insertions/deletions (indels) and structural variations (svs). The accuracy of detecting such variants largely depends on the accuracy of the alignment step. However, the alignment step is biased by sequencing error and genome characteristics such as repetitive regions [39]. Moreover, reads with indels are easily misaligned during the alignment step [40], leading to low accuracy of indel detection.

In order to improve variant detection accuracy, especially the accuracy of indel detection, GATK HC employs micro-assembly to correct the misalignment errors. Micro-assembly is to assemble reads aligned to a certain region of the reference genome into a long DNA sequence covering this region, which is referred to as haplotype. Haplotypes are then used in GATK HC to identify variants.

The workflow of GATK HC is quite different from many variant callers, which is divided into the following four main steps [41]:

1. **Define active regions**—Active regions are determined based on the presence of significant evidence for variation. The following steps only operate on the active regions and ignore the inactive regions.

2. **Determine haplotypes**—For each active region, a de Bruijn-like graph is built to reassemble the active region and a list of haplotypes is determined based on the graph. This process is referred as to micro-assembly or local assembly. Then each haplotype is realigned against the reference sequence using the semi-global pairwise sequence alignment algorithm in order to identify potential variant sites.

3. **Determine likelihoods of the haplotypes**—For each active region, a pairwise alignment of each read against each haplotype is performed using the pair-HMMs forward algorithm, which produces a matrix of likelihoods of haplotypes given the reads. Each read is then realigned to the haplotype which has the maximum likelihood using the semi-global pairwise sequence alignment algorithm.

4. **Assign genotypes**—For each potential variant site, Bayes' rule is applied to calculate the likelihoods of each genotype using the likelihoods of haplotypes given the reads. The genotype with the largest likelihoods is selected.

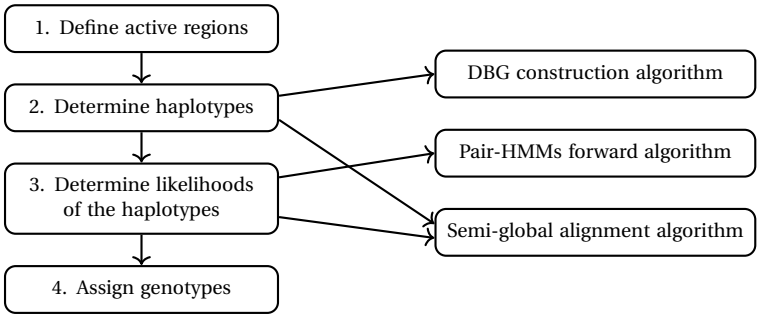


Figure 1.3: Computationally intensive algorithms used in step 2 and step 3 of the workflow of GATK HC

In the workflow of GATK HC, there are three computationally intensive algorithms, which are de Bruijn graph (DBG) construction algorithm for micro-assembly, pair-HMMs forward algorithm and semi-global pairwise sequence alignment algorithm. They are used in step 2 and step 3 of the workflow of GATK HC, as shown in Fig.1.3.

The DBG construction algorithm for micro-assembly uses reads aligned to an active region to construct a de Bruijn graph, which includes two steps: (a) Each read is first decomposed into subsequences of k consecutive bases, referred to as k-mers; (b) The k-mers and overlap relationships between k-mers are then used to construct a de Bruijn graph. One of the challenges of genome assembly is repeats, which would cause cycles in the graph. Unlike other DBG construction algorithms, the DBG construction algorithm in GATK HC handles repeats in a different way. In GATK HC, k-mers are classified into two groups: unique k-mers and repeat k-mers. If a k-mer occurs twice or more than twice in a read, it is a repeat k-mer; otherwise, it is a unique k-mer. During graph construction, unique k-mers are collapsed into single nodes, while repeat k-mers are not collapsed into single nodes. In this way, some cycles in the graph caused by repeats are avoided.

The pair-HMMs (pair hidden Markov models) forward algorithm is used to calculate the overall alignment probability of two sequences by summing overall the alignment probability of all possible alignments of the two sequences. Pair-HMMs are evolved from the basic HMMs. In GATK HC, the pair-HMMs forward algorithm is to calculate the overall alignment probability of a pair of read and haplotype, which is executed millions of times for a typical dataset. The algorithm is a dynamic programming algorithm with a computational complexity of $O(nm)$ (n and m are the lengths of read and haplotype, respectively), which is very large for long sequences. This drawback influences the performance of GATK HC.

The semi-global pairwise sequence alignment algorithm in GATK HC is used to get the optimal semi-global alignments of two sequences. It is implemented in two steps: (a) a dynamic programming kernel is performed to calculate the score matrix and the backtracking matrix; (2) a traceback (or backtracking) kernel is performed to find the optimal alignment by using the backtracking matrix. The computational complexity of the dynamic programming kernel in the first step is $O(nm)$ (n and m are the lengths of two sequences, respectively). Although there has been much research done to reduce

the execution time of pairwise alignment algorithms, most of them mainly focus on the first step and do not pay attention to the second step.

1.2. MOTIVATION AND CHALLENGES

With the use of micro-assembly, GATK HC has a higher accuracy of detecting variants than many other variant callers. However, this comes at the cost of longer execution time, which would limit the feasibility of GATK HC in many situations. For example, [42] compares the runtime of five variant callers, including GATK HC, GATK UnifiedGenotyper, SAMtools, Platypus and Fuwa, using a whole-genome sequencing data as input. Results show that GATK HC is the slowest among the five variant caller, which ran for two days and a half.

In order to reduce the execution time of GATK HC, Intel processors and IBM POWER processors both exploited vector instructions to speed up the pair-HMMs forward algorithm [43, 44] in 2014. In July of 2017, GATK 3.8 was released, which includes the FPGA-based implementation of the pair-HMMs forward algorithm [45]. However, the FPGA-based implementation in GATK 3.8 is fairly experimental. Moreover, multiple papers propose to exploit Spark to optimize its performance, such as [46] and [47]. Furthermore, in early 2018, the Broad Institute released GATK 4.0 [48], which exploits Spark to improve its performance. GATK 4.0 contains both Spark and non-Spark implementations of many tools. In GATK 4.0, GATK HC can either run on a local machine or run in a massively parallel way on a cluster or in the cloud computing environment.

In this thesis, in order to improve the performance of GATK HC, we use GPUs and FPGAs to accelerate the three computationally intensive algorithms of GATK HC: the DBG construction algorithm for micro-assembly, the pair-HMMs forward algorithm and the semi-global pairwise sequence alignment algorithm. In this thesis, we focus on GATK 3.x, including GATK 3.2-3.7, which are released from July of 2014 to December of 2016. The implementation of GATK HC in GATK 3.x does not change much. While our aim is to improve the performance of GATK HC, the techniques and methods presented in this thesis can be extended to other DNA analysis tools.

During this work, we need to address many challenges, which can be divided into two groups: (a) Common challenges shared by the three algorithms; (b) Challenges specific to each one of the algorithms. We first discuss the common challenges, which are presented below:

- The first challenge is related to the effort needed to understand the source code of GATK HC. Understanding the source code of GATK HC is necessary since it is the basis starting point for acceleration in this thesis. Due to the many tools included in GATK, the amount of the source code is large, increasing the difficulty to read and understand the code. We first used several Java profiling tools, including YourKit [49], perf [50] and Flame Graphs [51], to have a global understanding of the execution paths of GATK HC and find the time-consuming kernels. We then inspected the source code carefully with the help of the online documents supplied by GATK.

- Another challenge is related to the need to modify the code in order to produce the input datasets of the three algorithms, which are used to compare the perfor-

mance of the proposed implementations of the three algorithms with the original implementations. Moreover, it is necessary to investigate the characteristics of these input datasets. By using these characteristics, we can optimize the implementations of the three algorithms in the way most suitable for GATK HC.

- It is hard to integrate the GPU-based and FPGA-based implementations of the three algorithms into GATK HC. There are two limitations. One is that GATK is programmed using the Java language, but the Java program cannot directly launch the programming code of FPGAs and GPUs. In order to address this limitation, we need to include solutions provided by packages and interfaces, such as JNI (Java Native Interface) and JCuda [52]. The other limitation is that the size of intermediate input of the algorithms should be large enough to fully utilize the computing sources on GPUs and FPGAs. In order to address this limitation, we need to adapt the source code of GATK HC.

For the acceleration of each algorithm, we face specific challenges, which are discussed below:

- **DBG construction algorithm for micro-assembly**—Unlike other DBG construction algorithms, which collapse repeat k-mers into single nodes, the DBG construction algorithm for micro-assembly in GATK HC maps repeat k-mers to multiple nodes. Thus, previous research published on accelerating DBG construction algorithms is not suitable for the DBG construction algorithm for micro-assembly in GATK HC.

- **Pair-HMMs forward algorithm**—At the time we started to accelerate the pair-HMMs forward algorithm, there was no research on the acceleration of this algorithm on FPGAs or GPUs. Although similar HMM-based algorithms used in the field of computational biology have been accelerated on FPGAs and GPUs, this algorithm is different from previously published ones. Moreover, we have to focus on the characteristics of the input datasets when designing the implementation.

- **Semi-global pairwise sequence alignment algorithm**—Existing GPU/FPGA accelerated implementations mainly focus on calculating the optimal alignment score and omit identifying the optimal alignment itself. Moreover, when we design the implementation of semi-global alignment algorithm for GATK HC, the characteristics of the input datasets should be taken into consideration.

1.3. OUR CONTRIBUTION

We can sum up our contributions in this thesis as follows.

1. We propose a GPU-based DBG construction algorithm for micro-assembly in GATK HC. Compared with the software-only implementation, it achieves a speedup of up to 3x using synthetic datasets and a speedup of up to 2.66x using human genome datasets.

2. We propose a systolic array design to accelerate the pair-HMMs forward algorithm on FPGAs. Experimental results show that the FPGA-based implementation is up to 67x faster than the software-only implementation.

3. We model the performance characteristics of the systolic array design of the pair-HMMs forward algorithm on FPGAs, and propose a novel architecture that allows the computational units to continuously perform useful work on the input data. The implementation achieves up to 90% of the theoretical throughput for a real dataset.

4. We propose several GPU-based implementations of the pair-HMMs forward algorithm. Experimental results show that the GPU-based implementations of the pair-HMMs forward algorithm achieve a speedup of up to 5.47x over existing GPU-based implementations.

5. One of GPU-based implementations of the pair-HMMs forward algorithm is integrated into GATK HC. In single-threaded mode, the GPU-based GATK HC is 1.71x faster than the baseline implementation.

6. For multi-process mode, a load-balanced multi-process optimization to ensure a more equal distribution of computation load between different processes is proposed. The GPU-based GATK HC implementation achieves up to 2.04x in load-balanced multi-process mode over the baseline GATK HC implementation in non-load-balanced multi-process mode.

7. We propose to accelerate the semi-global pairwise sequence alignment algorithm with traceback to obtain the optimal alignment on GPUs. Experimental results show that the GPU-based implementation is up to 14.14x faster than the software-only implementation.

8. The GATK HC integrated with the GPU-based implementations of the semi-global alignment algorithm with traceback and the pair-HMMs forward algorithm is 2.3x faster than the baseline GATK HC implementation.

1.4. THESIS ORGANIZATION

This thesis consists of six chapters. After introducing the background in Chapter 1, each of the remaining chapters resolves one research question, which is self-contained and can be read independently of others. Their relationships are presented in Fig. 1.4. The thesis is organized as follows.

In **Chapter 2**, we present the GPU-based implementation of the DBG construction algorithm for micro-assembly in GATK HC.

In **Chapter 3**, we show the FPGA-based implementations of the pair-HMMs forward algorithm.

In **Chapter 4**, we present the GPU-based implementations of the pair-HMMs forward algorithm and the load-balanced multi-process optimization for the GPU-based

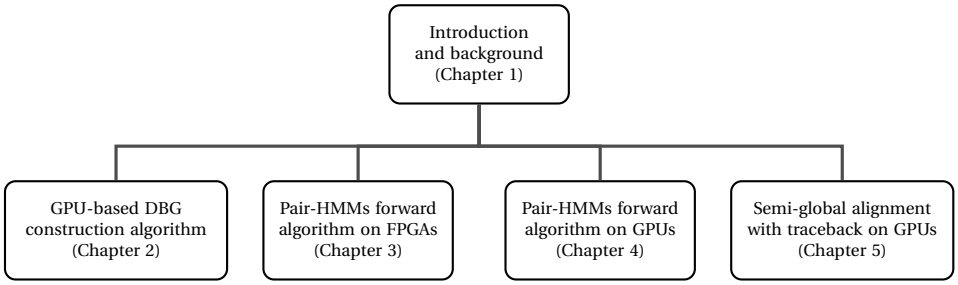


Figure 1.4: Overview of the structure of the main work presented in this thesis

GATK HC.

In **Chapter 5**, we introduce the GPU-based implementation of the semi-global pairwise sequence alignment algorithm.

In **Chapter 6**, we conclude this thesis by summarizing the main contributions and discuss the limitations and possible directions for future research.

REFERENCES

[1] A. M. Maxam and W. Gilbert, *A new method for sequencing dna,* Proceedings of the National Academy of Sciences **74**, 560 (1977), http://www.pnas.org/content/74/2/560.full.pdf .

[2] F. Sanger, S. Nicklen, and A. R. Coulson, *DNA sequencing with chain-terminating inhibitors,* Proc. Natl. Acad. Sci. U.S.A. **74**, 5463 (1977).

[3] J. M. Heather and B. Chain, *The sequence of sequencers: The history of sequencing dna,* Genomics **107**, 1 (2016).

[4] W. Ansorge, B. S. Sproat, J. Stegemann, and C. Schwager, *A non-radioactive automated method for dna sequence determination,* Journal of Biochemical and Biophysical Methods **13**, 315 (1986).

[5] J. Prober, G. Trainor, R. Dam, F. Hobbs, C. Robertson, R. Zagursky, A. Cocuzza, M. Jensen, and K. Baumeister, *A system for rapid dna sequencing with fluorescent chain-terminating dideoxynucleotides,* Science **238**, 336 (1987), cited By 615.

[6] J. A. Luckey, H. Drossman, A. J. Kostichka, D. A. Mead, J. D'Cunha, T. B. Norris, and L. M. Smith, *High speed dna sequencing by capillary electrophoresis,* Nucleic Acids Research **18**, 4417 (1990), /oup/backfile/content$_public/journal/nar$/18/15/10.1093/nar/18.15.4417/3/18–15–4417.pdf .

[7] H. Swerdlow and R. Gesteland, *Capillary gel electrophoresis for rapid, high resolution dna sequencing,* Nucleic Acids Research **18**, 1415 (1990), /oup/backfile/content$_public/journal/nar$/18/6/10.1093/nar/18.6.1415/2/18 – 6 – 1415.pdf .

[8] M. Margulies, M. Egholm, W. E. Altman, S. Attiya, *et al.*, *Genome sequencing in microfabricated high-density picolitre reactors,* Nature **437**, 376 EP (2005), article.

[9] A. Valouev, J. Ichikawa, T. Tonthat, J. Stuart, S. Ranade, H. Peckham, K. Zeng, J. A. Malek, G. Costa, K. McKernan, A. Sidow, A. Fire, and S. M. Johnson, *A high-resolution, nucleosome position map of C. elegans reveals a lack of universal sequence-dictated positioning,* Genome Res. **18**, 1051 (2008).

[10] K. Wetterstrand, *Dna sequencing costs: Data from the nhgri genome sequencing program (gsp),* www.genome.gov/sequencingcostsdata, accessed February 15, 2018.

[11] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, *Big data: Astronomical or genomical?* Plos Biology **13**, e1002195 (2015).

[12] J. M. Abuín, J. C. Pichel, T. F. Pena, and J. Amigo, *Bigbwa: Approaching the burrows-wheeler aligner to big data technologies.* Bioinformatics **31**, 4003 (2015).

[13] B. Schmidt and A. Hildebrandt, *Next-generation sequencing: big data meets high performance computing.* Drug Discovery Today **22** (2017).

[14] *November 2010,* www.top500.org/lists/2010/11 (), accessed August 14, 2018.

[15] *June 2018,* www.top500.org/lists/2018/06 (), accessed August 14, 2018.

[16] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W. Hwu, *Gpu clusters for high-performance computing,* in *IEEE International Conference on CLUSTER Computing and Workshops* (2009) pp. 1–8.

[17] R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo, *General-purpose computation on gpus for high performance cloud computing,* Concurrency Computation Practice Experience **25**, 1628 (2013).

[18] W. Augustin, V. Heuveline, and J. P. Weiss, *Convey hc-1 hybrid core computer – the potential of fpgas in numerical simulation,* Preprint (2010).

[19] AWS, *Src computers launches saturn 1 server, the first reconfigurable hyperscale server,* https://www.marketwatch.com/press-release/src-computers-launches-saturn-1-server-the-first-reconfigurable-hyperscale-server-2015-05-28, accessed August 14, 2018.

[20] J. Cruickshank, *Power 8, gpus and fpgas for workload acceleration,* http://conferences.gse.org.uk/attachments/presentations/OwhdwZ_1446474154.pdf, accessed August 14, 2018.

[21] *Cray cs500 cluster supercomputer,* https://www.cray.com/products/computing/cs-series/cs500, accessed August 14, 2018.

[22] *Paderborn university will offer intel cpu-fpga cluster for researchers,* https://www.top500.org/news/paderborn-university-will-offer-intel-cpu-fpga-cluster-for-researchers/, accessed August 14, 2018.

[23] *Amazon EC2 F1 instances—run customizable fpgas in the aws cloud,* https://aws.amazon.com/ec2/instance-types/f1/, accessed August 15, 2018.

[24] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, *Spark: cluster computing with working sets,* in *Usenix Conference on Hot Topics in Cloud Computing* (2010) pp. 10–10.

[25] T. White, *Hadoop: The Definitive Guide* (O'Reilly Media, Inc., 2009).

[26] M. P. Forum, *MPI: A Message-Passing Interface Standard* (University of Tennessee, 1994) p. 179.

[27] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. Mcdonald, and R. Menon, *Parallel programming in OpenMP* (Morgan Kaufmann Publishers,, 2001).

[28] J. Luo, W. Min, D. Gopukumar, and Y. Zhao, *Big data application in biomedical research and health care: A literature review,* Biomedical Informatics Insights **8**, 1 (2016).

[29] M. S. Wiewiórka, A. Messina, A. Pacholewska, S. Maffioletti, P. Gawrysiak, and M. J. Okoniewski, *Sparkseq: fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision,* Bioinformatics **30**, 2652 (2014).

[30] M. C. Schatz, *Cloudburst: highly sensitive read mapping with mapreduce.* Bioinformatics **25**, 1363 (2009).

[31] M. S. Nobile, P. Cazzaniga, A. Tangherloni, and D. Besozzi, *Graphics processing units in bioinformatics, computational biology and systems biology,* Briefings in Bioinformatics **18**, 870 (2016).

[32] M. Korpar and M. Šikić, *Sw–gpu-enabled exact alignments on genome scale,* Bioinformatics **29**, 2494 (2013).

[33] J. S. Torres, I. B. Espert, A. T. Dominguez, V. Hernendez, I. Medina, J. Terraga, and J. Dopazo, *Using gpus for the exact alignment of short-read genetic sequences by means of the burrows-wheeler transform,* IEEE/ACM Transactions on Computational Biology Bioinformatics **9**, 1245 (2012).

[34] G. Lightbody, V. Haberland, F. Browne, L. Taggart, H. Zheng, E. Parkes, and J. K. Blayney, *Review of applications of high-throughput sequencing in personalized medicine: barriers and facilitators of future progress in research and clinical application,* Briefings in Bioinformatics , bby051 (2018).

[35] E. B. Fernandez, J. Villarreal, S. Lonardi, and W. A. Najjar, *Fhast: Fpga-based acceleration of bowtie in hardware,* IEEE/ACM Transactions on Computational Biology Bioinformatics **12**, 973 (2015).

[36] TimeLogic, *Accelerated blast performance with tera-blast: a comparison of fpga versus gpu and cpu blast implementations,* 2013.

[37] *Gatk,* `https://software.broadinstitute.org/gatk/` (), accessed August 19, 2018.

[38] H. Li and R. Durbin, *Fast and accurate long-read alignment with burrows–wheeler transform,* Bioinformatics **26**, 589 (2010).

[39] B. Langmead and S. L. Salzberg, *Fast gapped-read alignment with bowtie 2.* Nature Methods **9**, 357 (2012).

[40] Q. Liu, Y. Guo, J. Li, J. Long, B. Zhang, and Y. Shyr, *Steps to ensure accuracy in genotype and snp calling from illumina sequencing data,* BMC Genomics **13**, S8 (2012).

[41] *HaplotypeCaller Call germline SNPs and indels via local re-assembly of haplotypes* (), `https://software.broadinstitute.org/gatk/documentation/tooldocs/current/org_broadinstitute_gatk_tools_walkers_haplotypecaller_HaplotypeCaller.php/`.

[42] Z. Li, Y. Wang, and F. Wang, *A study on fast calling variants from next-generation sequencing data using decision tree,* BMC Bioinformatics **19**, 145 (2018).

[43] A. Proffitt, *Broad, Intel Announce Speed Improvements to GATK Powered by Intel Optimizations*, http://www.bio-itworld.com/2014/3/20/broad-intel-announce-speed-improvements-gatk-powered-by-intel-optimizations.html.

[44] G. VdAuwera, *Speed up HaplotypeCaller on IBM POWER8 systems*, https://software.broadinstitute.org/gatk/blog?id=4833.

[45] *Version highlights for gatk version 3.8,* https://software.broadinstitute.org/gatk/blog?id=10063, accessed August 21, 2018.

[46] X. Li, G. Tan, C. Zhang, X. Li, Z. Zhang, and N. Sun, *Accelerating large-scale genomic analysis with spark,* in *IEEE International Conference on Bioinformatics and Biomedicine* (2017) pp. 747–751.

[47] H. Mushtaq and Z. Al-Ars, *Cluster-based apache spark implementation of the gatk dna analysis pipeline,* in *IEEE International Conference on Bioinformatics and Biomedicine* (2015) pp. 1471–1477.

[48] *Introducing gatk 4.0,* https://www.yourkit.com/java/profiler/ (), accessed August 19, 2018.

[49] *Java profiler,* https://software.broadinstitute.org/gatk/gatk4, accessed August 19, 2018.

[50] *perf(linux),* https://en.wikipedia.org/wiki/Perf_(Linux)#cite_note-2, accessed August 21, 2018.

[51] *Flame graphs,* http://www.brendangregg.com/flamegraphs.html, accessed August 19, 2018.

[52] Y. Yan, M. Grossman, and V. Sarkar, *Jcuda: A programmer-friendly interface for accelerating java programs with cuda,* in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09 (Springer-Verlag, Berlin, Heidelberg, 2009) pp. 887–899

2

GPU-BASED DBG CONSTRUCTION FOR MICRO-ASSEMBLY

SUMMARY

Micro-assembly is used in multiple variant callers to improve the accuracy of detecting variants. In this chapter, we propose a novel GPU-based algorithm of de Bruijn graph construction for micro-assembly in the GATK HaplotypeCaller. The proposed algorithm assumes that there are no repeat k-mers in the dataset and first calculates the occurrences of (k+1)-mers in parallel on the GPU, thereby achieving high speedup. Then the dataset is inspected for repeat k-mers, and only these repeats are re-evaluated on the CPU. Experimental results show that the speedup of our implementation compared with the CPU benchmark implementation for synthetic datasets is up to 3x, while the speedup achieved for real human genome datasets can reach 2.66x.

This chapter is based on the following paper.

S. Ren, N. Ahmed, K.L.M. Bertels, Z. Al-Ars, *An Efficient GPU-based de Bruijn Graph Construction Algorithm for Micro-Assembly*, 18th annual IEEE International Conference on BioInformatics and BioEngineering (BIBE 2018), October 29-31, 2018 [Conference]

An Efficient GPU-based de Bruijn Graph Construction Algorithm for Micro-Assembly

Shanshan Ren Nauman Ahmed Koen Bertels Zaid Al-Ars
Quantum & Computer Engineering Dept.
Delft University of Technology, 2628 CD Delft, The Netherlands
{s.ren, n.ahmed, k.l.m.bertels, z.al-ars}@tudelft.nl

***Abstract*—In order to improve the accuracy of indel detection, micro-assembly is used in multiple variant callers, such as the GATK HaplotypeCaller to reassemble reads in a specific region of the genome. Assembly is a computationally intensive process that causes runtime bottlenecks. In this paper, we propose a GPU-based de Bruijn graph construction algorithm for micro-assembly in the GATK HaplotypeCaller to improve its performance. Various synthetic datasets are used to compare the performance of the GPU-based de Bruijn graph construction implementation with the software-only baseline, which achieves a speedup of up to 3x. An experiment using two human genome datasets is used to evaluate the performance shows a speedup of up to 2.66x.**

***Index Terms*—GPU acceleration; de Bruijn graph construction; micro-assembly; repeat k-mers**

I. INTRODUCTION

Alignment-based variant discovery is a widely used approach to identify variants in genomic data. This approach first aligns the sequencing dataset of a sample genome to a reference genome using alignment tools. It then compares the aligned dataset to the reference genome and extracts the genome positions where the sample genome differs from the reference genome using variant callers. The variants found through this approach include single nucleotide variations (snv's), small insertions/deletions (indels) and structural variations (svs). However, reads with indels are easily misaligned during the alignment step, leading to low accuracy of indel detection.

In order to improve the accuracy of variant detection of indels in particular, various local assembly based variant callers are proposed to correct the misalignment errors of alignment-based variant discovery approach, such as Scalpel [1], Platypus [2] and GATK HaplotypeCaller (HC) [3], [4]. In local assembly based variant callers, reads aligned to a certain region of the reference genome are assembled into a long DNA sequence covering this region. This process is referred to as micro-assembly or local assembly. Assembly based variant callers not only improve the accuracy of indel detection, but also enhance the accuracy of snv detection by making use of linkage disequilibrium between nearby variants.

One of the challenges of genome assembly is repeats in the genome. In most of local assembly based variant callers, a popular method to handle repeats is to avoid cycles in the graph [5], used in Scalpel, GATK HC, and ABRA. If cycles are detected in the graph, the region is reassembled using higher k-mer sizes until there are no cycles in the graph. GATK HC, which is widely used in many large-scale sequencing project, takes more measures to handle repeats. In GATK HC, k-mers are classified into two groups: unique k-mers and repeat k-mers. If a k-mer occurs twice or more than twice in a read, it is a repeat k-mer; otherwise, it is a unique k-mer. During graph construction, unique k-mers are collapsed into single nodes, while repeat k-mers are not collapsed into single nodes. In this way, some cycles in the graph caused by repeats are avoided, which reduces the probability of reassembly with larger k-mer sizes.

Existing assembly algorithms used by many genome assemblers use de Bruijn graphs (DBGs) construction methods. However, since previous efforts to accelerate DBG construction on GPU (such as [6], [7] and [8]) collapse repeat k-mers into single nodes, these methods are not suitable for DBG construction of micro-assembly in GATK HC. In this paper, we propose a novel GPU-based algorithm for DBG construction for micro-assembly in GATK HC.

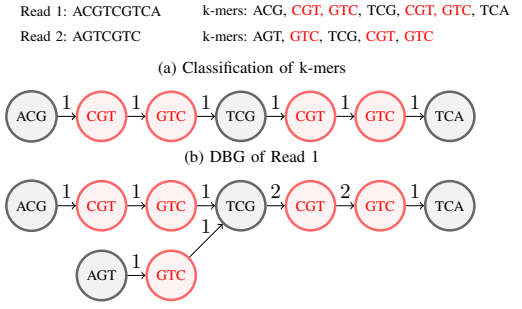
The rest of this paper is organized as follows. Section II describes the algorithm of DBG construction in GATK HC. Section III presents the proposed algorithm of DBG construction. Section IV presents and discusses the experimental results. Finally, Section V concludes this paper.

II. DBG CONSTRUCITON IN GATK HC

The DBG construction for micro-assembly in GATK HC is divided into two main steps.

In step 1, each read aligned to a region is decomposed into multiple k-mers and each k-mer is checked to find repeat k-mers. A region is identified based on significant evidence of variation, which is referred to as active region.

In step 2, each read is then considered as a candidate to create new nodes, create new edges and increase edge weights. For a read, the first unique k-mer is identified ignoring the repeat k-mers before it. From this unique k-mer, the k-mers in this read and the overlap relationships between k-mers are used to construct the de Bruijn graph. This construction is performed as follows: (a) If the node mapped by this unique k-mer has not been created, a new node mapped by this unique k-mer is created. Otherwise, the node mapped by this unique k-mer is identified. Let the node obtained be Node A. (b) The program then checks whether Node A has an edge, which connects Node A and the node mapped by the next k-mer. If this edge is found, the weight of the edge is increased. Otherwise, the program checks whether the next k-mer is a unique k-mer or a repeat k-mer. If it is a unique k-mer, the program finds or creates a node mapped by the next k-mer. If it is a repeat k-mer, a node mapped by the next k-mer is



(a) Classification of k-mers

(b) DBG of Read 1

(c) DBG of Read 1 and Read 2

Fig. 1. Illustration of a simple DBG construction in GATK HC

created. An edge connecting this node and the node mapped by the next k-mer is then created and the weight is set to 1. In this step, the node mapped by the next k-mer is obtained. (c) Let the node obtained in step (b) be Node A and repeat step (b) until there are no k-mers in the read.

Fig. 1 shows a simple example of DBG construction in GATK HC for two reads using a k-mer size of 3. The two reads are first decomposed into multiple k-mers, which are then classified into unique k-mers and repeat k-mers. The repeat k-mers are marked in red. The two reads are then taken in turn to construct the graph. In the graph, black nodes and red nodes represent nodes mapped by unique k-mers and repeat k-mers, respectively. Edges represent overlap relationships between k-mers and the numbers above these edges represent occurrences of the overlap relationships, referred to as edge weights. As shown by Fig. 1, the unique k-mers with the same value are mapped to single nodes in the graph, such as "ACG" and "AGT". However, the repeat k-mers with the same value are mapped to multiple nodes in the graph, such as "CGT" and "GTC". Moreover, the repeat k-mers with the same value from different reads may be collapsed into single node. For example, the second "CGT" in Read 1 and "CGT" in read 2 are mapped to one node. This is because the node mapped by the second "CGT" in Read 1 is created in Fig. 1 (b) and when Read 2 are taken to construct a graph, there is already an edge connecting the node mapped by "TCG" and the node mapped by "CGT" in the graph. As explained in the workflow, the node mapped by a repeat k-mer is created when there is no edge connecting Node A and the node mapped by the repeat k-mer in the graph. Thus, the repeat k-mer "CGT" in Read 2 do not create new nodes and the weight of the edge is increased twice. This example indicates that the creation of the node mapped by a repeat k-mer depends on the early computation results.

III. GPU-BASED DBG CONSTRUCTION

A. Algorithm idea

There are two kinds of nodes in the graph: unique nodes, which are nodes mapped by unique k-mers, and repeat nodes, which are nodes mapped by repeat k-mers. Hence, there are four kinds of edges in the graph: U-U, U-R, R-R and R-U. U-U edge stands for an edge that starts from a unique node and ends with a unique node, etc.

DBG construction can be divided into two parts. One part is to create repeat nodes and U-R, R-R and R-U edges, while the other part is to create unique nodes and U-U edges. The former can be calculated by handling the special subsequences of reads using the method in GATK HC. The special subsequences are defined as two kinds of subsequences: (a) the subsequence having at least three k-mers, among which the first and last k-mers are unique k-mers and the other k-mers are repeat k-mers, and (b) the subsequence having at least two k-mers, among which the first k-mer is a unique k-mer and the other k-mers are repeat k-mers and the last repeat k-mer is the last k-mer of the read where the subsequence comes from. The later is similar to the common way of DBG construction, where the same k-mers are collapsed into single nodes. One of the most popular acceleration methods of the common way of DBG construction is to calculate the occurrences of (k+1)-mers in parallel.

Thus, in this paper we propose the following GPU algorithm for DBG construction. First, we assume there are no repeat k-mers and calculate the occurrences of (k+1)-mers in parallel on the GPU. We then check whether there are repeat k-mers. If there are no repeat k-mers, (k+1)-mers and their occurrences are used to construct the graph. Otherwise, (k+1)-mers having one or two repeat k-mers are deleted and the special subsequences are identified. The remaining (k+1)-mers and their occurrences and the special subsequences are then used to construct the graph.

B. Workflow of GPU algorithm

Fig. 2 shows the workflow of the GPU-based DBG construction algorithm. The input data are the reads aligned to multiple active regions. The output data are the de Bruijn graphs stored using ReadThreadingGraph in GATK HC.

The DBG construction algorithm is implemented through a C program, a CUDA program and a Java program together. There are in total twelve steps. Since GATK HC is a Java-based program, the input data are transferred to the C program through JNI (Java Native Interface) and then transferred to GPU. On the GPU, the 64-bit values of k-mers and (k+1)-mers are generated. The 64-bit values of (k+1)-mers are processed to obtain their occurrences, while the 64-bit values of k-mers are handled to calculate the number of k-mers in each active region after reducing the repeat k-mers in each sequence, which is used to check whether there are repeat k-mers in each active region. The computation results on the GPU are then transferred back to the host. For each active region, if there are repeat k-mers in the active region, the repeat k-mers are found and the special subsequences are identified. Moreover, the 64-bit values of (k+1)-mers having repeat k-mers and their occurrence are deleted. The remaining 64-bit values of (k+1)-mers are transferred to GPU, then transformed into (k+1)-mers and transferred back to the host. The computation results of the C program are transferred to the Java program and used to construct the graphs.

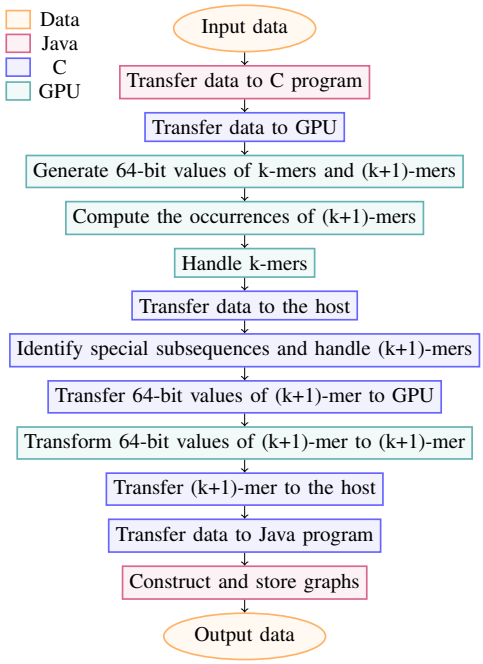


Fig. 2. Workflow of GPU-based DBG construction algorithm

C. Generate 64-bit values of k-mers and (k+1)-mers

This step is implemented on the GPU. Each thread block takes charge of one read to generate 64-bit values of its k-mers and (k+1)-mers, which is shown in Algorithm 1. Every 4 characters of the read are loaded by one thread and stored in the shared memory ($read\_s[]$). Every character of the read is transformed into one byte, which is 0, 1, 2, or 3, and then stored in the shared memory ($Rbyte[]$) by one thread. Finally, each thread performs bitwise operations to generate the 64-bit values of one k-mer and one (k+1)-mer and stores the 64-bit values in the global memory ($kmer\_64[]$ and $kmer\_add\_64[]$). Since reads are from multiple active regions, two arrays ($kmer\_active[]$ and $kmer\_add\_active[]$) are used to store the active region id of each k-mer and (k+1)-mer, respectively. Moreover, one more array ($kmer\_seq\_id[]$) is used to store the sequence id of each k-mer.

D. Compute the occurrences of (k+1)-mers

This step is implemented on the GPU. Functions from the Thrust library [9] are employed to compute the occurrences of (k+1)-mers. Since (k+1)-mers are from multiple active regions, the 64-bit value array of (k+1)-mers and the active region id array of (k+1)-mers are handled together. This step is implemented by three operations, which is shown with an example in Fig. 3.

(1) The elements in the 64-bit value array of (k+1)-mers are sorted in ascending order. In the meanwhile, the elements in the active region id array change their placements in correspondence to the sorting operations in the 64-bit value array of (k+1)-mers. This operation is implemented by the $sort\_by\_key()$ function.

(2) The elements in the active region id array are sorted in ascending order, leading to corresponding changes of element placements in the 64-bit value array of (k+1)-mers. This operation is implemented by the $stable\_sort\_by\_key()$ function.

(3) A constant array is used to store the current occurrence of each (k+1)-mer and the value of each element in this array is 1. The $Reduce\_by\_key()$ function is employed to calculate the number of the consecutive equal elements in the 64-bit value array of (k+1)-mers. The reduced results of the 64-bit value array of (k+1)-mers are stored in a new array and the occurrences of (k+1)-mers are stored in another new array.

The new 64-bit value array of (k+1)-mers and the occur-

Algorithm 1 Generating k-mers and (k+1)-mers

```
1: function GENERATE(read[], length,k_size active_id, seq_id,
   kmer_64[], kmer_active[], kmer_seq_id[], kmer_add_64[],
   kmer_add_active[])
2:     h ← (length + 4 − 1)/4
3:     t ← (h + blockDim.x − 1)/blockDim.x
4:     for i ← 0, t − 1 do                          ▷ Load and store a read
5:         j ← threadIdx.x + i ∗ blockDim.x
6:         if j < h then
7:             a ← read[j]                          ▷ type of a is char4
8:             read_s[j × 4] ← a.x
9:             read_s[j × 4 + 1] ← a.y
10:            read_s[j × 4 + 2] ← a.z
11:            read_s[j × 4 + 3] ← a.w
12:    end for
13:    __syncthreads()
14:    t ← (length + blockDim.x − 1)/blockDim.x
15:    for i ← 0, t − 1 do                          ▷ Transform into bytes
16:        j ← threadIdx.x + i ∗ blockDim.x
17:        if j < length then
18:            b ← read_s[j]
19:            c ← (b == 'A')?0 : ((b == 'C')?1 : ((b == 'G')?2 : 3))
20:            Rbyte[j] = c
21:    end for
22:    __syncthreads()
23:    kmer_number ← length − k_size + 1
24:    t ← (kmer_number + blockDim.x − 1)/blockDim.x
25:    for i ← 0, t − 1 do                          ▷ Generate 64-bit values
26:        j ← threadIdx.x + i ∗ blockDim.x
27:        val ← 0                                  ▷ type of val is uint64_t
28:        if j < kmer_number then
29:            for f ← 0, k_size − 1 do
30:                val ← (val << 2)|(Rbyte[j + f]&3)
31:            end for
32:            kmer_64[j] ← val
33:            kmer_active[j] ← active_id
34:            kmer_seq_id[j] ← seq_id
35:            if j! = kmer_number − 1 then
36:                val ← (val << 2)|(Rbyte[j + k_size]&3
37:                kmer_add_64[j] ← val
38:                kmer_add_active[j] ← active_id
39:    end for
40: end function
```

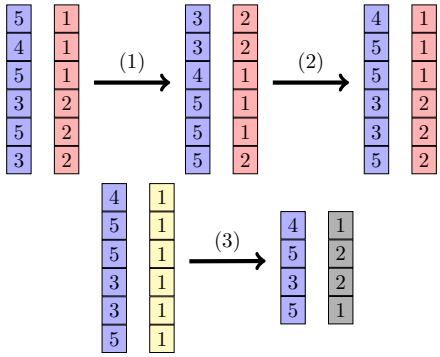


Fig. 3. An example of calculating the occurrences of (k+1)-mers. Purple array stores the 64-bit values of (k+1)-mers. Pink array stores the active region id of each (k+1)-mer. Grey array stores the occurrences of (k+1)-mers. Yellow array is a constant array.

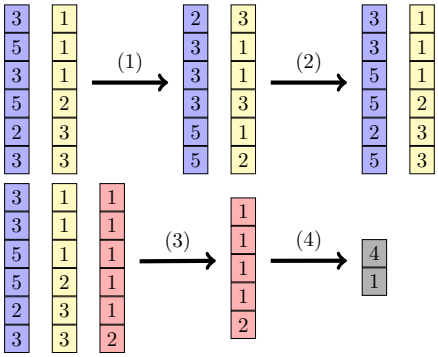


Fig. 4. An example of handling k-mers. Purple array stores the 64-bit values of k-mers. Yellow and pink array stores the sequence id and the active region id of each k-mer, respectively. Grey array stores the number of k-mers in each active region after reducing repeat k-mer in each sequence.

rence array of (k+1)-mers are the computation results of this step and will be used in the following steps.

E. Handle k-mers

This step is implemented on the GPU and also employs functions from the Thrust library. The 64-bit value array of k-mers, the sequence id array of k-mers and the active region id array of k-mers are handled together. This step is implemented by four operations, which is shown with an example in Fig. 4. The first two operations (1) and (2) are similar to the first two operations in Section III-D except the active region id array of (k+1)-mers is replaced by the sequence id array of k-mers. After the first two operations, the 64-bit values of the equal k-mers from the same sequence are consecutive in the 64-bit value array of k-mers.

The third operation (3) is to reduce the consecutive equal elements in the 64-bit value array of k-mers and the sequence id array is used to make sure only the consecutive equal elements from the same sequence are reduced, which is implemented by the $Reduce\_by\_key()$ function. The result of this operation is a new array, which stores the active region ids of the remaining k-mers.

The fourth operation (4) is to calculate the number of k-mers in each active region after reducing repeat k-mers in each sequence, which is implemented by the $Reduce\_by\_key()$ function. The result is stored in a new array, the length of which is equal to the number of active regions.

The array storing the number of k-mers in each active region after reducing repeat k-mers in each sequence is the computation result of this step and will be used in the following steps.

F. Identify special subsequences and handling (k+1)-mers

This step is implemented by a C program, which is shown in Algorithm 2. For each active region, if the number of k-mers is equal to the number of k-mers calculated in Section III-E, there are no repeat k-mers in the region. Otherwise, there are repeat k-mers in the region. Each read is then checked to find repeat k-mers. After all the repeat k-mers in the region are found, each read is checked again to identify the special subsequences.

Since operation (3) in Section III-D does not make sure that only the consecutive equal 64-bit values of (k+1)-mers from the same active region are reduced, extra operations are taken to remedy this, which are from line 11 to line 26 in Algorithm 2. For each active region, if the sum of the weights of existing (k+1)-mers of the region is smaller than the number of (k+1)-mers in the region, a (k+1)-mer has a chance to be added to the region and the weight of the (k+1)-mer added to the region is calculated from line 14 to line 19. If the 64-bit value of the (k+1)-mer has repeat k-mers, the (k+1)-mer will not be added to the region and neither will the calculated weight of the (k+1)-mer.

G. Transform 64-bit values of (k+1)-mer into (k+1)-mer

This step is implemented on the GPU, where each thread takes charge of one (k+1)-mer. Each thread first loads the 64-bit value of a (k+1)-mer into its registers and then transform every 2 bits to a character, which is then stored in the shared memory. After transformation, characters calculated by each thread in one thread block are stored in the consecutive addresses. Finally, each four characters composing a char4 value are stored in the global memory by one thread. In this way, the global memory accesses of the threads in a warp (32 consecutive threads) to store characters are coalesced. The computation result of this step is an array storing the characters of all (k+1)-mers.

H. Construct and store graphs

This step is implemented by a Java program. For each active region, (k+1)-mers are used to create unique nodes and U-U edges and occurrences of (k+1)-mers are the weights of U-U edges. If the active region has repeat k-mers, the special subsequences are handled using the method in GATK HC to create repeat nodes and U-R, R-R and R-U edges and increase the weights of these edges. All the nodes, edges and weights of these edges are stored using ReadThreadingGraph.

Algorithm 2 Identifying special subsequences and handling (k+1)-mers

```
1: function IDENTIFY(region_number, k_region[], k_add_region[], k_region_GPU[], kmer_add_64[], k_add_number[], new_64, new_number[])
2:     t ← 0, p ← 0
3:     for i ← 1, region_number do
4:         if k_region[i] == k_region_GPU[i] then
5:             for reads in the region do
6:                 find_repeat_kmer()
7:             end for
8:             for reads in the region do
9:                 find_subsequence()
10:            end for
11:        cur ← 0
12:        while cur < k_add_region[i] do
13:            cur+ = k_add_number[t]
14:            if cur > k_add_region[i] then
15:                sub ← cur − k_add_region[i]
16:                n ← k_add_number[t] − sub
17:                k_add_number[t] ← sub
18:            else
19:                n ← k_add_number[t]
20:            if Not_have_repeat(kmer_add_64[t]) then
21:                new_64[p] ← k_add_64[t]
22:                new_number[p] ← n
23:                p + +
24:            if cur <= k_add_region[i] then
25:                t ← t + 1
26:        end while
27:    end for
28: end function
```

IV. RESULTS AND DISCUSSION

A. Experimental setup

We compare the GPU-based DBG construction implementation and DBG construction implementation in GATK HC (CPU benchmark) with both synthetic and real datasets. The CPU implementation is achieved by modifying GATK HC 3.7 to read input datasets, construct DBGs and output graphs.

The input datasets are reads of multiple active regions. In order to get the synthetic input datasets, we first use Wgsim [10] to generate reads from a reference sequence, which is chromosome 19 of the human genome (UCSC hg19), then follow the GATK best practices pipeline to get the input datasets for GATK HC and use GATK HC to produce reads of multiple active regions. As to the real datasets, the processing steps are the same as for the synthetic datasets except that we use reads produced by an NGS platform instead of the reads simulated by Wgsim.

The output datasets are DBGs of multiple active regions. A simple program is designed to sort the nodes and edges in each graph in order to compare the output and assure the correctness of the results.

A server-class machine is used to perform all the experiments. This machine has two Intel Xeon processors, each of which has 14 two-way hyper-threaded cores running at 2.4 GHz, 192 GB of RAM, and an NVIDIA Tesla K40 card, which consists of 2880 cores running at 745 MHz.

B. Impact of coverage on performance

We generated 8 synthetic datasets using Wgsim with different levels of read coverage, ranging from 10x to 80x. For all these synthetic reads, we used the Wgsim parameters of 100bp for the read length, 0.01 for the mutation rate, and 0.15 for the indel fraction. The k-mer size is 10.

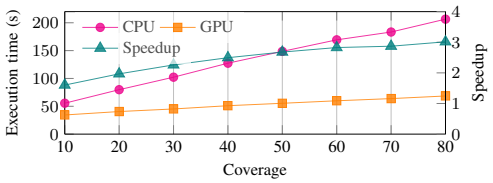


Fig. 5. Execution time and speedup vs coverage of the GPU and CPU implementations with the synthetic datasets

Fig. 5 shows that the execution time and speedup of the GPU and CPU implementations with respect to coverage of the synthetic datasets. The figure indicates that the execution time of both implementations increases with increasing coverage. However, the execution time of the CPU implementation increases faster than the GPU-based implementation. Thus, the speedup of the GPU-based DBG construction implementation increases with increasing coverage. When the coverage of the synthetic dataset is 80x, the speedup is the highest at 3x.

The execution time of the GPU-based implementation is divided into four parts: computation time on the GPU (including data transfer to/from GPU), computation time of the C program, computation time of the Java program and data transfer time using JNI. Fig. 6 shows the percentage of the four parts of the execution time of the GPU-based implementation with the synthetic datasets for different coverage levels. The computation time on the GPU occupies a large part of the total execution time, which increases when the coverage increases. This is because when the coverage increases, the number of reads aligned to each active region increase as well, which increases the number of k-mers and (k+1)-mers and in turn increases the computation time on the GPU.

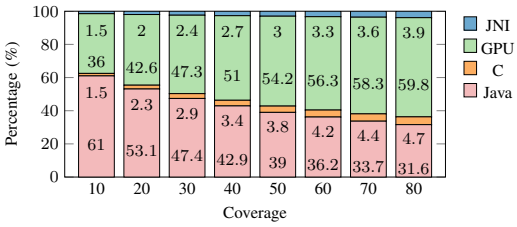


Fig. 6. Percentage of the four parts of the execution time of the GPU-based implementation with the synthetic datasets for different coverage levels

For the GPU-based implementation, the weights of edges are calculated in parallel on the GPU by calculating the occurrences of (k+1)-mers; while for the CPU implementation, the weights of edges are serially increased. Thus, the different methods of handling weights of edges make the execution time growth rates of the two implementations different when the coverage increases.

C. Impact of mutation rate on performance

We used a total of 16 synthetic datasets divided into 4 groups according to their coverage (20x, 40x, 60x and 80x). Each group consists of 4 synthetic datasets with a different mutation rate: 0.05%, 0.1%, 0.5% and 1%. The other parameters of Wgsim for each dataset are kept the same: read length is 100bp, indel fraction is 0.15. The k-mer size is 10.

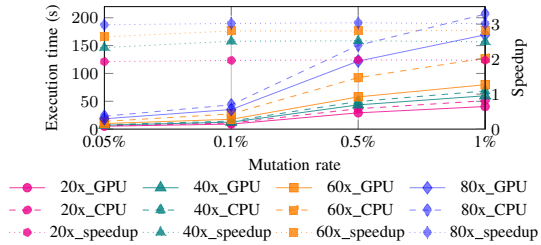


Fig. 7. Execution time and speedup of the two implementations with the synthetic datasets of different mutation rate

Fig. 7 shows the execution time and speedup of the two implementations with all 16 synthetic datasets. The execution time of the two implementations of different coverage increases when the mutation rate increases. This is because when the mutation rate increases, the number of regions which are chosen based on the significant evidence of variation increases and in turn the input data handled by the two implementations increases. In addition, Fig. 7 shows that the speedup of the GPU-based implementation does not change much when the mutation rate increases. To explain this behavior, we take the synthetic datasets with coverage 80x as an example. For this coverage level, Fig. 8 shows that the percentage of the computation time on the GPU changes a little when the mutation rate increases, resulting in little change of the speedup brought by GPU acceleration. In addition, the percentage of the computation time of the C program slightly increases when the mutation rate increases. This is because the number of different k-mers and different (k+1)-mers increase as well when the mutation rate increases.

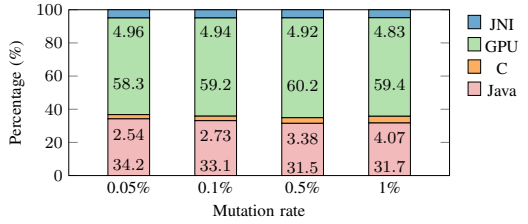


Fig. 8. Percentage of the four parts of the execution time of the GPU-based implementation with different mutation rates for coverage 80x

D. Performance with real datasets

We use 2 real datasets to compare the performance of the GPU and CPU implementations for a k-mer size of 10. The first dataset is chromosome 20 of NA12878 downloaded from the GATK resource bundle, coverage of which is ∼64x. The second dataset is chromosome 17 of G15512.HCC1954.1, coverage of which is ∼58x.

Table I shows the speedup of the two implementations with respect to the real datasets. The speedup of the first dataset is 2.66x and the speedup of the second dataset is 2.47x.

TABLE I
PERFORMANCE COMPARISON FOR REAL DATASETS

Dataset	CPU time (s)	GPU time (s)	Speedup
1	72.9	27.4	2.66x
2	53.8	21.7	2.47x

V. CONCLUSIONS

Micro-assembly is a widely used technique to increase the accuracy of variant callers, such as the popular GATK HC. This paper proposes a GPU-based DBG construction algorithm for micro-assembly in GATK HC. The proposed algorithm assumes that there are no repeat k-mers in the dataset and calculate the occurrences of (k+1)-mers in parallel on the GPU, thereby achieving high speedup. Then the dataset is inspected for repeat k-mers, and only these repeats are re-evaluated on the CPU. Experimental results show that the speedup of our implementation compared with the CPU benchmark implementation for synthetic datasets is up to 3x, while the speedup achieved for real human genome datasets can reach 2.66x.

REFERENCES

[1] H. Fang, E. A. Bergmann, K. Arora, V. Vacic, M. C. Zody, I. Iossifov, J. A. O'Rawe, Y. Wu, L. T. Jimenez Barron, J. Rosenbaum, M. Ronemus, Y. H. Lee, Z. Wang, E. Dikoglu, V. Jobanputra, G. J. Lyon, M. Wigler, M. C. Schatz, and G. Narzisi. Indel variant analysis of short-read sequencing data with Scalpel. *Nat Protoc*, 11(12):2529–2548, Dec 2016.

[2] A. Rimmer, H. Phan, I. Mathieson, Z. Iqbal, S.R.F. Twigg, WGS500 Consortium, A.O.M. Wilkie, G. McVean, and G. Lunter. Integrating mapping-, assembly- and haplotype-based approaches for calling variants in clinical sequencing applications. *Nat Genet*, 46(8):912–918, 08 2014.

[3] G. A. Van der Auwera, M. O. Carneiro, C. Hartl, R. Poplin, G. Del Angel, A. Levy-Moonshine, et al. From FastQ data to high confidence variant calls: the Genome Analysis Toolkit best practices pipeline. *Curr Protoc Bioinformatics*, 43:1–33, 2013.

[4] S. Ren, K.L.M. Bertels, and Z. Al-Ars. Efficient acceleration of the pair-hmms forward algorithm for gatk haplotypecaller on gpus. *Evolutionary Bioinformatics*, 14, March 2018.

[5] G. Narzisi and M. C. Schatz. The challenge of small-scale repeats for indel discovery. *Front Bioeng Biotechnol*, 3:8, 2015.

[6] Mian Lu, Qiong Luo, Bingqiang Wang, Junkai Wu, and Jiuxin Zhao. *GPU-Accelerated Bidirected De Bruijn Graph Construction for Genome Assembly*, pages 51–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[7] S. Qiu and Q. Luo. Parallelizing big de bruijn graph construction on heterogeneous processors. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1431–1441, June 2017.

[8] D. Li, C. M. Liu, R. Luo, K. Sadakane, and T. W. Lam. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics*, 31(10):1674–1676, May 2015.

[9] Thrust: A productivity-oriented library for cuda. In *GPU Computing Gems, Jade Edition*, pages 359 – 371. Morgan Kaufmann, Boston, 2012.

[10] Wgsim. https://github.com/lh3/wgsim. Accessed January 28, 2018.

3

FPGA ACCERLERATION OF THE PAIR-HMMS FORWARD ALGORITHM

SUMMARY

In this chapter, we use FPGAs to accelerate the pair-HMMs forward algorithm. We first propose a novel systolic array design to accelerate the pair-HMMs forward algorithm on FPGAs. We present an implementation of the design on the Convey supercomputing platform. Experimental results show that the FPGA implementation of the pair-HMMs forward algorithm is up to 67x faster, compared to software-only execution.

We then model the performance characteristics of the systolic array design. Based on this analysis, we propose a novel architecture to optimize the utilization of the systolic array. The implementation achieves up to 90% of the theoretical throughput for a real dataset and is 2.5x faster than the state-of-the-art implementation on a similar contemporary platform.

This chapter is based on the following papers.

1. **S. Ren**, V.M. Sima, Z. Al-Ars, *FPGA Acceleration of the Pair-HMMs Forward Algorithm for DNA Sequence Analysis*, International Workshop on High Performance Computing on Bioinformatics (HPCB 2015), November 9-12, 2015 [Conference]

2. J.W. Peltenburg, **S. Ren**, Z. Al-Ars, *Maximizing Systolic Array Efficiency to Accelerate the PairHMM Forward Algorithm*, IEEE International Conference on Bioinformatics and Biomedicine (BIBM 2016), December 15-18, 2016 [Conference]

2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)

FPGA Acceleration of the Pair-HMMs Forward Algorithm for DNA Sequence Analysis

Shanshan Ren[1] Vlad-Mihai Sima[1,2] Zaid Al-Ars[1,2]
[1]Computer Engineering Lab
Delft University of Technology
2628 CD Delft, The Netherlands
Email: {s.ren, z.al-ars}@tudelft.nl
[2]Bluebee
Molengraaffsingel 12–14
2629 JD Delft, The Netherlands
vlad.sima@bluebee.com

***Abstract*—Many DNA sequence analysis tools have been developed to turn the massive raw DNA sequencing data generated by NGS (Next Generation Sequencing) platforms into biologically meaningful information. The pair-HMMs forward algorithm is widely used to calculate the overall alignment probability needed by a number of DNA analysis tools. In this paper, we propose a novel systolic array design to accelerate the pair-HMMs forward algorithm on FPGAs. A number of architectural features have been implemented to improve the performance of the design, such as early exit points to increase the utilization of the array for small sequence sizes, as well as on-chip buffering to enable the processing of long sequences effectively. We present an implementation of the design on the Convey supercomputing platform. Experimental results show that the FPGA implementation of the pair-HMMs forward algorithm is up to 67x faster, compared to software-only execution.**

Keywords—NGS, FPGA, pair-HMMs, hardware acceleration.

I. INTRODUCTION

NGS technology [1] provides a high-throughput and cost-effective sequencing method of DNA, creating vast opportunities for profound understanding of human disease. The analysis and interpretation of large-scale sequencing data produced by NGS is a major challenge, requiring complex statistical models and sophisticated bioinformatics tools to turning raw sequencing data into biologically meaningful information. There are a number of such tools currently available and being used widely, such as BWA, SAMtools, SOAP, VarScan and GATK.

Pair-HMMs (pair hidden Markov models) [2] are very popular for finding pairwise alignment of DNA sequences. There are two ways to use pair-HMMs in biological sequence alignment: 1) identifying optimal sequence alignment, and 2) providing the overall alignment probability. Using pair-HMMs to find the optimal sequence alignment is very popular in many different biological sequence analysis tools, such as ProbCons [3], PicXAA [4] and GLProbs [5]. Pair-HMMs identify the alignment with the largest probability as the optimal sequence alignment. The algorithm to find the optimal sequence alignment of pair-HMMs is called the Viterbi algorithm.

If the similarity of the two sequences is not strong, it is hard to find the correct alignment that gives biological meaning. Instead, pair-HMMs can then be used to calculate the probability that two sequences are related, which is referred to as the overall alignment probability [2]. The overall alignment probability is widely used in many biological sequence analysis tools. For example, [6] exploits the overall alignment probability to find the evolutionary distance between two sequences. The GATK HaplotypeCaller [7] calculates the overall alignment probability of the sequences and the candidate mutations to identify their occurrence reliability.

Pair-HMMs forward algorithm computes the overall alignment probability by summing over all possible alignments of a given pair of DNA sequences. The forward algorithm is a dynamic programming algorithm with a computational complexity of O(nm) (n and m are the length of two sequences), which is very large for long sequences. This drawback would influence the performance and limit the feasibility of pair-HMMs. In this paper, we investigate and propose an FPGA-based acceleration of the pair-HMMs forward algorithm with the purpose of improving its performance.

In this paper, we present the following contributions: (1) propose a novel systolic array design of the pair-HMMs forward algorithm; (2) analyze a number of optimization techniques to improve performance; and (3) present an implementation of the design on the Convey supercomputing platform. The results shows that the FPGA-based implementation is around 67x faster, compared to the software-only execution.

The rest of this paper is organized as follows. Section 2 presents a brief overview of related work. Section 3 discusses the details of the pair-HMMs forward algorithm. Section 4 discusses the design specification and optimizations of the accelerated version of the algorithm. The implementation results are discussed in Section 5. We conclude the paper and discuss future work in Section 6.

II. RELATED WORK

FPGAs are widely used to accelerate biological algorithms to achieve large speedup as many bioinformatics workloads lend themselves well to parallel execution. Examples range from commercially available implementations such as the Tera-BLAST [8], to more research oriented algorithm acceleration, such as the acceleration of SAMtools [9]. Tera-BLAST is an FGPA implementation of the BLAST aligner that achieves a 27x speedup over a 32-core CPU implementation. [9] accelerates SAMtools on FPGAs, which achieve a speedup of 2.93x over the original version of SAMtools.

As the Viterbi algorithm has been used in many topics, such as pairwise alignments, multiple sequence alignment and gene prediction, there is much research focusing on the

978-1-4673-6799-8/15/$31.00 ©2015 IEEE

acceleration of the Viterbi algorithm [10][11][12]. The acceleration of the Viterbi algorithm commonly utilizes a log-transformation of the original equations, which transform floating-point multiplication operations into floating-point addition operations. The forward algorithm, on the other hand, requires an addition operation in the probability domain, which prevents using the log-transformation to the forward algorithm. Therefore the acceleration approach used for the Viterbi algorithm cannot be applied to accelerate the forward algorithm.

III. PAIR-HMMS FORWARD ALGORITHM

Pair-HMMs have evolved from the basic HMMs. In a pair-HMM, the HMM model generates an aligned pair of sequences instead of only a single sequence. Figure 1 shows a typical pair-HMM, which is widely used in biological sequence analysis. We assume the two sequences generated by the pair-HMM are sequence X and Y. Figure 1(a) shows the state transition of the pair-HMM, which has three hidden states I_X, I_Y and M. I_X and I_Y are used to emit a single unaligned symbol only in sequence X and Y, respectively. M is used to emit an aligned pair of two symbols, where one symbol is added to sequence X and the other symbol is added to sequence Y. By traversing between the states I_X, I_Y and M, the pair-HMM generates two sequences. Figure 1(b) shows an example of an aligned pair of sequences X=ACGTC and Y=ACGAA, which are generated according to the hidden state sequence $MMMI_XI_XI_YI_Y$. The first three symbols in sequence X and sequence Y are emitted by state M. The last two symbols in sequence X are emitted by state I_X and the last two symbols in sequence Y are emitted by state I_Y. The probability of the generated alignment is the product of the state transition probabilities.

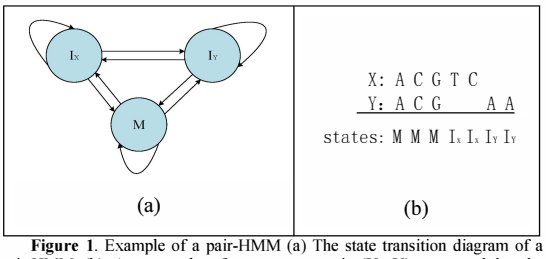


Figure 1. Example of a pair-HMM (a) The state transition diagram of a pair-HMM (b) An example of a sequence pair (X, Y) generated by the pair-HMM

From the simple example shown by Figure 1, we could see that a hidden state sequence generates an aligned pair of sequences with a specific alignment probability. If we want to find the overall alignment probability, we need to add the alignment probability of all hidden state sequences. Obviously, it is not practical to enumerate all hidden state sequences. Thus, the forward algorithm is proposed to solve this problem.

The pair-HMMs forward algorithm is implemented as a dynamic programming algorithm, as shown by Equations (1) to (3), where n and m are the length of sequence X and Y, respectively. α, β, γ, δ, ε, ζ and η are the transmission probabilities, while λ, θ, and υ are the emission probabilities. The transmission probabilities and the emission probabilities are supplied by the two sequences. In these equations, $M_{i,j}$ stands for the overall alignment probability of two sub-sequence X[1]…X[i] and Y[1]…Y[j]. $I_{i,j}$ stands for the overall alignment probability of X[1]…X[i] and Y[1]…Y[j] with X[i] aligned to gap. $D_{i,j}$ stands for the overall alignment probability of X[1]…X[i] and Y[1]…Y[j] with Y[j] aligned to gap.

Initialization:

$$\begin{cases} M_{0,0} = 1, I_{0,0} = D_{0,0} = 0 \\ M_{i,0} = I_{i,0} = 0, 0 < i \le n \\ M_{0,j} = D_{0,j} = 0, 0 < j \le m \end{cases} \quad (1)$$

Recurrence:

$$\begin{cases} M_{i,j} = \lambda \times (\alpha M_{i-1,j-1} + \beta I_{i-1,j-1} + \gamma D_{i-1,j-1}) \\ I_{i,j} = \theta \times (\delta M_{i-1,j} + \varepsilon I_{i-1,j}) \\ D_{i,j} = \upsilon \times (\zeta M_{i,j-1} + \eta D_{i,j-1}) \end{cases} \quad (2)$$
$$(1 \le i \le n, 1 \le j \le m)$$

Termination:

$$Result = M_{n,m} + I_{n,m} + D_{n,m} \quad (3)$$

As shown by these equations, three matrices are filled and the process to fill these matrices contains much inherent parallelism. Each matrix element of $M_{i,j}$, $I_{i,j}$ and $D_{i,j}$ only depends on the up-left, up and left neighbor elements of each matrix. This implies all elements on the same anti-diagonal in each matrix can be computed in parallel.

Algorithm 1 shows a pseudo code of the pair-HMMs forward algorithm. As shown by Algorithm 1, the computation complexity of the pair-HMMs forward algorithm is $O(nm)$. When the lengths of the two sequences increase, the execution time of the dynamic programming algorithm increases quadratically, which causes the high computational complexity of pair-HMMs.

Numerical underflow is a significant problem when implementing the pair-HMMs forward algorithm, as the probability of some alignments would be smaller than the smallest representable floating-point value. There are two methods to solve this problem: implementing a "log-sum" operation and rescaling [13][14]. The first method transforms the floating-point multiplications into floating-point addition in the log probability domain, but it needs to build a large look-up table containing the values used in the computation process. The second method is rescaling, which rectifies the intermediate results, however leading to many extra computations.

Some DNA sequence analysis tools, such as GATK HapltoypeCaller, implement the pair-HMMs forward algorithm directly in the probability domain without using either of these two complex methods. The reason is that, on the one hand, the numerical underflow problem does not frequently occur in DNA sequence analysis, as the length of DNA sequences is 70~250bps and numbers in the double floating-point format have an approximate range of 10^{-308} to 10^{308}. On the other hand, if a number underflows, the impact on the final result is negligible. Thus, we would implement the pair-HMMs forward algorithm in the probability domain as well.

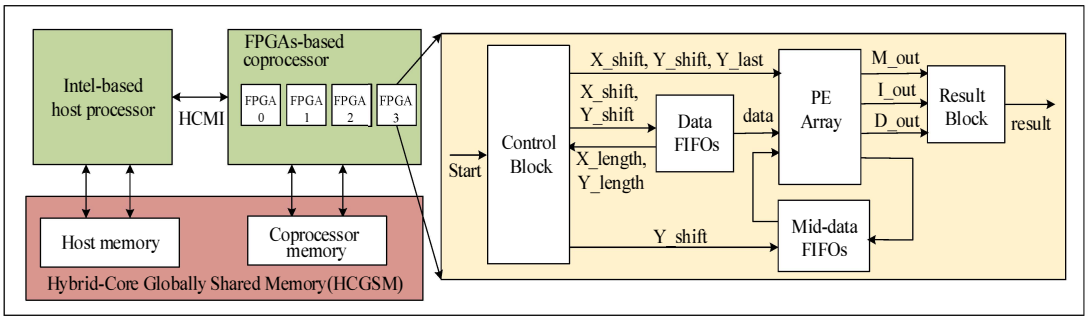


Figure 2. Block diagram of the overall architecture design

Algorithm 1. The pseudo code of the pair-HMMs forward algorithm

```
M[0][0]=1;
D[0][0]=I[0][0]=0;
for(i=1;i<=n;i++)
begin
     M[i][0]=I[i][0]=0;
end
for(i=1;i<=m;i++)
begin
     M[0][i]=D[0][i]=0;
end
for(i=1;i<=n;i++)
begin
     for(j=1;j<=m;j++)
     begin
     M[i][j]=λ×
(α×M[i-1][j-1]+β×I[i-1][j-1]+γ×D[i-1][j-1])
     I[i][j]=θ×(δ×M[i-1][j]+ε×I[i-1][j])
     D[i][j]=υ×(ζ×M[i][j-1]+η×D[i][j-1])
     end
end

result=M[n][m]+I[n][m]+D[n][m];
```

IV. DESIGN SPECIFICATION

In this section, we first present an overview of the system architecture of the FPGA implementation and then introduce the details of the hardware design.

A. Architecture overview

Convey proposed innovative hybrid-core platforms (HC-1, HC-1ex, HC-2 and HC-2ex) [15], which combine classic Intel x86 microprocessors with a coprocessor comprised of FPGAs. The platforms are useful for acceleration of computationally intensive applications, by offloading complex kernels onto the FPGA at runtime, while running the other part of the application on the host processor. We exploited HC-2ex to implement the acceleration of the pair-HMMs forward algorithm. Figure 2 shows the overall architecture of the design.

As shown in Figure 2, the host processor and coprocessor have their own physical memory. The two physical memories are mapped in the same virtual memory address space, which makes it very easy for the application developers. Efficient data transfer mechanisms are provided for transferring between the two memories. The hybrid-core memory interconnect (HCMI) is responsible for signals between the host processor and the coprocessor, such as the start signal sent by the host processor and the finish signal sent by the coprocessor.

The pair-HMMs forward algorithm is implemented on the coprocessor. The function of each block is described below:

Control Block: This block is used to control the progress of the pair-HMMs algorithm. It gets the start signal from the host processor, which indicates that the FPGAs can start their computational cycle; it gets the length of two sequences from Data FIFOs.

Data FIFOs: These FIFOs are used to store data from the memory and output data to the PE array and Control Block according to the control signals from the Control Block.

PE Array: This is used to compute the elements in the three matrixes M(i,j), I(i,j) and D(i,j). It is described in Section IV.B.2.

Mid-data FIFOs: This block is used to store the intermediate data generated by the PE array.

Result Block: This block consists of three floating-point adders and one register to calculate the result.

B. Systolic array mapping

1) Systolic arrays

The potential parallelism of the pair-HMMs forward algorithm allows us to exploit systolic arrays to accelerate the performance on FPGAs. Systolic arrays were proposed by H. T. Kung and C. E. Leiserson [16] and have since been widely used in computing matrix multiplication, LU-decomposition and dynamic programming algorithms. A systolic array is an array of identical Processing Elements (PEs), each of which gets its inputs from the previous PE and passes its outputs to the next PE. All these PEs operate in parallel. This characteristic can be used to compute the elements on the same anti-diagonal in each matrix, which can be computed in parallel.

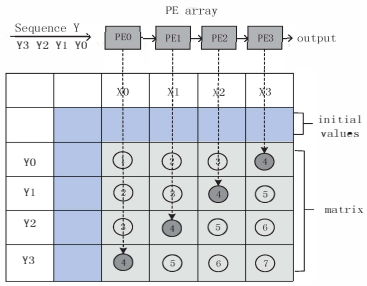


Figure 3. Mapping the pair-HMMs forward algorithm to a systolic array

We map the pair-HMMs forward algorithm to a systolic array, as shown by Figure 3. Sequence X is shifted through the array and each PE stores one of the elements of X. Then the bases of Sequence Y shift through the PEs. In each cycle, a PE calculates one element of each matrix and passes the resulting values to the next element. As the PEs run in parallel, the PE array calculates the elements in the same anti-diagonal in each cycle. For example, in the fourth clock cycle, the PE array is mapped to calculate the elements marked with 4.

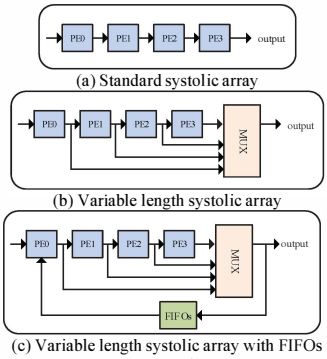


Figure 4. Systolic arrays

Figure 4(a) shows one implementation of the PE array in case the number of PEs is equal to the read length. In this case, the total computation time is shown by Equation (4).

$$T = 2 \times \text{Length}(X) + \text{Length}(Y) - 1 \quad (4)$$

If the FPGA is able to host more PEs than the needed length of Sequence X, the total computation time would be

$$T = 2 \times \text{PEnumber} + \text{Length}(Y) - 1 \quad (5)$$

In this case, the computation time is not limited by the actual length of the sequence, but by the time needed for the systolic array to complete processing all its PEs. This is caused by the fact that Sequence X and Y need to travel through the entire PE array in order to produce the final result to the output, thereby causing unnecessary latency, in addition to a reduced utilization efficiency of the systolic array. To overcome this limitation, we insert exit points after each PE in the PE array, as shown in Figure 4(b) [17]. Using this method, the bases of Sequence X are copied into the needed PEs and Sequence Y only needs to travel through the PEs where the bases are stored. Thus, the total computation time is reduced according to Equation (4).

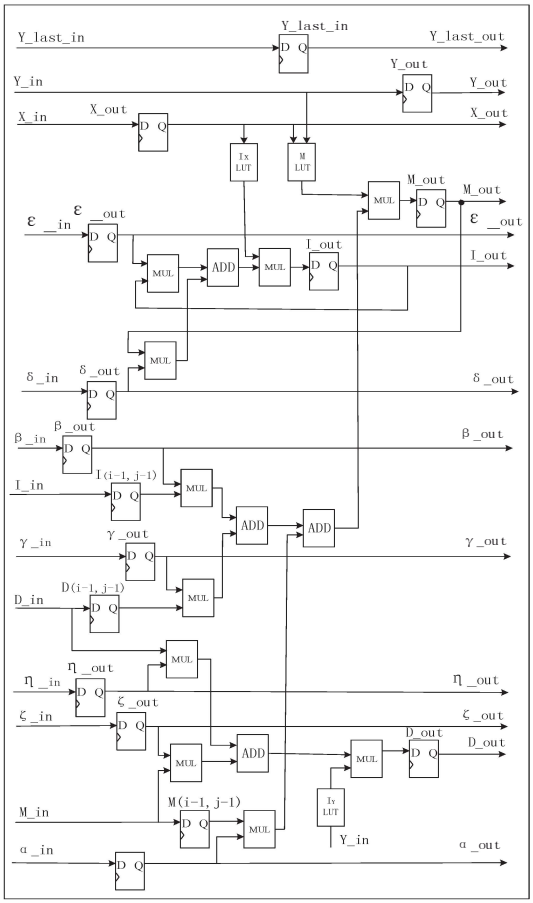


Figure 5. PE schematic for the pair-HMMs forward algorithm

If the number of PEs is smaller than the length of Sequence X, we divide the read into sub-sequences. In each iteration, one of the sub-sequences is shifted into the systolic array and the results are stored in the FIFOs. In subsequent iterations, the results in the Mid-data FIFOs and the next sub-sequence are shifted into the systolic array. This PE array is shown in Figure 4(c). The number of iterations is specified according to the length of Sequence X and the number of PEs in the systolic array. The actual systolic array implemented on the Convey FPGA is the one shown in Figure 4(c).

2) PE schematic

Figure 5 shows the PE schematic for the pair-HMMs forward algorithm. X_in and Y_in represent the base of the Sequence X and Y respectively. α_in, β_in, γ_in, δ_in, ϵ_in, ζ_in and η_in, represent α, β, γ, δ, ϵ, ζ and η respectively.

M_in, D_in and I_in respectively represent M(i,j), D(i,j) and I(i,j), which are calculated by the previous PE. Y_last_in indicates the last base of the Sequence Y is shifting into the PE

As shown by Figure 5, there are 10 floating-point multipliers and 4 floating-point adders. In order to avoid the high area costs of floating point arithmetic units, we use the DSP DSP48E1 components on the FPGAs to implement these arithmetic computations.

C. Transfer overhead

When the host processor has a pair of sequences to compute its overall alignment probability, the first step is that the host processor sends a start signal to the coprocessor and the coprocessor start to execute. Then the Control Block sends request to the memory to transfer data and store data in FIFOs on FPGAs. When the data transfer finishes, the Control Block sends signal to the PE array and FIFOs to start to compute. When the computation finishes, the Control Block writes results back to memory.

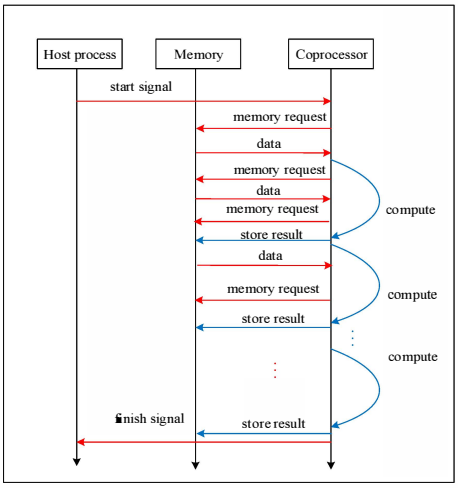


Figure 6. The flow chart of the execution for the pair-HMMs forward algorithm.

If we have several pairs of sequences to deal with, the steps described above are not an efficient solution as the data transfer is very time consuming and it makes the execution time very large. We propose a flow chart of the execution of the forward algorithm, as shown by Figure 6. For the first pair of sequences, the coprocessor waits for the data transfer from memory. As the coprocessor is computing the result of the first pair of sequences, it continues to load data from memory to be stored in the Data FIFOs. The size of Data FIFOs decides the size of the preload data. Based on profile of the FPGA implementation we determined that the most efficient is that the Data FIFOs holds 4 pairs. When the coprocessor finishes the computation of the first pair, it does not need to wait for data transfer of the following pairs of sequences and starts to compute immediately. In this way, data transfer and computation work in parallel, reducing the total execution time by hiding the data transfer overhead.

V. EXPERIMENTAL RESULTS

A. Experimental setup

All tests were run on the Convey HC-2EX platform. The platform has two Intel Xeon E5-263 processors (four cores each, HyperThreading disabled) running at 3.3 GHz with 64 GB of DDR3 memory and four Xilinx Virtex-6 LX760 FPGA co-processors each with 64 GB of SG-DIMM of memory.

Each FPGA is programmed with a pair-HMMs forward algorithm module. All modules on each FPGA run in parallel. Each FPGA contains the same number of PEs, which is limited by the available resources on the FPGA chip. The PE array working frequency is 75MHz.

The pair-HMMs forward algorithm module is implemented using single-precision floating-point variables as it is the case in software packages use pair-HMMs. In general, this is adequate for most sequence analysis tools. If numerical underflow occurs using single precision floats, this will be signaled and recalculated by the pair-HMMs forward algorithm in double precision in the software.

We use the datasets downloaded from [18] to evaluate the performance. The dataset represents pair-HMMs inputs generated by HaplotypeCaller from GATK version 2.7, while calling the 1000 Genomes Project sample NA12878 which is publicly available in the 1000G FTP.

B. Speedup

First, we run the software-only implementation programmed in C++ with default parameters (according to the pseudo code shown by Algorithm 1) to measure the baseline performance. We also run 5 different FPGA implementations of the algorithm with various number of exit points: 1, 2, 3, 4 and all possible exit points. The actual locations of the exit points in each of these designs are listed in Table 1. Note that the increase in the number of exit points uses a larger portion of the FPGA, which results in a corresponding drop in the number of synthesizable PEs. With only one exit point, 96 PEs can be synthesized, while using all exit points reduces this number to 91 PEs.

Table 1. Location of exit points in the different FPGA designs

# exit points	Location of exit points	# synthesizable PEs
1	96	96 PEs
2	46, 93	93 PEs
3	31, 62, 93	93 PEs
4	25, 50, 75, 93	93 PEs
91	All	91 PEs

Figure 7 shows the maximum speedup attainable from the 5 different implementations with different number of exit points as compared with the software-only implementation. The implementation with 91 PEs and 91 exit points achieves the highest amount of speedup of 67x. The implementation with only one exit point results in the least amount of performance achieving a speedup of 62x. It is interesting to note that, the speedup correlates better with the number of exit points rather than the number of PEs, thereby increasing the performance

with the increasing number of exit points. This can be explained by the fact that PEs are relatively large in size and therefore limited in number, which means that reducing the number of PEs by a limited amount can free enough resources to connect all PEs to an exit point.

This also indicates that the performance depends on the length of the sequences in the input dataset, which causes the performance to change with a changing data set.

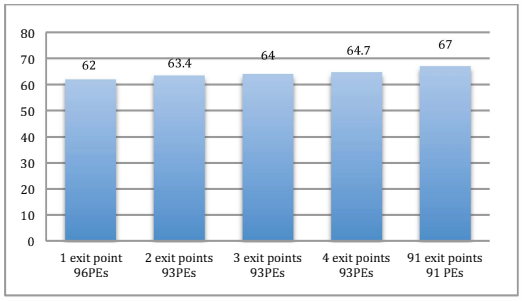


Figure 7. Speedup of the different number of exit points implementation

Table 2 lists the hardware resource utilization of the various designs used in the analysis. The table shows that the implementations are mainly limited by the number of used DSPs (used for single-precision floating-point calculations) and the occupied slices. The table also shows that registers are under utilized in all designs.

Table 2. Hardware resource utilization of different designs

# exit points	# PEs	%LUTs	%Registers	%DSPs	%Occupied slices
1	96	92%	24%	100%	98%
2 to 4	93	90%	23%	97%	98%
91	91	89%	23%	95%	98%

VI. Conclusions and Future Work

In this paper, we propose a novel systolic array design to accelerate the pair-HMMs forward algorithm on FPGAs. A number of architectural features have been implemented to improve the performance of the design, such as early exit points to increase the utilization of the array for small sequence sizes, as well as on-chip buffering to enable the processing of long sequences effectively. We implemented the design on the Convey supercomputing platform. Experimental results show that the improved pair-HMMs algorithm achieves a speedup of 67x, compared to software-only execution. The main limitation in the FPGA implementation is related to the complex floating point calculations needed by the pair-HMMs forward algorithm. This limits the design frequency to 75MHz.

In the future, we plan to accelerate the pair-HMMs forward algorithm on a GPU platform to investigate its capabilities to efficiently process floating-point operations in parallel.

References

[1] Jay Shendure and Hanlee Ji, Next-generation DNA sequencing. Nature. Biotechnology 26, 2008, 1135–1145

[2] Richard Durbin, et al., 1998. Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids, page90-109

[3] Chuong B. Do, Mahathi S.P. Mahabhashyam, Michael Brudno, and Serafim Batzoglou. ProbCons: Probabilistic consistency-based multiple sequence alignment. Genome Res 2005. 15(2):330-340.

[4] Sayed Mohammad Ebrahim Sahraeian and Byung-Jun Yoon. PicXAA: greedy probabilistic construction of maximum expected accuracy alignment of multiple sequences. Nucleic Acids Res 2010, 38(15): 41-4928.

[5] Yongtao Ye, et al.. GLProbs: Aligning Multiple Sequences Adaptively. IEEE/ACM transactions on computational biology and bioinformatics, vol. 12, No.1, 2015, 1: 67-78.

[6] Bjarne Knudsen, Michael M. Miyamoto, Sequence Alignments and Pair Hidden Markov Models Using Evolutionary History. J. Mol. Biol., 2003, 333, 453-460.

[7] DePristo M, et al., A framework for variation discovery and genotyping using next-generation DNA sequencing data. 2011 NATURE GENETICS 43:491-498

[8] Accelerated BLAST Performance with Tera-BLASTTM: a comparison of FPGA versus GPU and CPU BLAST implementations," May 2013, TimeLogic Division, Active Motif Inc. [Online]. http://www.timelogic.com/documents/TimeLogic_Tera-BLAST_whitepaper_v1.0.pdf

[9] Brett Dutro, Hardware acceleration of the SAMtools variant caller, 2015. https://www.ideals.illinois.edu/bitstream/handle/2142/72860/Brett_Dutro.pdf?sequence=1

[10] Oliver TF, Schmidt B, Jakop Y, Maskell DL. High speed biological sequence analysis with hidden Markov models on reconfigurable platforms. IEEE Transactions on Infromation Technology in Biomedicine, vol. 13, no.5, pp。740-746， 2009.

[11] Yangteng Sun, et al., HMMer acceleration using systolic array based reconfigurable architecture, in Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '09), New York, NY, USA, May 2009

[12] Steven Derrien and Patrice Quinton, Parallelizing HMMER for hardware acceleration on FPGAs, in Proceedings of the International Conference on Application-specific Systems, Architectures and Processors (ASAP '07), pp. 10–17, Montreal, Canada, July 2007

[13] Sean S. Eddy, Accelerated Profile HMM Searches. *PLoS Comput. Biol.*7, Oct. 2011. pcbi:1002195

[14] W. Kurdthongmee, A Modified HMM Forward Algorithm for an Embedded Motion Type Classification. International Journal o fSignal processing System, Vol.2, No.2, Dec., 2014, pp.84-90

[15] The Convey HC-2 ™ Computer Architectural Overview http://www.conveycomputer.com/files/4113/5394/7097/Convey_HC-2_Architectual_Overview.pdf

[16] Hsiang Tsung Kung and Charles Eric Leiserson, 1978. Systolic Arrays for VLSI, Interim report, Department of Computer Science, Carnegie Mellon University

[17] Ernst Houtgast, et al., An FPGA-based systolic array to accelerate the BWA-MEM Genomic Mapping Algorithm (September 2015). International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XV 2015, July 2015, Greece.

[18] https://github.com/MauricioCarneiro/PairHMM/tree/master/test_data

2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)

Maximizing Systolic Array Efficiency to Accelerate the PairHMM Forward Algorithm

Johan Peltenburg, Shanshan Ren, Zaid Al-Ars, Computer Engineering Laboratory, TU Delft
E-mail: {j.w.peltenburg, s.ren, z.al-ars}@tudelft.nl

***Abstract*—In the analysis of next-generation DNA sequencing data, Hidden Markov Models (HMMs) are used to perform variant calling between DNA sequences and a reference genome. The PairHMM model is solved by the Forward Algorithm, for which the performance and power efficiency can be increased tremendously using systolic arrays (SAs) in FPGAs. We model the performance characteristics of such SAs, and propose a novel architecture that allows the computational units to continuously perform useful work on the input data. The implementation achieves up to 90% of the theoretical throughput for a real dataset. The implementation of the proposed architecture achieves more than 2.5x throughput over the state-of-the-art on a similar contemporary platform.**

***Keywords*—High-Throughput Sequencing, GATK, HaplotypeCaller, PairHMM, Systolic Array, FPGA**

I. INTRODUCTION

Next-generation DNA sequencing methods allow cost-effective sampling of DNA [1]. This data is used e.g. to understand and treat human diseases. The analysis of the huge amounts of data resulting from such samples is still a computational challenge today. Hidden Markov Models (HMM) are used during analysis to find pairwise alignments of DNA sequences. More specifically PairHMMs [2] can be used to calculate the probability that two sequences are related, which is called the overall alignment probability. In this work, we consider the alignment probability of a read to a haplotype.

Because of the computational complexity and the data volume, PairHMM calculations in genome analysis pipelines (such as Genome Analysis ToolKit or GATK [3]) take a long time to complete on conventional machines. However, the PairHMM Forward Algorithm, which is also used in the software implementation of the GATK HaplotypeCaller, is an algorithm exhibiting a long datapath. Such algorithms are often good candidates for FPGA implementation. An FPGA accelerator is often able to achieve a high throughput and high power-efficiency. In other research, it has been shown that FPGAs can be suitable candidates to implement the algorithm using Systolic Arrays (SAs). However, a drawback of some architectures is that the computational resources are sometimes under-utilized due to control issues or data padding.

In this work, we attempt to optimize SA utilization, allowing for near continuous processing on all the computational elements of the SA. Our future aim is to implement many small but efficient SAs instead of implementing one large but inefficient SA. Our contributions are as follows:

- We provide a model to calculate the utilization of an SA.
- We analyze architectural alternatives allowing continuous processing of the PairHMM Forward Algorithm.
- We implement one such architecture that is more than 2.5x faster than the state-of-the-art FPGA implementation and 10x faster than a state-of-the-art CPU.

II. BACKGROUND

A. PairHMM Forward Algorithm

Algorithm 1 PairHMM Forward Algorithm used in the GATK HaplotypeCaller

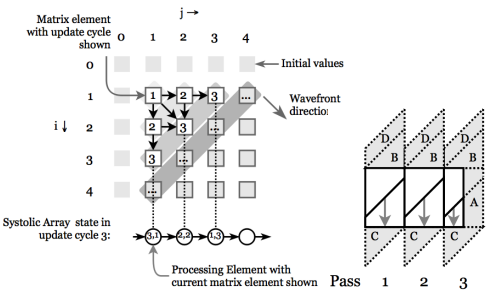
$M \leftarrow I \leftarrow D \leftarrow 0_{X+1,Y+1}$
$D_{0,0\ldots Y} \leftarrow C_{init}$
for $i \leftarrow 1, X$ **do**
  for $j \leftarrow 1, Y$ **do**
    $M_{i,j} \leftarrow \alpha_{i,j} \cdot (\beta_i \cdot M_{i-1,j-1} + \gamma_i \cdot I_{i-1,j-1} + \gamma_i \cdot D_{i-1,j-1})$
    $I_{i,j} \leftarrow \delta_i \cdot M_{i-1,j} + \epsilon_i \cdot I_{i-1,j}$
    $D_{i,j} \leftarrow \eta_i \cdot M_{i,j-1} + \zeta_i \cdot D_{i,j-1}$
return $\sum_{j=0}^{Y} M_{X,j} + I_{X,j}$

The PairHMM Forward Algorithm as implemented in the HaplotypeCaller is seen in Algorithm 1. M, I and D are the matrices for match, insertion and deletion probabilities. $\alpha_{i,j}$ is the emission probability: for each position in the read i it can have two different values, depending on the bases of the read and haplotype at position i and j. β, γ, δ, ϵ, η and ζ are transmission probabilities that only depend on the read position i. In the software implementation, all probabilities are floating-point values. We define X and Y as the length of the read and haplotype, respectively.

When updating some cell (i, j) of the matrices M, I and D, a dependency exists on the values of cells $(i-1, j-1)$, $(i-1, j)$ and $(i, j-1)$. Thus, only matrix cells laying on the anti-diagonals of the matrix can be updated in parallel. Therefore, Algorithm 1 is commonly implemented in hardware using a one-dimensional systolic array (SA) consisting of a number of processing elements (PEs). Each PE implements the inner loop in the algorithm, updating one cell in each of the matrices M, I, and D. During every update cycle, the SA updates the cells on the anti-diagonal of the matrices (sometimes called a 'wavefront'). A simplified diagram of such an SA can be seen in Fig. 1a. As the anti-diagonal grows, the amount of exploitable parallelism grows as well.

When the length of the haplotype (or read) is larger than the number of elements in the SA, the SA can compute the matrices by making multiple vertical (or horizontal) *passes* through the matrix, processing only a subset of columns (or rows) and wrapping back to the top (or side) of the matrix after completion of a pass. This can be seen in Fig. 1b. The values in the last column (or row) in the pass are often stored

978-1-5090-1610-5/16/$31.00 ©2016 IEEE



(a) One pass detail (b) Multiple passes and potential overhead

Fig. 1: An example of how an SA can solve a PairHMM using the Forward Algorithm (Algorithm 1).

in a FIFO buffer. Whenever a pass is shorter than the amount of PEs in the SA, padded data is inserted (Fig. 1b case A).

B. Related work

Earlier research discussed using SAs to solve similar HMM-based algorithms in the field of computational biology [4] [5]. These proposed SA designs introduce overhead when model parameters must be reconfigured between subsequent passes or workloads. Subsequent research such as [6] and [7] show more advanced SA designs, deploying double buffering of model parameters of alternating passes and workloads, allowing for near continuous processing.

More recent work implements the same PairHMM Forward Algorithm as this work in FPGA on the Convey Computer platform, showing higher throughput than single threads of the host processor [8]. However, the architecture introduces overhead when switching between passes, as parameters are shifted into the PEs. In [9], which we consider as the current state-of-the-art FPGA implementation, PEs are partially internally pipelined, achieving a high throughput. This design uses the CAPI interface of the IBM POWER8 platform, which we will also use in this work.

In this paper, we introduce a new architecture that is able to continuously perform *useful* calculations in the PEs of the SA. Once the first input data pair is loaded, our design wastes virtually no cycles due to memory latency or parameter reconfiguration. Thus, the design is able to achieve extremely close to the maximum theoretical performance of a fixed-size SA.

III. Performance model

We define the length of the read and the haplotype as X and Y. The total amount of cell updates required to process the Forward Algorithm is $X \times Y$. A useful measure of performance for the Forward Algorithm is the throughput in number of cell updates per second (CUP/s). In this paper, we will only count *effective* cell updates, which are cell updates that contribute to the final result (i.e. not on padded data).

The throughput of an SA design is affected by the average utilization of the PEs. We observe that while processing the Forward Algorithm with an SA, under-utilization of the PEs may be introduced in several cases (also shown in Fig. 1b):

(A) When data is padded if a pass is not as wide as the SA.
(B) If the PEs in the SA may only work on one pass at a time, under-utilization of the PEs occurs at the start of a pass.
(C) Same as B, but at the bottom of a pass.
(D) When switching between passes, to update the model (α, β, etc.) in the PEs.
(E) When the height of the matrix is shorter than the number of PEs, and more than one pass is required, the read must be padded. Otherwise, the feedback FIFO will not contain any data yet for the first PE to work on in the next pass. (Not shown in Fig. 1b).

We consider an SA of fixed size, thus the overhead introduced in case A and E is inevitable. However, we aim to eliminate the other causes of overhead.

A. Fixed-size systolic array performance

Consider the processing of the Forward Algorithm in an SA where; W is the width of the matrix, H is the height of the matrix and E is the number of PEs in the SA. Also, assume one cell update per clock cycle. In the ideal case, if we would process a large amount of pairs (thereby ignoring initial and final latency), that are of similar size, and if the input data is available at any time at the inputs of the PEs, the average utilization of the whole SA for one pair is given by:

$$\text{Avg. utilization} = \frac{WH}{E\lceil \frac{W}{E} \rceil \cdot \max(E, H)} \tag{1}$$

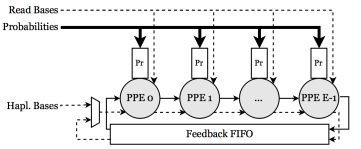
Eq. 1 takes the number of cells in the original matrices and divides this by the number of cells in the padded matrices. This gives the ratio of effective cell updates verses all cell updates (including padding). In the case of such a workload, we may obtain the average number of *effective* cell updates U_{avg} per clock cycle by multiplying the average utilization by the number of PEs in the SA:

$$U_{avg}(W, H, E) = \frac{WH}{\lceil \frac{W}{E} \rceil \cdot \max(E, H)} \tag{2}$$

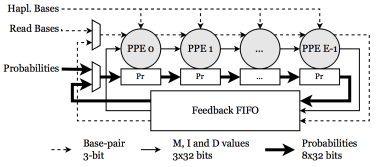
Thus, cells padded to the bottom of the matrix (in each pass, only when $H < E$) and cells padded to the right of the matrix in the final pass are also taken into account.

If the height of the matrix is equal or larger than the number of PEs (i.e. $H \geq E$) and the width of the matrix is an integer multiple of the number of PEs (i.e., $W = nE, n \in \mathbb{Z}_{>0}$), all PEs perform useful work in every pass. In this case, maximum throughput is achieved ($U = E$). This also shows an SA of length $E = 1$ is always maximally efficient (i.e. an SA of this size needs no padding, since passes are of width 1).

Modern FPGAs contain enough computational fabric to implement a large number of PEs. However, the number of SAs cannot be as high, since it quickly becomes bounded by the available memory and interconnect. For example, the FPGA used for this work offers enough resources to implement 112



(a) Architecture HS: haplotype data is streamed in horizontally, read data is streamed in vertically.



(b) Architecture RS: read data is streamed in horizontally, haplotype data is streamed in vertically

Fig. 2: Two SA architectures.

PEs, but the FPGA lacks resources to implement 112 SAs in parallel, requiring 112 controllers, input buffers, feedback FIFOs and other items in the data and control paths. A more feasible combination would be to have, e.g. 7 SAs of 16 PEs each. This work focuses on implementing an architecture for a single SA, that achieves as close to the maximum performance of Eq. 2 as possible.

IV. ALTERNATIVE ARCHITECTURES

A. Alternative architectures

To achieve the maximum performance, the matrix can be mapped onto the SA in two ways. In one, (HS in Fig. 2a), the data that depends on the haplotype position (haplotype bases) is streamed-in at the head of the SA. The data that depends on the read position (probabilities and read bases) is fed vertically into the PEs. In this approach, the matrix is mapped to have the read on the horizontal axis, and the haplotype on the vertical axis of the matrices. The other approach (RS, Fig. 2b) has horizontal and vertical data streams swapped.

All data that is fed *horizontally* can be streamed from input FIFOs into the head of the SA. When reuse of this data is required in a new pass, the feedback FIFO will provide this data and intermediate values that were streamed out of the SA after processing the last column of the previous pass. All data that is fed *vertically* can be distributed to the respective PEs using a bus connected to registers (or RAM).

Although architectures similar to HS are often used (with the exception of [6]), we argue the use of RS. The reason to select RS is related to the sizes of the read and haplotype, X and Y. The haplotype is at least as long as the read, but often much longer. Consider again Eq. 2. When the ratio between fully utilized passes and underutilized passes is high (i.e. when Y is large) the efficiency is also high, since a relatively larger number of passes will have full SA utilization.

Internally, the PEs are pipelined, such that the critical path in the circuit is reduced, allowing higher clock frequencies for the whole SA. The throughput of the SA is directly proportional to its clock frequency.

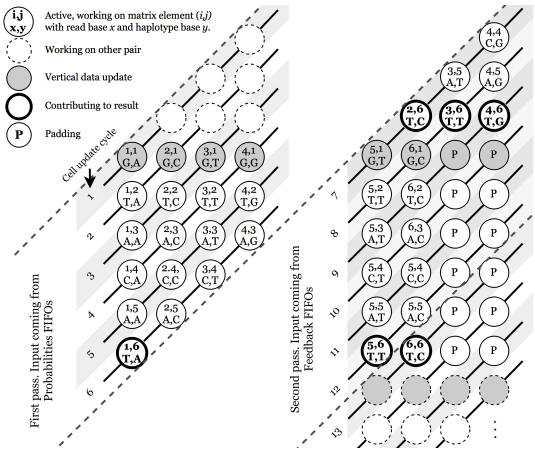


Fig. 3: Example of processing a pair for which the read length $X = 6$, the haplotype length $Y = 6$ and the number of PEs $E = 4$.

B. Maximizing utilization

To achieve maximum utilization, overhead from the cases B, D and C described in Section III must be prevented. This can be done by observing that, during one cell update cycle, the vertical data of *at most* one PE needs to be updated, i.e. at most one PE in the SA will enter a new pass in each cell update cycle. Therefore, a bus connected to the vertical data registers needs to transfer the vertical data of only one PE per cycle.

In this way, any data that is still in the SA from a previous pass or pair does not have to be completely streamed out, allowing cell updates between passes and pairs to take place within the SA (solving case B and C). Furthermore, when the vertical data bus is able to transfer all required data in one cycle, overhead caused by updating model parameters in the PEs can be avoided (solving case D)

An example of continuous processing on the RS architecture is given for the following case: The number of PEs, $E = 4$, the length of the read $X = 6$, the length of the haplotype: $Y = 6$, the read is 'GTACAT' and the haplotype is 'ACTGTC'.

As shown in Fig. 3, on each anti-diagonal, the state of the complete SA is depicted during one cell update cycle, and superimposed over the matrix cells of a pass. For each cell update cycle, the vertical data of *at most* one PE must be updated. Similarly, the output of at most one PE holds data contributing to the final result. Therefore, the M and I output of each PE are logically OR-ed with each other and sent to an accumulator. This implements the last line of the procedure in Algorithm 1. By setting the haplotype and read base to a value called "Padding" (denoted by 'P' in the figure), the PEs output will be invalidated.

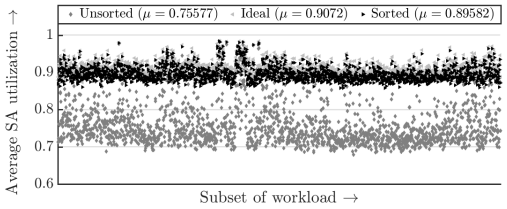


Fig. 4: Effect of sorting on the efficiency of the SA, with E=16.

C. Control mechanism

Since PEs are internally pipelined (Section IV-A), to allow multiple PairHMMs to run in each of the pipeline slots, one could use BRAM and allocate a specific region for each of the N pairs that is active in an N stage pipeline. However, such a control mechanism is complex, since it must track all SA control signals, as well as RAM addresses, for each of the N pipeline slots independently. At the side of the memory interface, it must keep track of N pointers, data counters, and other control information.

The control mechanism can be extremely simplified by allowing the smallest unit of processing to be *batches* of N pairs. By implementing FIFOs for the input data, the host can prepare a batch of N pairs to be processed, ordering the batch in memory in such a way that the accelerator itself does not have to deal with ordering at all. The accelerator keeps track of control signals of only one batch instead of keeping track of all control signals for each of the N pairs.

Although simplifying control complexity, batches have a minor drawback in terms of performance; if the pairs contained in the batch are of completely different sizes, smaller pairs require a lot of padding, in turn decreasing SA efficiency.

Consider the processing of N pairs in a batch, where the n-th pair has read length X_n and haplotype length Y_n. The total amount of work required in cell updates U_{req} to process the batch is given by:

$$U_{req} = \sum_{n=0}^{N-1} X_n Y_n \tag{3}$$

When the amount of work done on a batch U_{batch} is determined by the largest read and haplotype, it can be calculated (containing overhead due to padding) using Eq. 2 as follows:

$$U_{batch} = N \cdot U_{avg}(\max_n Y_n, \max_n X_n, E) \tag{4}$$

Dividing Eq. 3 by Eq. 4 gives the efficiency per batch.

When the read and haplotype lengths are different, the SA has low efficiency due to abundant padding. A large portion of this drawback can be mitigated by sorting the pairs by number of passes required, then sorting each list of pairs with the same number of passes by read size. After sorting, the batches are created by the host and sent to the accelerator. When the workload is very large, sorting makes it likely that haplotypes and reads inside a batch share a similar number of passes and read size.

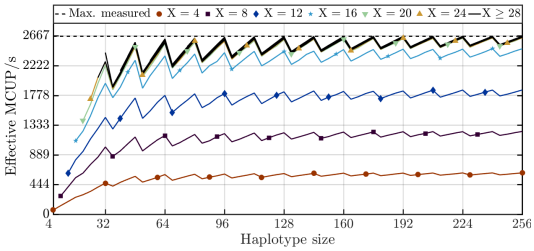


Fig. 5: Synthetic benchmark. PEs: $E = 16$. Workload size: 2^{14}. Step size: 4. Read size: X. Theoretical maximum throughput: 2667 MCUP/s. Max. measured: 2661 MCUP/s.

To reduce the sorting time, we sort only small subsets of the workload. For the whole genome sequencing dataset we used for this work (see Section VI), we split the workload into 1832 subsets of 2^{14} pairs and sort them. In Fig. 4, we compare it to the SA utilization when using unsorted subsets and the ideal utilization given by Eq. 2, in the case where we would not use batches, but are able to start working on pairs in independent pipeline slots. We find that using sorted batches almost achieves ideal performance.

V. Implementation

We implemented architecture RS using an AlphaData ADM-PCIE-7V3 FPGA accelerator card, for which a POWER8 CPU on an IBM Power System S824L (8247-42L) serves as a host. This system offers the Coherent Accelerator Processor Interface (CAPI) to the accelerator through IBMs Power Service Layer (PSL) interface. The memory interface at the host side is therefore similar to [9]. To abstract away the PSL interface, we use the CAPI Streaming Framework from [10].

The SA consists of E Pipelined Processing Elements (PPEs). Each PPE implements the inner loop of Algorithm 1 as a 16-stage pipeline. The maximum number of PPEs we could fit (using Vivado 2016.2) was 112. This bound is determined by the number of DSP blocks. The DSP blocks are used by the floating-point units in the PPEs. The FPGA allows 3600 DSP blocks to be used, but the PSL is distributed as a pre-routed design and prevents the use of a quarter of the DSP blocks. In this work, we implement the SA using $E = 16$ and $E = 32$.

VI. Experimental results

To measure the performance for different sizes, we generate workloads of increasing read (X) and haplotype (Y) size, where $Y \geq X$, in steps of 4. Each workload contains 2^{14} pairs. The performance for each workload is shown in Fig. 5. Our SA runs at 166.7 MHz, thus the maximum theoretical throughput is $E \cdot f$ in cell updates per second (CUP/s).

Padding in the horizontal direction (when $X < E$), deteriorates the throughput, as the utilization of the SA is very low. When there is no padding in the horizontal direction, the throughput quickly grows towards the maximum theoretical throughput. Also, the effect of having haplotype sizes of integer multiples of the number of PEs is clearly visible.

TABLE I: FPGA post-routing power estimate and area

Part	LUTs	Registers	RAM36	DSP	Power(W)
Available: 7VX690	433200	866400	1470	3600	
16 PEs + interfaces	119937	140397	473	378	11.212
16 PEs, this work only	47346	60525	181	354	2.721
32 PEs + interfaces	163450	189085	473	730	13.213
32 PEs, this work only	90862	109213	181	706	4.585

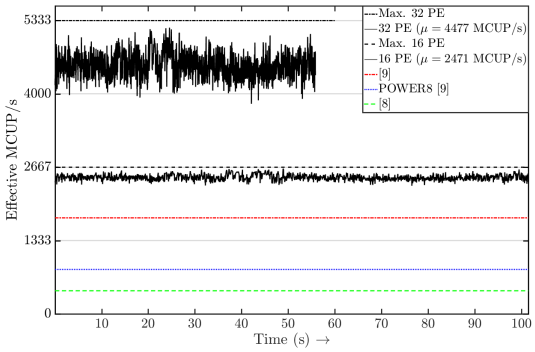


Fig. 6: SA throughput using a real dataset with $E = 16$ and $E = 32$. Subsets size 2^{14}

In this case, the performance nears the maximum theoretical throughput. The highest throughput measured was 99.76% of the maximum. The last bit of overhead is introduced by the memory latency at initialization and termination.

For a realistic benchmark, we use the same dataset as the work presented in [9] (whole human genome dataset G 15512.HCCI954.1 mapped to chromosome 10). The dataset contains over 30 million pairs. We split and sort the dataset in subsets of 2^{14} pairs. The results for sizes $E = 16$ and $E = 32$, the maximum theoretical throughput for each SA, the reported throughput of [9] and [8] and the reported maximum for the POWER8 host CPU are shown in Fig. 6. For $E = 32$, we achieve a throughput of 84% of the maximum performance; for $E = 16$, this is 93%. The lower throughput for $E = 32$ is caused by the large number of reads in the dataset of which the size is smaller than E, resulting in much variation. However, for the SA with $E = 16$, we observe that the utilization is higher, since padding occurs less. Although for $E = 32$, the SA is twice as long as for $E = 16$, the run-time is only 1.8x lower. Furthermore, with the same amount of processing elements, our architecture shows an average improvement of throughput of 2.5x over the state-of-the-art. With half the processing elements, our implementation achieves a 1.4x higher throughput.

In Table I the area statistics of the SA design with 16 and 32 PEs are shown after placing and routing. We show the logic available in the device, the logic utilization of our system (including interfaces) and for our design only. Moreover, the power estimation of Xilinx Vivado is included. From Table I and Fig. 6, we estimate the power efficiency to be $339 \cdot 10^6$ CUP/J.

VII. CONCLUSION

We analyzed the efficiency of systolic arrays that implement the PairHMM Forward Algorithm to find the overall alignment probability of a read to a haplotype. This paper shows architectures which can implement fixed-size SAs in such a way that the overhead is minimal. We implemented one of the architectures, where the data corresponding to the read position is streamed through the systolic array. This implementation achieves 99.76% of the theoretical maximum performance for a synthetic dataset, and around 90% for a real dataset, depending on the size of the systolic array and the read-haplotype pairs. A systolic array with 32 processing elements is able to calculate the overall alignment probabilities of a whole genome dataset mapped to chromosome 10 in under 60 seconds, while only using approximately one third of the FPGAs DSP resources.

In future work, we aim to implement several small SAs in parallel, such that each SA may achieve a high utilization, increasing the overall throughput.

Acknowledgment- This work was supported by the European Commission in the context of the ARTEMIS project ALMARVI (project #621439).

REFERENCES

[1] J. Shendure and H. Ji, "Next-generation dna sequencing," *Nature biotechnology*, vol. 26, no. 10, pp. 1135–1145, 2008.

[2] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

[3] M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. Del Angel, M. A. Rivas, M. Hanna, *et al.*, "A framework for variation discovery and genotyping using next-generation dna sequencing data," *Nature genetics*, vol. 43, no. 5, pp. 491–498, 2011.

[4] A. C. Jacob, J. M. Lancaster, J. D. Buhler, and R. D. Chamberlain, "Preliminary results in accelerating profile HMM search on FPGAs," in *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–8, IEEE, 2007.

[5] K. Benkrid, P. Velentzas, and S. Kasap, "A high performance reconfigurable core for motif searching using profile HMM," in *Adaptive Hardware and Systems, 2008. AHS'08. NASA/ESA Conference on*, pp. 285–292, IEEE, 2008.

[6] Y. Sun, P. Li, G. Gu, Y. Wen, Y. Liu, and D. Liu, "Accelerating HMMer on FPGAs using systolic array based architecture," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–8, IEEE, 2009.

[7] M. N. M. Isa, K. Benkrid, and T. Clayton, "A novel efficient FPGA architecture for HMMER acceleration," in *2012 International Conference on Reconfigurable Computing and FPGAs*, pp. 1–6, IEEE, 2012.

[8] S. Ren, V.-M. Sima, and Z. Al-Ars, "FPGA acceleration of the pair-HMMs forward algorithm for DNA sequence analysis," in *IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pp. 1465–1470, IEEE, 2015.

[9] 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)*, pp. 1–3, IEEE, 2016.

[10] M. Brobbel, "CAPI Streaming Framework." https://github.com/mbrobbel/capi-streaming-framework, 2016.

4

GPU ACCELERATION OF THE PAIR-HMMS FORWARD ALGORITHM

SUMMARY

In this chapter, we first propose to accelerate the pair-HMMs forward algorithm on GPUs. We present several GPU-based implementations of the pair-HMMs forward algorithm and analyze the performance bottlenecks of these implementations on an NVIDIA Tesla K40 card with various datasets. Based on these results and the characteristics of the GATK HaplotypeCaller (HC), we are able to identify the GPU-based implementations with the highest performance for the various analyzed datasets. Experimental results show that the GPU-based implementations of the pair-HMMs forward algorithm achieve a speedup of up to 5.47x over existing GPU-based implementations.

Next, we focus on integrating the GPU-based implementation of the pair-HMMs forward algorithm into GATK HC to improve its overall performance. In single-threaded mode, the GPU-based GATK HC is 1.71x faster than the baseline implementation and 1.21x faster than the vectorized GATK HC implementation. For multi-process mode, We propose a load-balanced multi-process optimization that divides the genome into regions of different sizes to ensure a more equal distribution of computation load between different processes. The GPU-based implementation achieves up to 2.04x and 1.40x speedup in load-balanced multi-process mode over the baseline implementation and vectorized GATK HC implementation in non-load-balanced multi-process mode, respectively.

This chapter is based on the following papers.

1. **S. Ren**, K.L.M. Bertels, Z. Al-Ars, *Efficient Acceleration of the Pair-HMMs Forward Algorithm for GATK HaplotypeCaller on Graphics Processing Units*, Evolutionary Bioinformatics, 14:1176934318760543, 2018 [Journal]

2. **S. Ren**, K.L.M. Bertels, Z. Al-Ars, *GPU-Accelerated GATK HaplotypeCaller with Load-Balanced Multi-Process Optimization*, 17th annual IEEE International Conference on BioInformatics and BioEngineering (BIBE 2017), October 23-25, 2017 [Conference]

Efficient Acceleration of the Pair-HMMs Forward Algorithm for GATK HaplotypeCaller on Graphics Processing Units

Evolutionary Bioinformatics
Volume 14: 1–12

DOI: 10.1177/1176934318760543
SAGE

Shanshan Ren, Koen Bertels and Zaid Al-Ars

Computer Engineering Lab, Delft University of Technology, Delft, The Netherlands.

ABSTRACT: GATK HaplotypeCaller (HC) is a popular variant caller, which is widely used to identify variants in complex genomes. However, due to its high variants detection accuracy, it suffers from long execution time. In GATK HC, the pair-HMMs forward algorithm accounts for a large percentage of the total execution time. This article proposes to accelerate the pair-HMMs forward algorithm on graphics processing units (GPUs) to improve the performance of GATK HC. This article presents several GPU-based implementations of the pair-HMMs forward algorithm. It also analyzes the performance bottlenecks of the implementations on an NVIDIA Tesla K40 card with various data sets. Based on these results and the characteristics of GATK HC, we are able to identify the GPU-based implementations with the highest performance for the various analyzed data sets. Experimental results show that the GPU-based implementations of the pair-HMMs forward algorithm achieve a speedup of up to 5.47× over existing GPU-based implementations.

KEYWORDS: Pair-HMMs forward algorithm, GPU acceleration, memory access, GATK HaplotypeCaller

RECEIVED: May 19, 2017. **ACCEPTED:** November 17, 2017.

TYPE: Review: Special Collection: Computational Bioinformatics Tools for Evolutionary Genomics

FUNDING: The author(s) received no financial support for the research, authorship, and/or publication of this article.

DECLARATION OF CONFLICTING INTERESTS: The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

CORRESPONDING AUTHOR: Shanshan Ren, Computer Engineering Lab, Delft University of Technology, 2628CD Delft, The Netherlands. Email: s.ren@tudelft.nl

Introduction

Next-generation sequencing (NGS) platforms are able to generate large amounts of DNA sequencing data at low cost, which provides great opportunities to deeply understand human genetics and identify genetic diseases.[1] However, handling the large amount of DNA sequencing data produced by NGS platforms consumes much computation time. This is caused by the computationally intensive genomics analysis tools developed to help researchers study and investigate such DNA data. One such tool is GATK HaplotypeCaller (HC),[2,3] which is a widely used variant caller tool in practice.

Variant callers are used to identify DNA variants by comparing a patient DNA sequencing data with a reference genome. Compared with many other variant callers, GATK HC is highly accurate in detecting variants. However, it comes at the expense of long execution time. Therefore, optimizing GATK HC to make it more efficient is important.

In GATK HC, the pair-HMMs forward algorithm (or PFA) accounts for a large percentage of the total execution time. It is applied to study the overall alignment probability of 2 sequences. Pair-HMMs forward algorithm is a computationally intensive algorithm in GATK HC, which is executed repeatedly millions of times for a typical data set.

To address this computational challenge, graphics processing units (GPUs) and field programmable gate arrays (FPGAs) are commonly used in many bioinformatics tools to accelerate the computationally intensive algorithms and improve their performance.[4,5] Thus, this article accelerates PFA on GPUs to improve the performance of GATK HC.

The contributions of this article can be summarized as follows: (1) evaluate 2 approaches to implement PFA on GPUs, (2) present several GPU-based implementations of PFA based on these 2 approaches and compare their performance with different data sets, and (3) choose one implementation to integrate into GATK HC.

Background

GATK HaplotypeCaller

The GATK HC program consists of 4 main steps.[6] (1) Active regions of the genome which have significant evidence of variation are determined. (2) For each active region, haplotypes are determined based on a de Bruijn–like graph and then haplotypes are realigned against the reference sequence using the Smith-Waterman algorithm. (3) For each active region, PFA is applied to perform a pairwise alignment of each read against each haplotype. (4) Bayes' rule is applied to find the most likely genotypes.

Because the number of reads and the number of haplotypes for each active region are not the same, the number of read-haplotype pairs processed by PFA in the third step is different for each active region.

Pair-HMMs forward algorithm

Pair-HMMs forward algorithm in GATK HC is performed as shown in equations (1) to (3).[7] m and n are the length of the read R and the haplotype H, respectively. $M_{i,j}$ is the overall alignment probability of 2 subsequences $R_1 \dots R_i$ and $H_1 \dots H_j$ when R_i is aligned to H_j. $I_{i,j}$ is the overall alignment probability of $R_1 \dots R_i$ and $H_1 \dots H_j$ when R_i is aligned to a gap.

$D_{i,j}$ is the overall alignment probability of $R_1 \ldots R_i$ and $H_1 \ldots H_j$ when H_j is aligned to a gap.

Initialization:

$$\begin{cases} M_{i,0} = I_{i,0} = D_{i,0} = 0 & (0 \le i \le m) \\ M_{0,j} = I_{0,j} = 0 & (0 \le j \le n) \\ D_{0,\mathrm{j}} = 1/n & (0 \le j \le n) \end{cases} \quad (1)$$

Recurrence:

$$\begin{cases} M_{i,j} = \lambda_{i,j}(\alpha_i M_{i-1,j-1} + \beta_i I_{i-1,j-1} + \beta_i D_{i-1,j-1}) \\ I_{i,j} = \delta_i M_{i-1,j} + \varepsilon_i I_{i-1,j} \\ D_{i,j} = \zeta_i M_{i,j-1} + \varepsilon_i D_{i,j-1} \end{cases} \quad (2)$$

Termination:

$$Result = \sum_{j=1}^{n} (M_{m,j} + I_{m,j}) \quad (3)$$

α_i, β_i, δ_i, ε_i, ζ_i, and η_i are transmission probabilities that depend on the read position i. In GATK HC, β_i and ε_i are set to be constant. $\lambda_{i,j}$ is the emission probability. Equation (4) shows how to calculate $\lambda_{i,j}$, where Q_i is the base quality score of the read at position i, R_i and H_j are the value of the read base at position i and the haplotype base at position j, respectively:

$$\lambda_{i,j} = \begin{cases} Q_i / 3 & (if \quad R_i \ne H_j) \\ 1 - Q_i & (if \quad R_i = H_j) \end{cases} \quad (4)$$

The pseudocode of PFA is illustreated in Algorithm 1. The input data include 6 arrays and 2 integers. Among them, $R[\,]$ and $H[\,]$ are used to store read bases and haplotype bases; $Q[\,]$ is used to store the base quality of the read; $\alpha[\,]$, $\delta[\,]$, and $\zeta[\,]$ are used to store transmission probabilities; m and n are the length of read and haplotype, respectively. The output is the overall alignment probability.

Algorithm 1 employs a 2-layer loop to calculate the elements of 3 matrices. Hence, the computational complexity of PFA is $O(mn)$. As shown in Algorithm 1, $M_{i,j}$, $I_{i,j}$, and $D_{i,j}$ are only decided by the left, top-left, and top neighbor elements of the 3 matrices. This implies that the elements on the same antidiagonal do not have data dependency, which results in the inherent parallelism of PFA. Thus, elements on an antidiagonal are able to be calculated in parallel.

GPU architecture

Modern GPUs are widely applied to accelerate computationally intensive algorithms. For NVIDIA GPUs, there are many cores which are able to execute in parallel, and all of the cores are organized into several groups, which are called streaming multiprocessors.

NVIDIA proposes CUDA to help users to efficiently perform general computing. When a GPU kernel is launched by the host processor, there are many threads produced on the GPU. These threads on GPUs are managed by CUDA using a 2-level thread hierarchy: block and grid. Threads produced by a GPU kernel are grouped into many blocks and these blocks are grouped in a grid. Threads produced by different GPU kernels are in different grids. A GPU card can execute one or more grids and a streaming multiprocessors can execute one or more blocks.

Moreover, threads with consecutive thread indexes in the same block are bundled into groups, which are called *warps*. For many NVIDIA GPUs, the size of warp is 32. In addition, due to the Single Instruction Multiple Thread (SIMT) execution model, threads in a warp execute each instruction in lock step.

CUDA also introduces a memory hierarchy, which includes global memory, texture memory/cache, constant memory/cache, local memory, shared memory, and registers. Due to the characteristics of input data of PFA and implementation designed in this article, only the global memory, constant memory/cache, shared memory, and registers are used.

Figure 1 shows a simplified representation of the CUDA memory hierarchy. Constant memory is to store data which would not change during execution to reduce memory bandwidth. Global memory is accessed by all the threads on a GPU. As it resides on the device DRAM, the latency of the global memory access is high. Coalescing global memory accesses is useful to decrease the latency of total global memory accesses. For the GPU used in this article, the width of one global memory access is 128 bytes. If each thread in the same warp loads data (4 bytes, for example) stored at unordered different addresses from global memory, there would be 32 sequential global memory accesses in the worst-case situation. However, if all the accesses are coalesced, which means the data are stored at neighboring addresses, there will be only one global memory access.

However, registers and shared memory are owned by each streaming multiprocessor. Each block running on a streaming multiprocessor has a private space of the shared memory, which is only accessible to the threads in that block, whereas each thread running on the streaming multiprocessor has private registers, which are not accessible to other threads. Because shared memory and registers are scarce resources for each streaming multiprocessor, they limit the number of threads and blocks running on a streaming multiprocessor.

Related work

Most research published regarding the optimization of GATK HC focused on increasing the performance of PFA. Intel and IBM researchers adopt vector instructions on their respective processors[8,9] to decrease the execution time by exploiting the inherent parallelism of PFA. There are also a couple of reports and publications on FPGA-based and GPU-based hardware acceleration of PFA.

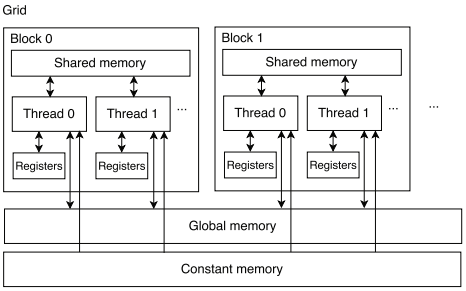


Figure 1. Simplified CUDA memory hierarchy.

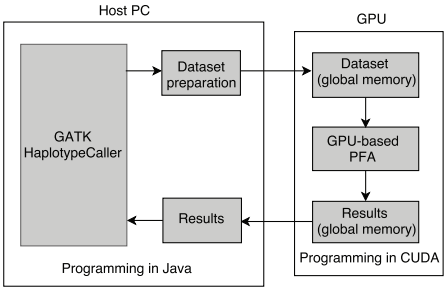


Figure 2. Block diagram of the GPU-based GATK HC implementation. GATK HC indicates GATK HaplotypeCaller; GPU, graphics processing unit.

Research on acceleration of PFA on FPGAs can be found in previous works.[10–14] Ren et al[10] used a systolic array to implement PFA on FPGAs, which exploits the inherent parallelism of PFA. Ito and Ohara[11] proposed pipelined processing elements within a systolic array. Peltenburg et al[12] reduced the overhead in the systolic array to improve the performance of the FPGA-based implementation of PFA. Altera[13] mapped the algorithm to a 2-dimensional systolic array, whereas Huang et al[14] mapped the algorithm to a ring-based systolic array.

Carneiro[15] and Ren et al[16] exploited GPU to accelerate PFA. Carneiro[15] implemented PFA on several NVIDIA GPUs and reported the runtime of their implementations, without describing the implementation details. Ren et al[16] proposed various GPU-based implementations of PFA by investigating 2 different acceleration approaches: intertask and intratask parallelization. In intertask parallelization, PFA is mapped to a single thread, such that each thread implements PFA independently. In intratask parallelization, PFA is mapped on multiple threads in a single block, instead of a single thread. It exploits the inherent parallelism of PFA, reducing the computational complexity of the algorithm to $O(m+n)$. However, it decreases the number of instances of PFA running in parallel on GPUs.

In this article, we analyze these 2 acceleration approaches in detail and compare the performance of several implementations of each approach using various data sets. Compared with the GPU-based implementation on NVIDIA Tesla K40 reported by Huang et al,[14] our implementations are up to 5.47× faster. Moreover, one GPU-based implementation of PFA is selected to integrate into GATK HC.

Methods

First, we present the general design of the GPU-based GATK HC implementation. We then focus on the 2 GPU acceleration approaches and describe their implementations in detail.

General design

Figure 2 shows a block diagram of the GPU-based GATK HC. On the host PC, data sets of read-haplotype pairs are produced during the execution of GATK HC. The size of the data sets is variable, ranging from a couple to 100 000s of pairs depending on the input DNA data. When a data set is produced, the host preprocesses the data set, copies the data set to GPU, launches the GPU kernel to execute PFA, and then copies the results back.

On the GPU, threads load the data set from the global memory, execute PFA independently or cooperatively, and write the results to the global memory.

In addition, as CUDA is not able to communicate with JAVA directly, JCUDA is used to connect the JAVA and CUDA code.

Intertask implementations

In the intertask approach, each thread implements PFA independently. The execution trace of each thread is similar to that described in Algorithm 1. However, extra operations are added to take advantage of the CUDA memory hierarchy.

We first present the naive implementation of the intertask approach, the pseudocode of which is illustrated in Algorithm 2. As shown in Algorithm 2, each thread exploits a 2-level loop to calculate the elements of the matrices.

Due to the limitations of the shared memory and registers size on the GPU, each thread cannot load the input data into the shared memory and registers in advance. All of the input data are loaded into the shared memory and registers when they are being processed. To decrease the number of global memory accesses required by the input data, the outer loop of the 2-level loop iterates through the read bases and the inner loop iterates through the haplotype bases. In this way, the base quality score (Q_i) and the transmission probabilities (α_i, δ_i, and ζ_i) of the read are loaded only once. Otherwise, these data are loaded many times.

Because each element is decided by the left, top-left, and top neighbor elements of the matrices, the intermediate results of PFA do not need to be stored for the entire duration of the execution time. As such, each thread uses 3 registers (MN, IN, and DN) to store the left neighbor elements, a register (MID) to store the result of a series of calculations of the top-left neighbor elements, and 3 vectors ($MM_{0...n}$, $II_{0...n}$, and $DD_{0...n}$) in the global

ALGORITHM 1. PSEUDOCODE OF PFA IN THE GATK HAPLOTYPECALLER.

1: **function PFA** $H[\ ]$, $R[\ ]$, $Q[\ ]$, $\alpha[\ ]$, $\delta[\ ]$, $\zeta[\ ]$, m, n

2: $M \leftarrow 0$ $I \leftarrow 0$ $D \leftarrow 0$

3: $D_{0,0...n} \leftarrow 1/n$

4: $\beta_{0...m} \leftarrow 0.9$

5: $\epsilon_{0...m} \leftarrow 0.1$

6: **for** $i \leftarrow 1,m$ **do**

7: **for** $j \leftarrow 1,n$ **do**

8: **if** R [i]==H[j] **then**

9: $\lambda_{i,j} \leftarrow Q_i / 3$

10: **else**

11: $\lambda_{i,j} \leftarrow 1 - Q_i$

12: **end if**

13: $M_{i,j} \leftarrow \lambda_{i,j}(\alpha_i M_{i-1,j-1} + \beta_i I_{i-1,j-1} + \beta_i D_{i-1,j-1})$

14: $I_{i,j} \leftarrow \delta_i M_{i-1,j} + \epsilon_i I_{i-1,j}$

15: $D_{i,j} \leftarrow \zeta_i M_{i,j-1} + \epsilon_i D_{i,j-1}$

16: **end for**

17: **end for**

18: **return** $\sum_{j=1}^{n}(M_{m,j} + I_{m,j})$

19: **end function**

memory to store the top neighbor elements. Using MID avoids loading the top neighbor elements from the global memory twice.

As shown in Algorithm 2, each iteration of the inner loop loads and stores 3 values from/into the 3 vectors, which requires $6 \times m \times n$ global memory accesses. Because the latency of global memory access is very high, it is necessary to decrease the global memory accesses required by the intermediate results. Hence, the tiling technique[17] is employed. The size of a tile is the number of the successive elements in one column covered by the tile.

The differences between the naive implementation and the tile-based implementation are as follows (the tile size is k): (1) the iteration times of the outer loop of the tile-based implementation is m/k and (2) each iteration of the inner loop of the tile-based implementation calculates a tile, which stands for k successive elements in a column, instead of one element. Figures 3 and 4 show the execution trace of the naive implementation and the tile-based implementation (the tile size is 2), which explain these 2 differences. In Figure 4, the iteration time of the outer loop is $6/2 = 3$ and each iteration of the inner loop calculates 2 elements.

In the inner loop of the tile-based implementation, 3 values from the 3 vectors are loaded to calculate the first element of a

ALGORITHM 2. PSEUDOCODE OF THE NAIVE IMPLEMENTATION OF THE INTERTASK APPROACH.

procedure PFA($H[\ ]$, $R[\ ]$, $Q[\ ]$, $\alpha[\ ]$, $\delta[\ ]$, $\zeta[\ ]$, m, n**)**

for $i \leftarrow 1,m$ **do**

$r \leftarrow R_i$

$q \leftarrow Q_i$

$\alpha \leftarrow \alpha_i$

$\delta \leftarrow \delta_i$

$\zeta \leftarrow \zeta_i$

for $j \leftarrow 1,n$ **do**

$h \leftarrow H_i$

if $i > 1$ **then**

$MU \leftarrow MM_j$

$IU \leftarrow II_j$

$DU \leftarrow DD_j$

else

$MU \leftarrow IU \leftarrow 0$

$DU \leftarrow \frac{1}{n}$

$MID \leftarrow 0.9 \cdot DU$

end if

if $H = R$ **then**

$\lambda \leftarrow q / 3$

else

$\lambda \leftarrow 1 - q$

end if

$DN \leftarrow \zeta \cdot MN + 0.1 \cdot DN$

$MN \leftarrow \lambda \cdot MID$

$IN \leftarrow \delta \cdot MU + 0.1 \cdot IU$

$MID \leftarrow \alpha \cdot MU + 0.9 \cdot IU + 0.9 \cdot DU$

$MM_j \leftarrow MN$

$II_j \leftarrow IN$

$DD_j \leftarrow DN$

if $i = m$ **then**

$result = MM_j + II_j$

end if

end for

end for

return *result*

end procedure

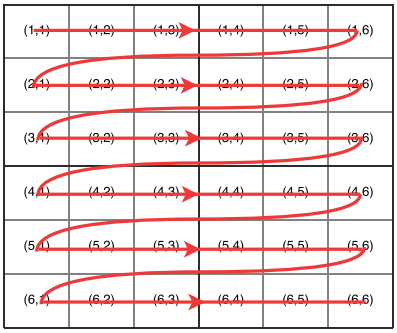


Figure 3. Execution trace of the naive implementation of the intertask approach.

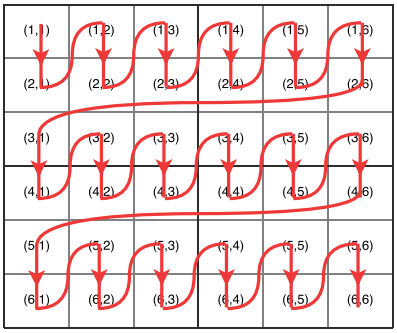


Figure 4. Execution trace of the tile-based implementation of the intertask approach (the tile size is 2).

tile and 3 values of the last element of a tile are stored in the 3 vectors. Thus, the number of global memory accesses required by the intermediate results is $(6\times m\times n)/k$ using tiling technique.

However, the cost of reducing the global memory accesses required by the intermediate results is that each thread uses more shared memory and registers to store left neighbor elements of each tile.

Data set preparation. To better use the GPU computation capabilities, the data sets need to be converted before transferring them to the GPU.

Although for the intertask approach all threads implement PFA independently, if the iteration numbers of the outer loop and the inner loop of each PFA are not the same, threads in a warp need to wait for each other because of the SIMT execution model. Thus, the data sets are sorted first according to the length of reads and then according to the length of haplotypes.

After sorting, 32 read-haplotype pairs are processed by 32 threads in the same warp. To coalesce global memory accesses of the input data, each group of 32 read-haplotype pairs is stored in an interlaced fashion.

Figure 5 shows how the read bases and haplotype bases are stored. Take haplotypes for example. We write the first 4 characters of the first haplotype, after which write the first 4 characters of the second haplotype, and so on. This way, the first 4 characters of 32 haplotypes make up 128 bytes, which could be loaded by one coalesced global memory access. For haplotypes shorter than the longest haplotype in the group, they are padded with dummy characters.

The transmission probabilities and base quality score of 32 reads in each group are also written in an interlaced fashion. As they are single-precision floating point numbers, for each 128 bytes, we only write 1 number instead of 4 numbers. For reads shorter than the longest read in the group, the transmission probabilities and base quality score are padded with dummy numbers.

Intermediate results. For each thread, each iteration includes 6 global memory accesses required by the intermediate results. If these global memory accesses of 32 threads in a warp are non-coalesced, there will be 6×32 global memory accesses in the worst-case situation.

As mentioned before, the intermediate results of each thread are stored in 3 vectors. To reduce global memory accesses, we use 3 big vectors to store the intermediate results of 32 threads in a warp, which are stored in an interlaced fashion. This way, each thread loads/writes the intermediate results from neighboring addresses. Because the intermediate results are single floating point numbers, 1 global memory access is able to satisfy the load/write requirements of 32 threads. Thus, there are 6 global memory accesses for 32 threads in a warp instead of 6×32 of each iteration.

Intratask implementations

In the intratask approach, threads in a block implement PFA cooperatively. The execute trace of each thread is different from that described in Algorithm 1.

We first present the naive implementation of the intratask approach. Depending on the number of threads in a block and the length of the read, there are 3 cases to analyze. We start with the simplest case, in which the number of threads in a block is equal to the length of the read. The execution trace of this case is shown in Figure 6A, in which the number of threads in a block and the length of the read is 4. As shown in Figure 6A, each thread calculates the elements in one column. For example, thread 0 (T0) calculates the elements in the first column. At each step, threads calculate the elements on an antidiagonal. For example, at step 3, T0 calculates $M_{3,1}$, $D_{3,1}$, and $I_{3,1}$; T1 calculates $M_{2,2}$, $D_{2,2}$, and $I_{2,2}$; and T2 calculates $M_{1,3}$, $D_{1,3}$, and $I_{1,3}$. Because there are $(m+n-1)$ antidiagonals in the matrices, the computational complexity of the naive implementation is $O(m+n)$.

The second case is that the number of threads in a block is bigger than the length of the read. The execution trace of the second case is similar to Figure 6A. However, there are some threads that remain idle during the whole execution period as

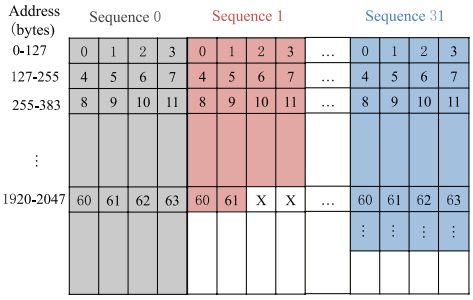


Figure 5. Writing bases of reads and haplotypes in an interlaced fashion.

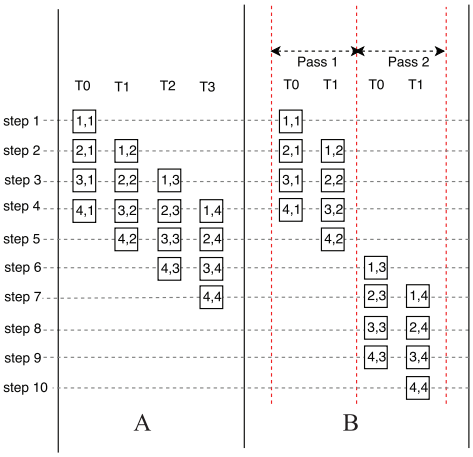


Figure 6. Execution trace of the naive implementation of the intratask approach (A) *without passes* (B) *with passes*.

the number of threads is bigger than the number of columns to be calculated.

The third case is that the number of threads in a block is smaller than the length of the read. The calculation is divided into several passes, which is shown in Figure 6B. The number of threads in a block is 2 and the length of the read is 4. Hence, there are in total 2 passes. In each pass, the execution trace is similar to Figure 6A.

For the intertask approach, because each block calculates one PFA, the input data of one PFA are able to be stored in the shared memory and registers in advance. Each read base and its corresponding base quality score and transmission probabilities are stored in the registers of each thread, whereas the haplotype bases are stored in the shared memory. The intermediate results produced by each thread are stored in 3 vectors in the shared memory. In addition, for the third case, 3 vectors in the global memory are used to store the intermediate results produced by the last thread of each pass, which will be used in the next pass.

For the intratask approach, the synchronization function is called to ensure that all the threads in the same block are synchronized and finish reading/writing the intermediate results in the shared memory. In each step, the synchronization function is called twice. Thus, the synchronization call will be called $O(2(m+n))$ times in total. However, the cost of synchronization call is high because it makes threads in a block stall to wait for each other. There are 2 solutions to decrease the number of the synchronization function calls.

One solution is to exploit the tiling technique. In the tile-based implementation (the tile size is k), each thread calculates k elements in a column before a synchronization function is called. Figure 7 shows the execution trace of the tile-based implementation (the tile size is 2). As shown in Figure 7, each thread calculates 2 elements in each column every 2 steps and thus the synchronization function is called every 2 steps. In this way, there are only $O(2(m+\lceil n/k \rceil))$ synchronization function calls in the tile-based implementation (the tile size is k). However, more shared memory and registers are used to store the intermediate results. Moreover, the number of execution steps of the tile-based implementation is more than that of the naive implementation. As shown in Figure 7, the execution steps of the tile-based implementation are 12, whereas the execution steps of the naive implementation would be only 9.

The other solution is the warp-based implementation, in which the number of threads in a block is equal to the number of threads in a warp (32). In this way, threads in a warp implement PFA cooperatively. Because the threads in the same warp work in the SIMT execution model, there is no need to call synchronization function in the warp-based implementation. Moreover, because threads within a warp can use shuffle instructions to exchange data, the intermediate results produced by each thread are not stored in the shared memory. The only intermediate results stored in the shared memory are the intermediate results between passes for the third case.

However, the warp-based implementation cannot effectively use the resources on GPUs because the number of threads in a block is small. One method to solve this problem is to increase the number of warps in a block and make each warp implement PFA independently. For example, if the number of threads in a block is 256, there are 8 warps in the block and each warp implements PFA independently. In the improved warp-based implementation, the intermediate results between passes for the third case produced by each warp are stored in the global memory.

Data set preparation. In the intratask approach, the read bases, base score quality, and the transmission probabilities, which are loaded into the registers of each thread, are written into 6 vectors separately, whereas the haplotype bases, which are loaded into the shared memory, are written into char4 to reduce the global memory accesses.

Intermediate results. The intermediate results inside one pass are stored in the shared memory using 3 vectors, except for the (improved) warp-based implementations (which use shuffle

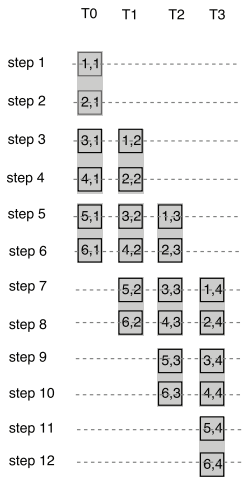


Figure 7. Execution trace of the tile-based implementation of the intratask approach (the tile size is 2).

instructions to exchange data), whereas the intermediate results between passes are stored in the global memory using 3 vectors, except for the warp-based implementation (which stores intermediate data in the shared memory).

Results and Discussion

Experimental setup

IBM Power System S823L (82478-42L) is used to perform all the experiments. This system has 2 IBM Power8 processors, each of which has 10 cores running at 3.6 GHz, 256 GB of DDR3 memory, and an NVIDIA Tesla K40 card. The NVIDIA Tesla K40 card has 2880 cores that run at up to 745 MHz and has a CUDA compute capability of 3.5.

We first compare the performance of these GPU-based PFA implementations with the synthetic and real data sets and then integrate the GPU-based PFA implementations into GATK HC 3.7 and compare the overall performance.

To evaluate these GPU-based PFA implementations, throughput is a key performance metric, which is measured by giga cell updates per second (GCUPS). For a data set of read-haplotype pairs, equation (5) defines how to calculate the value of GCUPS:

$$\frac{\sum_{i=1}^{s} m_i \times n_i}{t \times 10^9} \tag{5}$$

where t is the runtime in seconds, s is the number of the read-haplotype pairs in the data set, m_i and n_i are the length of ith read and ith haplotype, respectively. The runtime t is the computation time of PFA on GPUs.

Table 1. Synthetic data sets of the intertask implementations.

	1	2	3	4	5	6
Read length	24	48	72	96	120	144
Haplotype length	24	48	72	96	120	144

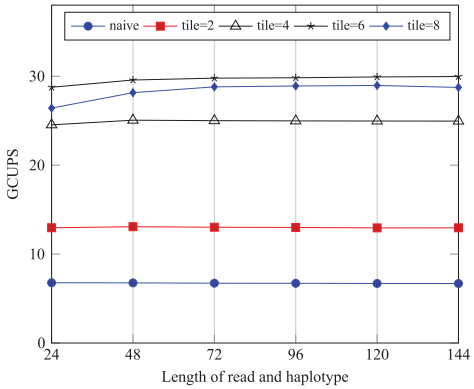


Figure 8. Performance comparison of the intertask implementation on the synthetic data sets.

Implementations of intertask approach

For the tile-based implementations of the intertask approach, the length of reads would slightly affect the performance if it is not a multiple of the tile size. To be fair and find the maximum achievable speedup of every implementations, 6 types of synthetic data sets are used and the length of read and haplotype in each data set are different, as shown in Table 1. In addition, the number of the read-haplotype pairs of each data set is 5×10^5.

Figure 8 shows the throughput of 5 implementations of the intertask approach: naive, tile = 2, tile = 4, tile = 6, and tile = 8. As shown in Figure 8, the throughput of the naive implementation is the lowest over all the implementations. In addition, the throughput of the implementation with tile = 6 is the highest over all the implementations.

We use NVIDIA profiling tools (NVVP) to find the performance bottleneck of these implementations. We run the implementations with data set 6. The profiling results are shown in Table 2. Table 2 shows that the global memory bandwidth reduces when the tile size increases. However, the registers per thread and shared memory per block increase when the tile size increases, which reduces the theoretical occupancy. In addition, the naive, tile = 2, and tile = 4 implementations are bounded by the global memory bandwidth; whereas the other 2 are bounded by the instruction and memory latency, which is caused by the low occupancy.

Table 2 shows that the implementation with tile = 6 strikes a trade-off between the decreasing global memory bandwidth

Table 2. Profiling results of the intertask implementations on synthetic data set 8.

	PERFORMANCE LIMITATION	GLOBAL MEMORY BANDWIDTH, GB/S	REGISTERS PER THREAD	SHARED MEMORY PER BLOCK, BYTES	THEORETICAL OCCUPANCY, %	ACHIEVED OCCUPANCY, %
Naive	Memory bandwidth	207	48	0	62.5	62.1
Tile = 2	Memory bandwidth	201	72	4096	43.8	43.3
Tile = 4	Memory bandwidth	193	73	8192	37.5	37.1
Tile = 6	Instruction and memory latency	154	104	12288	25	24.9
Tile = 8	Instruction and memory latency	101	142	16384	18.8	18.2

requirements as tile size increases, on the one hand, and between the increasing requirements of the instruction and memory latency, on the other hand. This explains the reason why the throughput of the implementation with tile = 6 is the highest on the synthetic data sets.

Implementations of intratask approach

This section compares the performance of the naive, tile = 2, warp-based, and improved warp-based implementations of the intratask approach. Here, the block size of the naive and tile = 2 implementations is 128. Table 3 shows 26 types of synthetic data sets and the length of read and haplotype in each data set are different. Data sets 1 to 21 make no thread idle during each pass for the (improved) warp-based implementations, whereas data sets 22 to 26 make no thread idle during each pass for all the implementations.

Figure 9 shows the throughput of 3 implementations of the intratask approach when the number of the read-haplotype pairs of each data set is 5×10^5. For the warp-based and improved warp-based implementations, if the length of read is the same, the throughput increases with the increase in the haplotype length. In contrast, if the haplotype length is the same, the throughput decreases with the increase in the read length, which is caused by the increased number of costly global memory accesses. In addition, the improved warp-based implementation achieves higher throughput than the warp-based implementation.

As shown in Figure 9, the throughput of the naive and tile = 2 implementations increases with the increase in the read/haplotype length, which is not the case for the (improved) warp-based implementations. Compared with tile = 2 implementation, the naive implementation achieves higher throughput.

Table 4 shows the NVVP profiling results of the 4 implementations running with data set 26. As shown in Table 4, the shared memory bandwidth of the naive implementation is higher than that of the tile = 2 implementation, which is because that the tiling technique reduces shared memory accesses. However, because the tile = 2 implementation has more shared memory and registers to store the intermediate results, its theoretical occupancy is smaller than that of the naive implementation. As shown in Figure 9, the throughput of the tile = 2 implementation is smaller than that of the naive implementation, which indicates that the decrease in the occupancy outweighs the reduction in the number of synchronization calls. If the tile size continues to increase, the theoretical/achieved occupancy will continue to reduce, which results in the decreasing throughput.

As shown in Table 4, the theoretical/achieved occupancy of the improved warp-based implementation is much bigger than that of the warp-based implementation, which is caused by the low GPU resources utilization of the warp-based implementation. Hence, the throughput of the improved warp-based implementation is higher than that of the warp-based implementation, as shown in Figure 9.

In Figure 8, the throughput of the intertask implementations is comparable when the length of the haplotype or read increases, whereas in Figure 9, the throughput of the intratask implementations increases when the length of the haplotype or read increases.

Figure 10 shows the throughput of the intratask implementations when the number of the read-haplotype pairs of each data set is reduced to only 200 (instead of 5×10^5 pairs used for Figure 9). In this figure, the naive implementation achieves the highest throughput for most of the data sets. This is because when the size of data set is small, the improved warp-based implementation does not have enough computation to fully use the GPU resources.

Comparison with other implementations

We compared our GPU-based implementation with other implementations proposed in the previous works.[13-15] The data

set used is the "10s" data set.[18] Despite its small size, this data set has published runtime baseline comparisons for different implementations and platforms.

Table 5 shows the performance of various implementations, including CPU, GPUs, multicores, and FPGAs. The runtime includes the data set preparation time, the computation time on GPU, and the data transfer time between host and GPU. As shown in Table 5, all the GPU-based implementations proposed in this article are faster than the Intel Xeon single core AVX implementation. Moreover, except for the naive intertask implementation, all the implementations proposed in this article are faster than the NVIDIA K40 GPU implementation proposed in the work by Huang et al.[14]

Our best case performance is the improved warp-based implementation. It is 5.47× faster than the K40 GPU implementation, 1.17× faster than the Intel Xeon 24 cores AVX implementation, and 843× faster than the original JAVA implementation. The table also shows that the FPGA-based implementations have the potential to achieve higher performance, albeit at the expense of long development time and the corresponding high design complexity and cost.

Table 3. Synthetic data sets of the intratask implementations (R and H stand for length of read and haplotype, respectively).

	1	2	3	4	5	6	7	8	9
R	32	32	32	32	32	32	32	32	64
H	32	64	96	128	160	192	224	256	64
	10	11	12	13	14	15	16	17	18
R	64	64	64	64	64	64	96	96	96
H	96	128	160	192	224	256	96	128	160
	19	20	21	22	23	24	25	26	
R	96	96	96	128	128	128	128	128	
H	192	224	256	128	160	192	224	256	

Real data set

In this section, 5 intertask implementations (naive, tile = 2, tile = 4, tile = 6, and tile = 8) and 4 intratask implementations (naive, tile = 2, warp-based, and improved warp-based) are compared using a real data set. To produce the real data set, we modified the source code of GATK HC 3.7 to output the

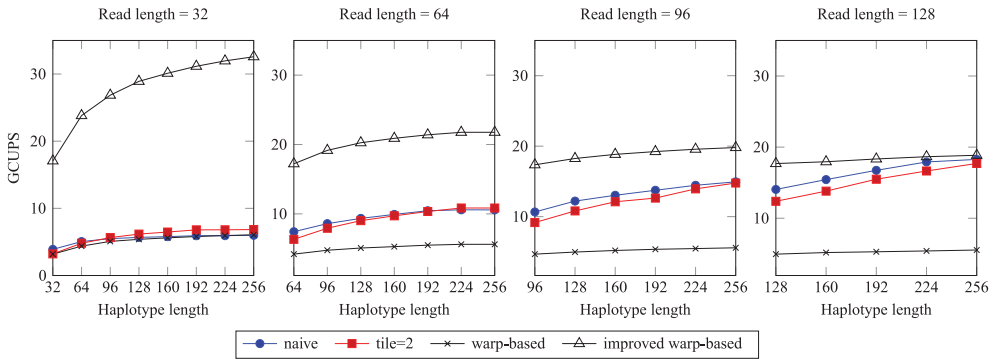


Figure 9. Performance comparison of the intratask implementations on the synthetic data sets (size: 5×10^5). GCUPS indicates giga cell updates per second.

Table 4. Profiling results of the intratask implementations with synthetic data set 26 (warp* stands for the improved warp based).

	PERFORMANCE LIMITATION	SHARED MEMORY BANDWIDTH, GB/S	REGISTERS PER THREAD	SHARED MEMORY PER BLOCK, BYTES	THEORETICAL OCCUPANCY, %	ACHIEVED OCCUPANCY, %
Naive	Memory bandwidth	2213	32	2124	100	99.8
Tile = 2	Memory bandwidth	2005	39	3660	75	74.9
Warp	Instruction and memory latency	362	32	6540	10.9	10.9
Warp*	Compute	262	44	2000	62.5	62.5

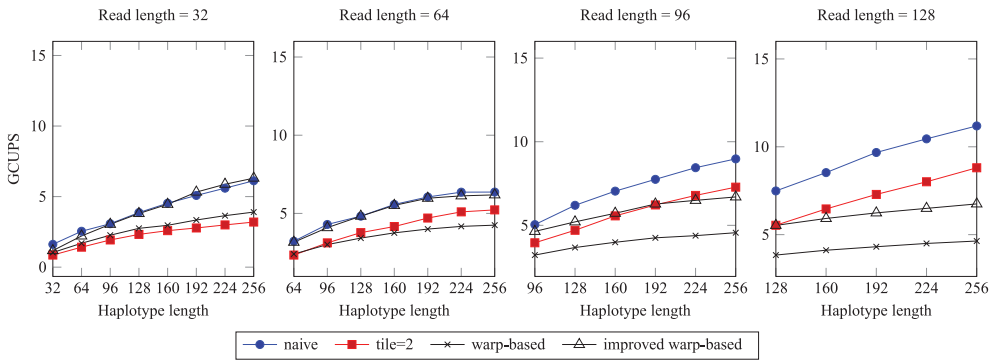


Figure 10. Performance comparison of the intratask implementations on the synthetic data sets (size: 200).

Table 5. Performance comparison of various implementations on a "10s" data set.

IMPLEMENTATIONS	RUNTIME, MS	SPEEDUP
Java on CPU[14]	10800	1×
C++ Baseline[14]	1267	9×
Inter Xeon AVX 1 Core[14]	138	78×
Intel Xeon 24 Cores[14]	15	720×
Alter OpenCL (Arria 10)[13]	2.8	3857×
PE Ring (Arria 10)[14]	2.6	4154×
NVIDIA Tesla K40 GPU[14]	70	154×
Naive intratask	14.2	761×
Tile=2 intratask	15.5	696×
Warp based	20.6	524×
Improved warp based	12.8	843×
Naive intertask	76.6	141×
Tile=2 intertask	45.1	239×
Tile=4 intertask	29.6	365×
Tile=6 intertask	26.4	409×
Tile=8 intertask	24.9	433.7×

read-haplotype pairs of each active region in the third step of the program "Determine likelihoods of the haplotypes". For the modified GATK HC 3.7, chromosome 10 of the whole human data set G15512.HCC1954.1 is used to produce the real data set, which is divided into small chunks with each chunk containing the read-haplotype pairs of one active region. The size of each chunk ranges from 4 to 38 912.

In addition, 2 software-based implementations of PFA are provided for performance comparison: Power8 single core implementation and Power8 20 cores implementation. Both of them are written in the C++ programming language, optimized with the vector instructions and compiled with gcc O3 optimization. Moreover, the Power8 20 cores implementation exploits OpenMP to run on 20 cores.

In Table 6, for the GPU-based implementation, T1 includes the computation time on GPU and the data transfer time between CPU and GPU, T2 is the data set preparation time, and T3 is the total time, which is the sum of T1 and T2. For the software implementations, T3 is the total time and there is no data set preparation. In this section, T3 is the runtime t in equation (5) to calculate GCUPS.

Table 6 shows the performance of 11 implementations on the real data set. The naive intratask implementation is the fastest over all the GPU-based implementations. In addition, the 4 intratask implementations are faster than the 5 intertask implementations. It is mainly because when the number of the read-haplotype pairs in each chunk is small, the intertask implementations cannot use the GPU resources efficiently.

As shown in Table 6, the naive intratask implementation is faster than the improved warp-based implementation. This is because 82% of chunks in the real data include less than 200 read-haplotype pairs. Moreover, Figure 10 shows that when the size of the data set is reduced to 200 pairs, the naive implementation of the intratask approach is faster than the improved warp-based implementation for most of the synthetic data sets.

For the total time (T3), all the GPU-based implementations are faster than the Power8 single core implementation. Specifically, the naive intratask implementation is 11.73× faster than the Power8 single core implementation. However, the Power8 20 cores implementation is faster than all the GPU-based implementations. Regardless of the data set preparation time, the naive intratask implementation (46 s) is much faster than the Power8 20 cores implementation (95 s).

Integration into GATK HC

The GPU-based implementation of PFA with the highest performance for the real data set (which is the naive implementation of the intratask approach) is integrated into GATK 3.7. There are other 2 GATK HC implementations to be compared with: (1) GATK HC with the PFA implemented with JAVA (referred to

Table 6. Performance comparison of implementations on a real data set.

IMPLEMENTATIONS	T1, S	T2, S	T3, S	THROUGHPUT (GCUPS)
Naive intratask	46	53	99	2.60
Tile=2 intratask	47	53	100	2.52
Warp based	81	51	132	1.91
Improved warp based	48	57	105	2.40
Naive intertask	701	73	774	0.33
Tile=2 intertask	382	73	455	0.56
Tile=4 intertask	238	75	312	0.81
Tile=6 intertask	213	75	288	0.88
Tile=8 intertask	217	73	290	0.87
Power8 single core	—	—	1161	0.22
Power8 20 cores	—	—	95	2.65

Abbreviation: GCUPS, giga cell updates per second.

Table 7. Results of the GATK HC implementations.

GATK HC	TOTAL TIME, S	SPEEDUP
Baseline	8034.05	—
Vector	5655.96	1.42×
GPU	4687.08	1.71×

Abbreviations: GATK HC, GATK HaplotypeCaller; GPU, graphics processing unit.

as baseline), which is download from the GATK Web site and (2) GATK HC with the PFA running on the CPU optimized using vector instructions (referred to as Vector), the library of which is implemented by IBM.[9] The data set is chromosome 10 of the whole human genome data set G15512.HCC1954.1.

Although GATK HC is able to run in single-thread and multithread mode, it usually runs in single-thread mode while executing several instances of GATK HC at the same time due to the inefficiency of the multithread mode of the program. Thus, we will only compare the performance of GATK HC running in single-thread mode.

As shown in Table 7, the baseline is slower than the other 2 implementations. Compared with the baseline, the vectorized GATK HC achieves 1.42× speedup and the GPU-based GATK HC achieves 1.71× speedup. In addition, the GPU-based GATK HC is 1.2× faster than the vectorized GATK HC.

Conclusions

In GATK HC, PFA accounts for a large percentage of the total execution time. This article proposes to accelerate PFA on GPUs to improve the performance of GATK HC. Due to the characteristics of PFA, there are 2 approaches to implement it on GPUs: intertask and intratask. This article first presented several GPU-based implementations of PFA for each approach. We executed all the implementations on an NVIDIA Tesla K40 card and compared their performance using different synthetic and real data sets. Experimental results show that our solution achieves a speedup up to 5.47× over other GPU-based implementations. In addition, the naive implementation of the intratask approach is integrated into GATK HC, resulting in an overall speedup of 1.71× over the baseline implementation and 1.2× over the vectorized GATK HC on a single core.

Acknowledgements

The authors wish to thank the Texas Advanced Computing Center (TACC) at the University of Texas at Austin and IBM for the giving access to the IBM Power8 machines used in this paper.

Author Contributions

SR designed and performed the experiments, analyzed the data, and wrote the manuscript. All the authors jointly developed the structure and arguments for the paper, made critical revisions and approved final version.

REFERENCES

1. Shendure J, Ji H. Next-generation DNA sequencing. *Nat Biotech.* 2008;26:1135–1145.
2. McKenna A, Hanna M, Banks E, et al. The genome analysis toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Res.* 2010;20:1297–1303.
3. DePristo M, Banks E, Poplin R, et al. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nature Genet.* 2011;43:491–498.
4. Lu M, Zhao J, Luo Q, et al. GSNP: a DNA single-nucleotide polymorphism detection system with GPU acceleration. Paper presented at: 2011 International Conference on Parallel Processing; September 13–16, 2011; Taipei, Taiwan.
5. Liu CM, Wong T, Wu E, et al. Soap3: ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics.* 2012;28:878.
6. HaplotypeCaller call germline SNPs and Indels via local re-assembly of haplotypes. https://software.broadinstitute.org/gatk/documentation/tooldocs/current/

org_broadinstitute_hellbender_tools_walkers_haplotypecaller_HaplotypeCaller.php. Accessed February 22, 2018.
7. Carneiro M, Poplin R, Biagioli E, et al. Enabling high throughput haplotype analysis through hardware acceleration. https://github.com/MauricioCarneiro/PairHMM/tree/master/doc. Accessed May 15, 2017.
8. Proffitt A. Broad, Intel announce speed improvements to GATK powered by Intel optimizations. http://www.bio-itworld.com/2014/3/20/broad-intel-announce-speed-improvements-gatk-powered-by-intel-optimizations.html. Accessed February 22, 2018.
9. VdAuwera G. Speed up HaplotypeCaller on IBM Power8 systems. https://software.broadinstitute.org/gatk/blog?id=4833. Accessed March 15, 2017.
10. Ren S, Sima VM, Al-Ars Z. FPGA acceleration of the pair-HMMs forward algorithm for DNA sequence analysis. Paper presented at: 2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM); November 9–12, 2015; Washington, DC, pp. 1465-1470. New York, NY: IEEE.
11. Ito M, Ohara M. A power-efficient FPGA accelerator: systolic array with cache-coherent interface for pair-HMM algorithm. Paper presented at: 2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX); July 7, 2016; Yokohama, Japan, pp. 1–3. New York, NY: IEEE.
12. Peltenburg J, Ren S, Al-Ars Z. Maximizing systolic array efficiency to accelerate the PairHMM forward algorithm. Paper presented at: 2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM); December 15-18, 2016; Shenzhen, China, pp. 758–762. New York, NY: IEEE.
13. Altera. Accelerating genomics research with OpenCL and FPGAs. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01262-accelerating-genomics-research-with-opencl-and-fpgas.pdf. Accessed February 22, 2018.
14. Huang S, Manikandan GJ, Ramachandran A, et al. Hardware acceleration of the pair-HMM algorithm for DNA variant calling. Paper presented at: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17; February 22-24, 2017; Monterey, CA, pp. 275–284. New York, NY: ACM.
15. Carneiro M. Accelerating variant calling. https://hpc.mssm.edu/files/Carneiro_workshop.pdf. Accessed March 15, 2017.
16. Ren S, Bertel K, Al-Ars Z. Exploration of alternative GPU implementations of the pair-HMMs forward algorithm. Paper presented at: 2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM); December 15-18, 2016; Shenzhen, China, pp. 902–909. New York, NY: IEEE.
17. Hains D, Cashero Z, Ottenberg M, Bohm W, Rajopadhye S. Improving CUDASW++, a parallelization of Smith-Waterman for CUDA enabled devices. Paper presented at: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum; September 1, 2011; Shanghai, China, pp. 490–501. New York, NY: IEEE.
18. Pair-HMMs forward algorithm test data. https://github.com/MauricioCarneiro/PairHMM/tree/master/test_data. Accessed May 15, 2017.

GPU-Accelerated GATK HaplotypeCaller with Load-Balanced Multi-Process Optimization

Shanshan Ren, Koen Bertels, Zaid Al-Ars
Computer Engineering Lab
Delft University of Technology
2628CD Delft,The Netherlands
{s.ren, k.l.m.bertels, z.al-ars}@tudelft.nl

***Abstract*—Due to its high-throughput and low cost, Next Generation Sequencing (NGS) technology is becoming increasingly popular in many genomics research labs. However, handling the massive raw data generated by the NGS platforms poses a significant computational challenge to genomics analysis tools. This paper presents a GPU acceleration of the GATK HaplotypeCaller (GATK HC), a widely used DNA variant caller in the clinic. Moreover, this paper proposes a load-balanced multi-process optimization of GATK HaplotypeCaller to address its implementation limitation which forces the sequential execution of the program and prevents effective utilization of hardware acceleration. In single-threaded mode, the GPU-based GATK HC is 1.71x and 1.21x faster than the baseline HC implementation and the vectorized GATK HC implementation, respectively. Moreover, the GPU-based implementation achieves up to 2.04x and 1.40x speedup in load-balanced multi-process mode over the baseline implementation and the vectorized GATK HC implementation in non-load-balanced multi-process mode, respectively.**

***Index Terms*—GPU acceleration; GATK HaplotypeCaller; multi-process; pair-HMMs forward algorithm;**

I. Introduction

Next Generation Sequencing (NGS) [1] technology makes DNA sequencing more affordable and accessible than ever before. DNA sequencing is essential for a deep understanding of human genetics and is considered as an enabler on personalized medicine. Many applications for DNA sequence analysis have been developing at a fast rate in the last decade, such as sequence alignment, genome de-novo assembly, variant calling, haplotype phasing and so on.

Variant calling is a crucial step for DNA sequence analysis, which is used to find the positions where a given patient DNA sequence is different from a reference genome in order to detect DNA variants. These variants include SNVs (single nucleotide variations), small insertions/deletions (INDELs) and structural variations (SVs). Many tools (called variant callers) have been proposed to detect variants in order to be used in practice to diagnose genetic disease, for example.

Early variant callers, such as the GATK UnifiedGenotyper [2], SAMtools [3] and VarScan2 [4], detect variants at different positions in isolation. These tools are very effective in detecting SNVs, but are lacking when it comes to the accuracy of identifying INDELs and SVs.

More recent haplotype-based callers, such as the GATK HaplotypeCaller [5], Platypus [6] and freebayes [7], have improved the accuracy of detecting INDELs. INDELs are easily misaligned when mapping the patient DNA to a reference genome, which is ahead of variant calling. In order to correctly identify INDELs, haplotype-based variant callers add extra steps, such as local de-novo assembly of haplotypes [5][6] or direct detection of haplotypes [7]. Moreover, haplotype-based callers enhance the accuracy of identifying SNVs by making use of linkage disequilibrium between nearby variants. The GATK UnifiedGenotyper, for example, is less effective than the GATK HaplotypeCaller in detecting INDELs. However, this comes at the cost of higher execution time.

The GATK HaplotypeCaller (or GATK HC) is widely used in many large-scale sequencing projects. However, GATK HC suffers from long execution time, which would limit its feasibility in many situation. In this paper, we investigate and propose the first GPU-accelerated version GATK HC to improve its performance. Regarding to the optimization of GATK HC, Intel processors and IBM POWER processors both exploit vector instructions to speed up the pairwise alignment kernel [8][9], which is the most time consuming part of GATK HC. There are also a couple of publications on FPGA-based and GPU-based hardware acceleration of the pairwise alignment kernel of GATK HC, but they do not discuss the acceleration of the overall application [10][11].

In this paper, we present an efficient GPU-accelerated implementation of GATK HC and evaluate its effectiveness. An important component of the work is integrating the GPU acceleration into the Java-based GATK HC code and minimizing the incurred overhead. Compared with the baseline implementation, it achieved 1.71x speedup in single-threaded mode. Moreover, we found an important limitation in the GATK HaplotypeCaller implementation which forces the sequential execution of the program and prevents effective utilization of the accelerated part. A load-balanced multi-process optimization is proposed to overcome this limitation, which makes the GPU-based implementation up to 2.04x and 1.40x faster than the baseline implementation and the vectorized implementation, respectively.

The rest of this paper is organized as follows. Section II presents a brief overview of GATK HC. Section III presents the details of the GPU-accelerated implementation of GATK HC and the load-balanced multi-process optimization. Section IV presents the experimental results along with analyses. Section V concludes this paper.

II. BACKGROUND

A. GATK HaplotypeCaller

GATK HC is a Java-based DNA variant caller that is widely used in practice. It is divided into the following four main steps [12].

(i) **Define active regions**—Active regions are determined based on the presence of significant evidence for variation. The following steps only operate on the active regions and ignore the inactive regions.

(ii) **Determine haplotypes**—For each active region, a de Bruijn-like graph is built to reassemble the active region and a list of haplotypes is determined based on the graph. Here, haplotype is a sequence covering the entire length of an active region. Then each haplotype is re-aligned against the reference sequence using the Smith-Waterman algorithm in order to identify potentially variant sites.

(iii) **Determine likelihoods of the haplotypes**—For each active region, a pairwise alignment of each read against each haplotype is performed using the pair-HMMs forward algorithm, which produces a matrix of likelihoods of haplotypes given the reads.

(iv) **Assign genotypes**—For each potential variant site, Bayes' rule is applied to calculate the likelihoods of each genotype using the likelihoods of haplotypes given the reads. The genotype with the largest likelihoods is selected.

GATK HC can run in single-threaded and multi-threaded mode, as shown in Figure 1. When GATK HC runs in single-threaded mode (Figure 1(a)), it first defines active regions (Step(i) in the list above). If there is an active region, it executes Step(ii), Step(iii) and Step(iv) in succession and jumps back to Step(i). If there are no more active regions in Step(i), GATK HC ends execution.

When GATK HC runs in multi-threaded mode (Figure 1(b)), each thread executes in the same way as in single-threaded mode. However, since the size of the active region is not known in advance, Step(i) has to first complete calculating the current active region before the next active region can be calculated. This leads to only one thread executing Step(i) at any time.

In order to investigate which of these steps is most time-consuming and which one is most suitable for GPU-based acceleration, we analyzed and profiled GATK HC (GATK version 3.7) with a typical workload (chromosome 10 of the whole human genome dataset G15512.HCC1954.1).

Firstly, GATK HC was executed in single-threaded mode in order to find which step is most time-consuming. The profiling results in single-threaded mode are shown in Table I. The relative execution time and type of processing are specified in the table as well. As shown in Table I, Step(iii) is most time-consuming, which consumes 48.5% of the total execution time. The main operation of Step(iii) is pairwise alignments implemented by the pair-HMMs forward algorithm, which is executed millions of times. Therefore, acceleration of the pair-HMMs forward algorithm is very important to improve the performance of GATK HC in single-threaded mode.

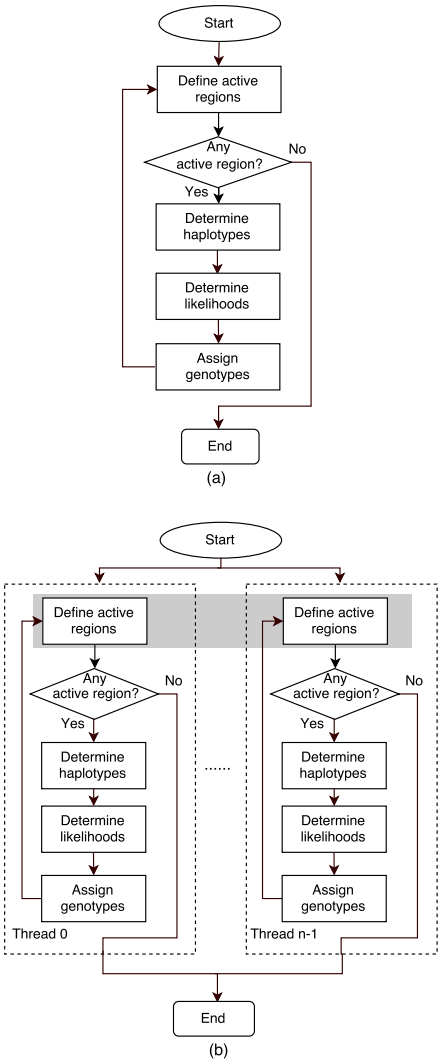


Fig. 1. GATK HC workflow in (a) single-threaded mode and (b) multi-threaded mode

GATK HC then was executed in multi-threaded mode. The total execution time with different number of cores was recorded, which is shown in Figure 2. Moreover, in order to find the influence of the data dependency of Step(i) across multiple threads, we modified GATK HC by disabling the execution of the other steps and made it only executed Step(i).

TABLE I
PROFILING RESULTS OF GATK HAPLOTYPECALLER

Steps	Time	Processing
Define active regions	15.5%	Sequential
Determine haplotypes	34.0%	Sequential
Determine likelihoods	48.5%	Parallel
Assign genotypes	1.3%	Parallel
Other	0.7%	

The total execution time of the modified GATK HC running on 20 cores is 1164 seconds, while the total execution time of original GATK HC running on 20 cores is 1249 seconds. This indicates that when the number of cores is big enough, GATK HC execution time becomes bottlenecked by Step(i) in multi-threaded mode.

In practice, GATK HC is usually executed in multi-process mode instead of in multi-threaded mode to overcome the data dependencies in Step(i). In multi-process mode, the input file is split into chunks based on genome regions and each chunk is processed by GATK HC independently. Although there may be some accuracy loss at the boundaries between consecutive chunks within the same chromosome, the accuracy loss can be negligible in many cases.

The simplest way to divide the input file is to split the genome regions into multiple intervals of equal length. [13][14] proposes to split the genome regions based on the number of reads mapped to each regions, taking the read coverage into consideration. However, both methods do not results in a balanced division of the input file in the case of GATK HC. In this paper, we present a load-balanced multi-process optimization to minimize the total execution time.

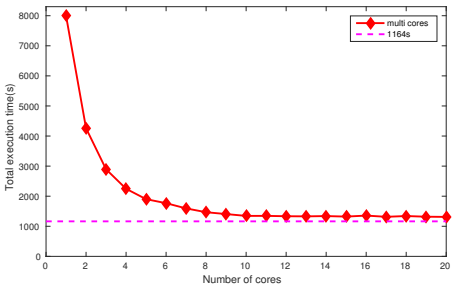


Fig. 2. Multi-threaded execution time of GATK HC

B. Pair-HMMs forward algorithm

In GATK HC, the pair-HMMs forward algorithm takes a read and a haplotype as the input and calculates the overall alignment probability. Previous research on increasing the speed of the pair-HMMs forward algorithm can be found in [8][9][10][11], most of which exploit the inherent parallelism of the algorithm. Intel and IBM researchers employ vector instructions on their respective processors [8][9] to reduce the execution time. These vectorization approaches have been implemented and can be integrated into GATK HC easily. The authors claim a dramatic improvement of the performance of single-threaded GATK HC.

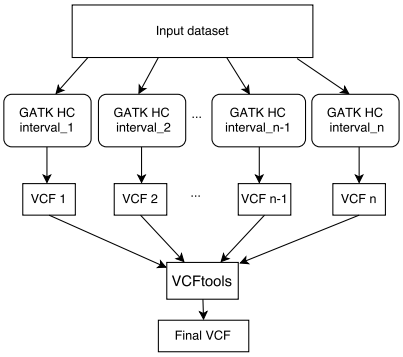


Fig. 3. Data flow of the multi-process GATK HC

On the other hand, [10][11][15] propose FPGA-based implementations of the pair-HMMs forward algorithm. [10] utilizes a systolic array to map the algorithm on FPGAs, while [11] proposes pipelined processing elements within a systolic array and [15] reduces the overhead in the systolic array. However, these implementations have not been integrated into GATK HC.

In addition, [16] proposes several GPU-based implementations of the pair-HMMs forward algorithm and compares these implementations using datasets with different number of read-haplotype pairs. It investigates two different acceleration approaches: the inter-task and intra-task parallelization. According to the paper, when the number of read-haplotype pairs is small, the intra-task GPU-based implementation outperforms all other investigated implementations.

III. METHODS

Our efforts to improve the performance of GATK HC can be divided into two aspects: (1) a load-balanced multi-process parallelization approach to reduce the sequential execution of Step(i) and (2) integration of GPU acceleration of the pair-HMMs forward algorithm into the multi-process GATK HC.

A. Load-balanced multi-process optimization

Figure 3 shows the data flow of the multi-process GATK HC. The GATK HC argument $-L$ is used to confine processing to a specific genome interval instead of actually dividing the input file into small parts. Each interval is processed individually by a GATK HC instance. The output of these GATK HC instances, represented by VCF files, are combined by VCFtools [17] into one VCF file.

As mentioned in Section II-A, Step(ii), Step(iii) and Step(iv) operate only on active regions. If each genome interval has the same number of active regions, this will most probably result in a load-balanced multi-process implementation. Since the active regions are determined by the presence of significant evidence of variation in Step(i), the number of variants in each genome interval is the key parameter to ensure load-balancing.

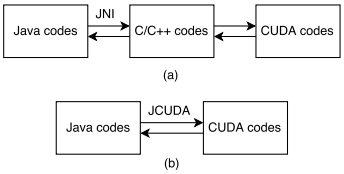


Fig. 4. Two methods of calling CUDA programming modules from Java code

When running GATK HC, the argument $-D$ and a dbsnp file is usually used in order to annotate variants found by GATK HC with the corresponding reference ID. Since the dbsnp file includes all known variants and the positions of these variants on the genome, we can use this file to divide the genome into regions based on the number of known variants. Although GATK HC might find novel variants not present in the dbsnp file, the number of these variants is relatively small and would have a negligible influence on the genome region division.

In order to divide the genome region using the dbsnp file, we modified BCFtools, which uses the HTSlib library to realize fast accesses of the dbsnp file. The modification of BCFtools is to add a function in the vcffilter.c file. The new function first calculates the total number of variants and then outputs the start and end positions of each genome interval, which has the same number of variants.

Besides the number of variants, the number of reads on each genome interval may also influence the load-balancing. One solution is to take these two factors together into consideration. However, it turns out that calculating the total number of reads in the input file is very time consuming. Moreover, for each new input file, the calculation of the total number of reads has to be done again. Thus, the genome region is divided only based on the variant numbers in the dbsnp file.

B. Application level GPU acceleration

In GATK HC, the number of read-haplotype pairs processed by the pair-HMMs forward algorithm depends on the number of the reads and haplotypes found in each active region. For example, the number of read-haplotype pairs ranges from 4 to 38912 for each active region in chromosome 10 of the whole human genome dataset G15512.HCCI954.1. According to [16], for this type of dataset, the intra-task GPU-based implementation of the pair-HMMs forward algorithm is the most effective GPU-based implementation that gives the highest performance.

The GPU acceleration of the pair-HMMs forward algorithm is implemented using CUDA. Although CUDA is able to support various programming languages, it does not support Java, which is the programming language of GATK HC. One method to call CUDA code from Java is using JNI (Java Native Interface), which enables Java code to call modules written in programming languages such as C and C++. As shown by Figure 4(a), this is done by first using JNI to call C++ code from Java, which in turn calls CUDA code.

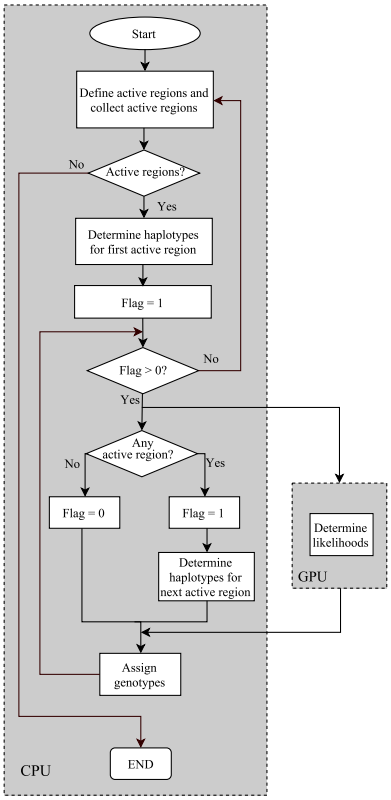


Fig. 5. GATK HC new workflow in single-threaded mode

Another method is to apply JCuda[18], which supplies direct accesses of the CUDA code from Java code. In essence, the design of JCuda also uses JNI to call C++ code and lets C++ code call CUDA programming modules. However, JCuda hides these details from the user. Thus, we can directly call CUDA programming modules from Java codes, as shown by Figure 4(b). For GATK HC, JCuda is employed to call CUDA code from Java.

As shown by Figure 1(a) and (b), the last three steps of the workflow (Determine haplotypes, Determine likelihoods and Assign genotypes) are executed sequentially for each active region. This prevents hiding the execution time of the GPU accelerated part. If the Java implementation of the pair-HMMs forward algorithm is replaced by the GPU implementation, the CPU would be idle and wait for the GPU results. In order to allow hiding GPU execution time, we need to modify the workflow in order to make GPU and CPU run in parallel.

Figure 5 is the new workflow of GATK HC in single-threaded mode. GATK HC first produces and collects multiple active regions and passes these active regions to subsequent steps. Then, Determine haplotypes is executed for the first

active region and the results are transferred to the GPU. Next, Determine likelihoods for the first active region is executed on GPU while Determine haplotypes for the second active region is executed on CPU simultaneously. The rest of the active regions are processed in the same manner. In this way, Determine likelihoods on GPU and Determine haplotypes on CPU are executed simultaneously.

With regard to multi-threaded mode, GATK HC first produces multiple active region, which is still sequentially executed. The other steps of multi-threaded mode are modified in the same way as the steps of single-threaded mode in Figure 5. For each thread, Determine haplotypes on CPU and Determine likelihoods on GPU are executed simultaneously.

IV. Results and Discussion

A. Experimental Setup

In this paper, we use an IBM Power System S824L (82478-42L) to perform experiments and measure performance results. This system includes two IBM Power8 processors (10 cores each) running at 3.42 GHz, 256 GB of DDR3 memory, and an NVIDIA Tesla K40 card. The NVIDIA Tesla K40 card has 2880 cores running at up to 745 MHz, with CUDA compute capability 3.5.

This paper uses GATK version 3.7 for the analysis, which is the latest version of GATK at the time of writing. The paper compare the performance of three GATK HC implementations: (1) the baseline GATK HC with the pair-HMMs forward algorithm implemented in Java, which is downloaded from the GATK website; (2) GATK HC with the pair-HMMs forward algorithm optimized using vector instructions, the library of which is implemented by IBM research [9]; (3) GATK HC with the pair-HMMs forward algorithm accelerated on GPU. The dataset used for the measurement is chromosome 10 of the whole human genome dataset G15512.HCC1954.1.

B. Single-threaded

Table II shows the results of the GATK HC implementations in single-threaded mode. As shown by Table II, the baseline implementation took the longest time (8034.05 seconds). The vectorized GATK HC is 1.42x faster than the baseline implementation. The GPU-based GATK HC is 1.71x and 1.21x faster than the baseline implementation and the vectorized GATK HC implementation, respectively.

TABLE II
Results of GATK HC implementations in single-threaded mode.

GATK HC	Total time [s]	Time pair-HMMs [s]	Speedup
Baseline	8034.05	3676.12	—
Vectorized	5655.96	1289.62	1.42x
GPU-based	4687.08	hidden	1.71x

C. Multi-threaded

Figure 6 shows results of the three GATK HC implementations in multi-threaded mode with thread number ranging from 2 to 20.

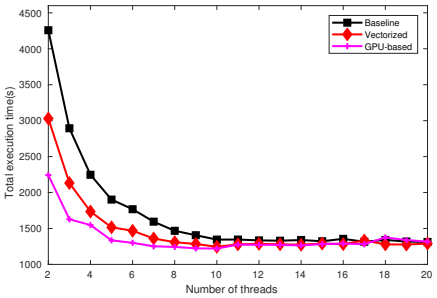


Fig. 6. Total execution time of GATK HC in multi-threaded mode

The execution time of all three implementations decreases while the thread number increases. This is the benefit of using multi-thread, which makes Step(ii) Determine haplotypes, Step(iii) Determine likelihoods and Step(iv) Assign genotypes be executed in parallel.

When the thread number is small, the execution time of the GPU-based implementation is the lowest. However, the execution time of the three implementations becomes very similar when the number of threads becomes bigger than 10. This means that the acceleration of the pair-HMMs forward algorithm does not have big effects on the total execution time when the number of threads is big enough.

D. Multi-process

The three implementations were executed with different number of processes from 2 to 20 both in the load-balanced and non-load-balanced multi-process mode. In the load-balanced multi-process mode, the input file is divided according to the method proposed in Section III-A; in the non-load-balanced multi-process mode, the input file is divided simply using equal number of bases in each segment of the genome.

In order to verify the accuracy of GATK HC in multi-process mode, VCFtools is used to merge the output files produced by the GATK HC instances into one file and compare this file with the output file produced by the baseline implementation in single-threaded mode. The comparison results show that there is no accuracy loss for all the multi-process GATK HC experiments with process number from 2 to 20. For other input file, there might be some accuracy loss.

Generally, the execution time of each GATK HC instance in multi-process mode is not the same. Therefore, the maximal execution time is measured and used as a metric for comparison. Actually, the maximal execution time presents the total time used to handle the input file in multi-process mode. Figure 7 shows the maximal execution time of the three implementations running with different number of processes.

As shown in Figure 7, the maximal execution time of the three implementations in the load-balanced multi-process mode is smaller than these in the non-load-balanced multi-process mode.

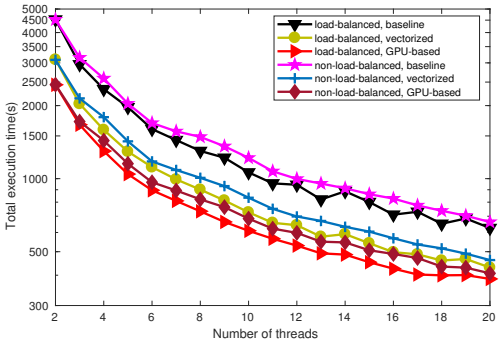


Fig. 7. Maximal execution time of GATK HC implementations in multi-process mode

Moreover, the GPU-based GATK HC implementation in the load-balanced multi-process mode is the fastest. Compared with the baseline implementation and the vectorized GATK HC implementation in the non-load-balanced multi-process mode, the GPU-based GATK HC implementation in the load-balanced multi-process mode achieves up to 2.04x and 1.40x speedup, respectively.

V. CONCLUSIONS

This paper presents a novel implementation of a GPU accelerated GATK HC to improve the overall performance of this computationally intensive application. The paper also proposes a load-balanced multi-process optimization that divides the genome into regions of different sizes to ensure a more equal distribution of computation load between different processes. In addition, the paper compares the GPU-based, vectorized and baseline GATK HC implementations in single-threaded, multi-threaded and multi-process modes.

In single-threaded mode, the GPU-based GATK HC is 1.71x faster than the baseline implementation and 1.21x faster than the vectorized GATK HC implementation. In multi-threaded mode, the GATK HC workflow limits the performance improvement achievable by accelerating the pair-HMMs kernel. In multi-process mode, the GPU-based GATK HC implementation is the fastest. In addition, the GPU-based implementation achieves up to 2.04x and 1.40x speedup in load-balanced multi-process mode over the baseline implementation and vectorized GATK HC implementation in non-load-balanced multi-process mode, respectively.

REFERENCES

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

[2] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark A DePristo. The genome analysis toolkit: A mapreduce framework for analyzing next-generation dna sequencing data. *Genome Research*, 20(9):1297–1303, 09 2010.

[3] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, 08 2009.

[4] Daniel C. Koboldt, Qunyuan Zhang, David E. Larson, Dong Shen, Michael D. McLellan, Ling Lin, Christopher A. Miller, Elaine R. Mardis, Li Ding, and Richard K. Wilson. Varscan 2: Somatic mutation and copy number alteration discovery in cancer by exome sequencing. *Genome Research*, 22(3):568–576, 03 2012.

[5] Geraldine A. Van der Auwera, Mauricio O. Carneiro, Chris Hartl, Ryan Poplin, Guillermo del Angel, Ami Levy-Moonshine, Tadeusz Jordan, Khalid Shakir, David Roazen, Joel Thibault, Eric Banks, Kiran V. Garimella, David Altshuler, Stacey Gabriel, and Mark A. DePristo. From fastq data to high confidence variant calls: the genome analysis toolkit best practices pipeline. *CURRENT PROTOCOLS IN BIOINFORMATICS*, 11(1110):11.10.1–11.10.33, 10 2013.

[6] Andy Rimmer, Hang Phan, Iain Mathieson, Zamin Iqbal, Stephen R.F. Twigg, WGS500 Consortium, Andrew O.M. Wilkie, Gil McVean, and Gerton Lunter. Integrating mapping-, assembly- and haplotype-based approaches for calling variants in clinical sequencing applications. *Nat Genet*, 46(8):912–918, 08 2014.

[7] Erik Garrison and Gabor Marth. Haplotype-based variant detection from short-read sequencing. *arXiv preprint arXiv:1207.3907 [q-bio.GN]*, 2012.

[8] Allison Proffitt. *Broad, Intel Announce Speed Improvements to GATK Powered by Intel Optimizations*. http://www.bio-itworld.com/2014/3/20/broad-intel-announce-speed-improvements-gatk-powered-by-intel-optimizations.html.

[9] Geraldine VdAuwera. *Speed up HaplotypeCaller on IBM POWER8 systems*. https://software.broadinstitute.org/gatk/blog?id=4833.

[10] Shanshan Ren, Vlad M. Sima, and Zaid Al-Ars. Fpga acceleration of the pair-hmms forward algorithm for dna sequence analysis. *2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 1465–1470, 2015.

[11] Megumi Ito and Moriyoshi 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)*, pages 1–3. IEEE, 2016.

[12] *HaplotypeCaller Call germline SNPs and indels via local re-assembly of haplotypes*. https://software.broadinstitute.org/gatk/documentation/tooldocs/current/org_broadinstitute_gatk_tools_walkers_haplotypecaller_HaplotypeCaller.php/.

[13] Hamid Mushtaq and Zaid Al-Ars. Cluster-based apache spark implementation of the gatk dna analysis pipeline. In *2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 1471–1477, Nov 2015.

[14] Hamid Mushtaq, Frank Liu, Carlos Costa, Gang Liu, Peter Hofstee, and Zaid Al-Ars. Sparkga: A spark framework for cost effective, fast and accurate dna analysis at scale. In *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology,and Health Informatics*, ACM-BCB '17, pages 148–157, New York, NY, USA, 2017. ACM.

[15] Johan Peltenburg, Shanshan Ren, and Zaid Al-Ars. Maximizing systolic array efficiency to accelerate the pairhmm forward algorithm. In *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 758–762, Dec 2016.

[16] Shanshan Ren, Koen Bertels, and Zaid Al-Ars. Exploration of alternative gpu implementations of the pair-hmms forward algorithm. In *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 902–909, Dec 2016.

[17] Petr Danecek, Adam Auton, Goncalo Abecasis, Cornelis A. Albers, Eric Banks, Mark A. DePristo, Robert E. Handsaker, Gerton Lunter, Gabor T. Marth, Stephen T. Sherry, Gilean McVean, Richard Durbin, and . The variant call format and vcftools. *Bioinformatics*, 27(15):2156, 2011.

[18] Yonghong Yan, Max Grossman, and Vivek Sarkar. Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 887–899, Berlin, Heidelberg, 2009. Springer-Verlag.

5

GPU ACCELERATED SEQUENCE ALIGNMENT WITH TRACEBACK

SUMMARY

This chapter proposes to accelerate the semi-global pairwise sequence alignment with traceback on GPUs to improve the performance of GATK HaplotypeCaller (HC). Based on the characteristics of the semi-global alignment with traceback in GATK HC, the intra-task parallelization model is chosen. Moreover, our GPU implementation also records the length of consecutive matches/mismatches in addition to lengths of consecutive insertions and deletions as in the CPU implementation. Experimental results show that our alignment kernel with traceback is up to 14.14x faster than its CPU counterpart. When integrated into GATK HC (alongside a GPU accelerated pair-HMMs forward kernel), the GPU-based GATK HC implementation is 2.3x faster than the baseline GATK HC implementation, and 1.34x faster than the GATK HC implementation only with the integrated GPU-based pair-HMMs forward algorithm.

This chapter is based on the following paper.

S. Ren, N. Ahmed, K.L.M. Bertels, Z. Al-Ars, *GPU Accelerated Sequence Alignment with Trace-back for GATK HaplotypeCaller*. Accepted for publication in BMC Genomics, 2019 [Journal]

RESEARCH

GPU Accelerated Sequence Alignment with Traceback for GATK HaplotypeCaller

Shanshan Ren, Nauman Ahmed, Koen Bertels and Zaid Al-Ars*

*Correspondence:
z.al-ars@tudelft.nl
Delft University of Technology,
Mekelweg 4, 2628 CD Delft, The
Netherlands

Abstract

Background: Pairwise sequence alignment is widely used in many biological tools and applications. Existing GPU accelerated implementations mainly focus on calculating optimal alignment score and omit identifying the optimal alignment itself. In GATK HaplotypeCaller (HC), the semi-global pairwise sequence alignment with traceback has so far been difficult to accelerate effectively on GPUs.

Results: We first analyze the characteristics of the semi-global alignment with traceback in GATK HC and then propose a new algorithm that allows for retrieving the optimal alignment efficiently on GPUs. For the first stage, we choose intra-task parallelization model to calculate the position of the optimal alignment score and the backtracking matrix. Moreover, in the first stage, our GPU implementation also records the length of consecutive matches/mismatches in addition to lengths of consecutive insertions and deletions as in the CPU-based implementation. This helps efficiently retrieve the backtracking matrix to obtain the optimal alignment in the second stage.

Conclusions: Experimental results show that our alignment kernel with traceback is up to 80x and 14.14x faster than its CPU counterpart with synthetic datasets and real datasets, respectively. When integrated into GATK HC (alongside a GPU accelerated pair-HMMs forward kernel), the overall acceleration is 2.3x faster than the baseline GATK HC implementation, and 1.34x faster than the GATK HC implementation with the integrated GPU-based pair-HMMs forward algorithm. Although the methods proposed in this paper is to improve the performance of GATK HC, they can also be used in other pairwise alignments and applications.

Keywords: Semi-global alignment with traceback; optimal alignment; GATK HaplotypeCaller (HC); GPUs

Background

NGS (Next Generation Sequencing) platforms offer the capacity to generate large amounts of DNA sequencing data in a short time and at a low cost. However, the analysis of the dramatic amounts of DNA sequencing data is still a computational challenge. Researchers have proposed many methods to improve the performance of the DNA sequencing data analysis tools and applications. One method is to execute these tools and applications on high performance computing architectures, such as supercomputers, clusters and even cloud environments. Another method is to use accelerators, such as GPUs and FPGAs, to accelerate the time-consuming kernels of these tools and applications to improve their performance.

GATK HaplotypeCaller (HC) is a popular variant caller, which is used to find the differences (or variants) between the sample DNA sequence compared with

the reference sequence. Although GATK HC has higher accuracy of identifying variants compared with many other variant callers, its feasibility is limited by the long execution time needed for the analysis, which has proven to be difficult to optimize. This has driven researchers to improve its performance. Intel and IBM researchers both employ vector instructions to optimize the pair-HMMs forward algorithm [1, 2], which is the most time-consuming part of GATK HC, to reduce the total execution time. [3, 4] uses GPUs to accelerate the pair-HMMs forward algorithm in GATK HC, which achieved 1.71x speedup in single thread mode. After accelerating the pair-HMMs forward algorithm on GPUs, profiling of GATK HC shows that the semi-global pairwise sequence alignment accounts for around 34.5% of the overall execution time, making it the most time-consuming kernel in the application. This provides an opportunity to further improve the performance of GATK HC using GPU acceleration.

Pairwise sequence alignment, which includes global alignment, semi-global alignment and local alignment, is one of the commonly used techniques in many biological tools and applications. The global alignment and the semi-global alignment are calculated by the Needleman-Wunsch algorithm and the modified Needleman-Wunsch algorithm, respectively, while the local alignment is calculated by the Smith-Waterman algorithm. Although there are some differences existing in the three algorithms, the main framework of these algorithms is similar, which includes two stages: (1) a dynamic programming kernel to calculate the score matrices and find the optimal alignment score; (2) a traceback (or backtracking) kernel to find the optimal alignment.

Since three kinds of pairwise sequence alignment (global, semi-global and local) have the same framework and differ only in details, techniques of speeding up one can be applied to the other two with tiny modifications. Different kinds of high-performance platforms, especially accelerators, such as FPGAs [5, 6] and GPUs [7–16], are used to reduce their execution time.

There has been much research done to reduce the execution time of the three kinds of pairwise alignment on GPUs. There are two approaches to implement the first stage of the pairwise sequence alignment on GPUs (which is to calculate the optimal alignment score): inter-task parallelization model and intra-task parallelization model. The former is that each thread performs one alignment independently, such as [7] and [8]. The latter is that threads in a block cooperate to perform an alignment, such as [9]. If the pairwise sequence alignment is applied for sequence database scanning, aligning a query sequence with all database sequences for sequence similarity, a query profile and related data storage and access techniques are employed to reduce memory accesses on GPUs, such as [10] and [11]. In [11], alignments are performed in interleaved mode in order to amortize the cost of initiating each execution pass.

However, very few researchers implement the second stage on GPUs. The existing implementations can be classified into two groups. The implementations of the first group are based on backtracking matrices. [11] proposed to store the score matrices and backtrack the score matrices to obtain the optimal alignment. However, the method is not described clearly. gpu-pairAlign [12] proposed to store the alignment moves in four Boolean backtracking matrices during the first stage and retrieve the

four Boolean backtracking matrices instead of the score matrices. This group of implementations obtain the optimal alignment in linear time, but the disadvantage is that their space complexity is quadratic. The implementations of the second group are based on the Myers-Miller algorithm. MSA-CUDA [13] developed a stack-based iterative implementation of the Myers-Miller algorithm [17] to retrieve the optimal alignment in linear space. SW# [14] proposed a modified Myers-Miller algorithm. CUDAlign 2.0 [15] combined the Myers-Miller and Smith-Waterman algorithm. Moreover, with several versions of incremental optimizations, CUDAlign 4.0 [16] is able to achieve the optimal alignment of chromosome-wide sequences using multiple GPUs. However, their approaches have quadratic time complexity, making them only suitable for the pairwise alignment of very long DNA and protein sequences.

In this paper, we provide an accelerated solution tailored to GATK HC which implements the semi-global pairwise sequence alignment with traceback on GPUs to further improve the performance. The contributions of this paper are as follows:

- We first analyze the characteristics of the semi-global alignment in GATK HC and then propose a GPU-based implementation of the semi-global alignment with traceback based on the analysis.
- During the first stage, we propose to record the length of consecutive match(es)/mismatch(es) and store the alignment moves in a special backtracking matrix.
- We also propose a new algorithm that allows for retrieving the optimal alignment efficiently on GPUs.
- We benchmark the results and show an overall speedup of GATK HC of about 2.3x over the non-accelerated version.

Although this paper proposes to improve the performance of GATK HC, the GPU-based implementation of the semi-global alignment with traceback can be used in other applications and tools. Moreover, since there are only small differences among the global alignment, semi-global alignment and local alignment, the methods proposed in this paper can also be applied to the global alignment and local alignment.

Methods

A brief overview of semi-global alignment

R1:	A G T C T A G - - C G -
R2:	- G A C - - G A A C G T
Alignment:	M M M D D M I I M M

Figure 1 An example of an semi-global alignment of two sequences in GATK HC. R1 and R2 represent two sequences. Gaps ('-') at the start and end of two sequences are neglected. In the alignment, there are three kinds of operations indicating how R2 aligns with R1. 'M' indicates that a base in R2 aligns with a base in R1 (matches or mismatches); 'I' indicates that a base in R2 is not in R1; 'D' indicates that a base in R1 is not in R2.

Semi-global alignment finds the overlap between two sequences. Insertion and deletions introduce gaps in the alignment. Gaps at the start or end of the sequences may be neglected. Hence, different types of semi-global alignments are possible between two sequences. Figure 1 shows an example of the type of the semi-global alignment performed in GATK HC.

The pairwise sequence alignment is to find the optimal alignment between two sequences, which has the optimal alignment score. The modified Needleman-Wunsch algorithm with affine gap penalties to calculate the optimal alignment score of the semi-global alignment in GATK HC is defined as
Initialization:

$$\begin{aligned} M_{i,0} &= 0 \quad (0 \le i \le m) \\ M_{0,j} &= 0 \quad (0 \le j \le n) \end{aligned} \tag{1}$$

Recurrence:

$$\begin{aligned} M_{i,j} &= max \begin{cases} M_{i-1,j-1} + sbt(R1[i], R2[j]) \\ D_{i,j} \\ I_{i,j} \end{cases} \\ D_{i,j} &= max \begin{cases} D_{i-1,j} - \beta \\ M_{i-1,j} - \alpha \end{cases} \\ I_{i,j} &= max \begin{cases} I_{i,j-1} - \beta \\ M_{i,j-1} - \alpha \end{cases} \end{aligned} \tag{2}$$

Termination:

$$Result = max \begin{cases} \max_{\{1 \le i \le m\}} M_{i,n} \\ \max_{\{1 \le j \le n\}} M_{m,j} \end{cases} \tag{3}$$

where m and n are the length of R1 and R2, respectively. In these equations, $M_{i,j}$ represents the optimal alignment score of two subsequences $R1[1]...R1[i]$ and $R2[1]...R2[j]$, while $I_{i,j}$ and $D_{i,j}$ represent the optimal alignment score of two subsequences $R1[1]...R1[i]$ and $R2[1]...R2[j]$ with $R2[j]$ aligned to a gap and $R1[i]$ aligned to a gap, respectively. Here, the semi-global alignment uses an affine gap penalty model to calculate gap penalties, in which α and β are the gap open penalty and the gap extension penalty, respectively. *sbt* is the score of a match or mismatch. As shown by Equation 1, the penalties of gaps at the start and end of two sequences are neglected. As shown by Equation 3, the optimal alignment score of the semi-global alignment in GATK HC is the greatest value of the elements in the last row and the last column of the matrix M.

These equations indicate that the computation complexity of the modified Needleman-Wunsch algorithm is $O(mn)$, which makes the execution time increase quadratically with the sequence length. Usually, the algorithm is implemented using dynamic programming which solves three two dimensional matrices. According to the equations, $M_{i,j}$, $I_{i,j}$ and $D_{i,j}$ only depend on the up-left, up and left neighbor elements, which implies that the elements on the same anti-diagonal can be computed in parallel. Thus, a method employed by many researchers to reduce the execution time is to exploit this inherent parallelism in the algorithm.

If the alignment only needs to find the optimal alignment score of two sequences, the dynamic programming kernel can be calculated in linear space. Otherwise, the

alignment with affine gap penalties generally uses three backtracking matrices to store the scores or alignment moves calculated by the dynamic programming kernel. The optimal alignment traceback starts from the position of the element with the optimal alignment score until reaching any element in the first row or the first column of the backtracking matrices, which is calculated in linear time. Figure 2 presents an example of backtracking an optimal alignment based on the score matrices.

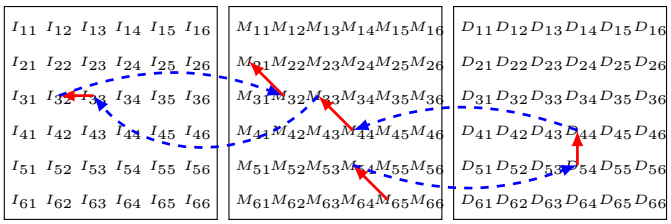


Figure 2 An example of backtracking an optimal alignment. The backtracking of an optimal alignment starts from $M_{6,5}$ (central matrix in the figure), passes through $M_{5,4}$, jumps to $D_{5,4}$ (right matrix in the figure), passes through $D_{4,4}$, jumps to $M_{4,4}$, passes through $M_{3,3}$, jumps to $I_{3,3}$ (left matrix in the figure), passes through $I_{3,2}$, jumps to $M_{3,2}$ and ends at $M_{2,1}$. The optimal alignment retrieved is "MDMIM".

Cigar format

In GATK HC, the goal is to get the optimal semi-global alignments, which are represented in the CIGAR format [18], and POS. CIGAR is a string including one or more number-character pair(s). The character, including 'M', 'I', 'D', 'N', 'S', 'H', 'P', '=' and 'X', defines an operation explaining how the base in R2 aligns to the base in R1. Table 1 shows the CIGAR operations used in GATK HC. The number defines the length of the consecutive operations. POS is 0-based left most position of the first matching base of R1, which indicates the position of R1 where the alignment starts.

Table 1 CIGAR operations used in GATK HC

Operation	Description
M	Match/mismatch
I	Insertion (gap in R1)
D	Deletion (gap in R2)
S	Soft clipping (base at the beginning or the end of R2 but not in R1)

Take the alignment in Figure 1 for example. The CIGAR representation of the alignment is "3M2D1M2I2M1S" and POS of the alignment is 1.

GPU architecture

Modern GPUs are widely used to accelerate computationally intensive algorithms. A GPU consists of thousands of small cores capable of executing one thread at a time. On NVIDIA GPUs, threads are grouped into *blocks* and these blocks are grouped into *grids*. Furthermore, consecutive threads in the same block are grouped into

warps. The size of a warp is usually 32. The memory hierarchy includes registers, shared memory, global memory, cache and so on. Each thread is assigned a set of registers. The shared memory is accessed by all threads in a block. Using the shared memory, the threads in a block can exchange data at a very fast rate. The global memory is accessed by all the threads on the GPU. The latency of the global memory access is high since it resides on the device DRAM. If the data accessed by each thread in the same warp are stored at consecutive addresses, the global memory accesses of these threads can be coalesced. Usually, the width of one global memory access is 128 bytes. If the global memory accesses of threads in a warp are coalesced, there will be only one global memory access when the data accessed by each thread is not more than 4 bytes. Otherwise, there would be 32 sequential global memory accesses in the worst-case situation.

Semi-global alignment in GATK HC

Implementation of alignment in GATK HC

In GATK HC, the semi-global pairwise alignment is performed in two stages.

The implementation of the first stage is realized with a two-layer loop, which results in the $O(mn)$ computational complexity. The results of the first stage are two matrices: the score matrix sw, which stores matrix M, and the backtracking matrix $btrack$. In $btrack$, the value of each element can be classified into three kinds, which is defined as follows:

- > 0 - indicates a deletion and the length of the consecutive deletion(s) is the value of the element
- $= 0$ - indicates a match or mismatch and the length of the consecutive match(es)/mismatch(es) is increased by 1
- < 0 - indicates an insertion and the length of the consecutive insertion(s) is the absolute value of the element

The absolute values of the elements in the backtracking matrix are calculated by recording the length of the consecutive deletion(s) and consecutive insertion(s) when calculating the score matrix.

The implementation of the second stage is to calculate the optimal alignment in CIGAR format and POS. The score matrix sw is first used to find the optimal alignment score and the backtracking matrix *btrack* is then used to obtain the optimal alignment and POS. The optimal alignment is calculated in linear time. The backtracking matrix in GATK HC is helpful during backtracking. It is much easier to identify the next move compared with other methods since it does not need to jump among several backtracking matrices (shown in Figure 1) or calculate the next move based on the current move [12]. Moreover, the lengths of the consecutive deletion(s) and consecutive insertion(s) are given by the element of the *btrack* matrix. However, the length of the consecutive match(es)/mismatch(es) is not given, which is increased by one instead.

Data analysis

In GATK HC, the semi-global alignment is performed in three situations:

1 Align the reference path with the dangling path to recover dangling branches for the local assembly.

2 Align the read with the assembled haplotype.
3 Align the assembled haplotype with the reference to decide whether the assembled haplotype satisfied the defined requirements.

We profiled GPU-based GATK HC [3] with a typical workload (Chromosome 10 of the whole human genome dataset G15512.HCC1954.1 [19]) to investigate which situation is most time-consuming. The profiling results in single-threaded mode are shown in Table 2, which specifies the relative execution time and the number of the semi-global alignments in each situation. As shown in the table, the execution time of all the semi-global alignment accounts for 34.5% of the total execution time. Moreover, situation 2 and 3 consumes around 100% of the semi-global alignment execution time and the execution time in situation 1 is negligible. However, although the number of semi-global alignments in situation 2 is much larger than that in situation 3, the execution time in situation 2 is smaller than that in situation 3.

Table 2 Execution time of the semi-global alignment in three situations of GATK HC

Situation	Number of alignments	Execution time
1	3529	0.03%
2	850376	14.58%
3	54802	19.89%
Total	908707	34.5%

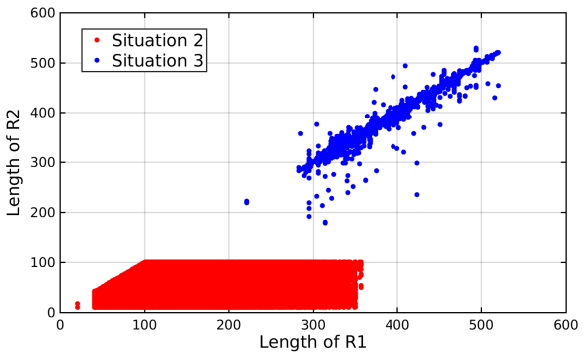


Figure 3 Lengths of sequence pairs in situation 2 and 3. The lengths of the sequence pairs in situation 2 (40∼350) are shorter than those in situation 3 (300∼520).

We then analyzed the lengths of the sequence pairs in situation 2 and 3. In situation 2, let R1 be the assembled haplotype and R2 be the read. In situation 3, let R1 be the assembled haplotype and R2 be the reference. Figure 3 shows a scatter plot of the lengths of the sequence pairs in these two situations. As shown in Figure 3, the lengths of the sequence pairs in situation 2 (40∼350) are shorter than those in situation 3 (300∼520). Since the computation complexity is $O(mn)$, the execution time of each semi-global alignment in situation 3 is bigger than that in situation 2. This explains why the total execution time of situation 3 is bigger than that of situation 2, which is shown in Table 2. Moreover, in situation 2, the length of R2 (the read) is shorter than the length of R1 (the assembled haplotype).

In addition, we investigated the optimal alignments achieved in situation 2 and 3 and added up the number of M/I/D/S operations in each optimal alignment. Figure 4 shows that the number of M operations is the largest. Especially in situation 2, the number of M operations accounts for 99.65% of the total operations. Moreover, we found that most of M operations are consecutive in each optimal alignment. However, the length of the consecutive match(es)/mismatch(es) is increased by one during the optimal alignment retrieval.

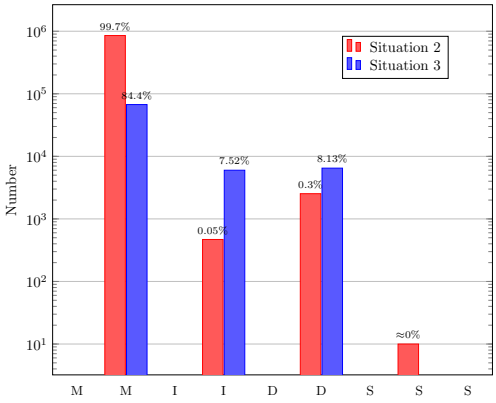


Figure 4 Numbers of M/I/D/S in situation 2 and 3. The number of M operations is the largest. Especially in situation 2, the number of M operations accounts for 99.65% of the total operations.

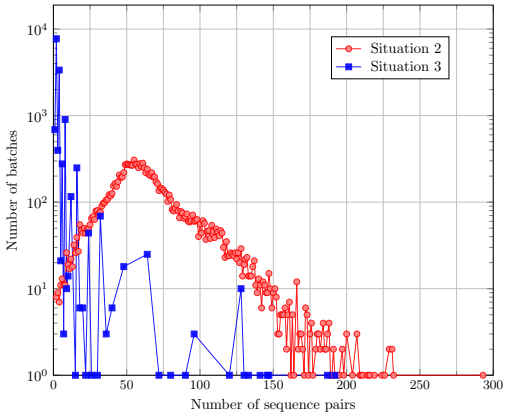


Figure 5 Numbers of batches including different number of sequence pairs in situation 2 and 3. The biggest number of sequence pairs in all the batches in situation 2 and 3 are 293 and 192, respectively. Furthermore, the majority of batches in situation 2 include 25 $\sim$ 125 of sequence pairs while the majority of batches in situation 3 include 1 $\sim$ 8 of sequence pairs.

Hence, although the computation complexity of the optimal alignment is linear, most of its execution time is used to calculate the length of the consecutive match(es)/mismatch(es).

We last studied the source code of GATK HC version 3.7 and found that the semi-global alignments in situation 2 and 3 can be grouped into many batches without big modifications of the source code. Each batch consists of many semi-global alignments of sequence pairs. The numbers of batches in situation 2 and 3 are 13142 and 13977, respectively. Figure 5 shows the number of sequence pairs of each batch in situation 2 and 3. The biggest number of sequence pairs in all the batches in situation 2 and 3 are 293 and 192, respectively. Furthermore, the majority of batches in situation 2 include $25 \sim 125$ of sequence pairs while the majority of batches in situation 3 include $1 \sim 8$ of sequence pairs.

Implementation on GPUs

The implementation of the semi-global pairwise alignment for GATK HC on GPUs is performed in two stages. In the first stage, it performs the modified Needleman-Wunsch algorithm in order to obtain the backtracking matrix and the position of the optimal alignment score. In the second stage, it retrieves the backtracking matrix in order to obtain the optimal alignment in CIGAR format and POS.

First stage implementation

Intra-task parallelization As mentioned in Subsection "Data analysis", the number of sequence pairs in each batch is less than 300. In order to effectively use the resources on GPUs, the intra-task parallelization model is employed to implement the modified Needleman-Wunsch algorithm on GPUs. For the implementation on GPUs, the elements on the same anti-diagonal of the score matrix M, I and D and backtracking matrix are calculated in parallel, reducing the computational complexity to $O(m+n)$. Figure 6 shows the calculation of matrix M as an example to explain the implementation. Let R1 and R2 be the two sequences. There are in total 6 threads in the block and the size of matrix M is 6×6. At each step, the elements on an anti-diagonal are calculated in parallel and every element is calculated by one thread. For example, at step 5 (S5), $M_{5,1}$, $M_{4,2}$, $M_{3,3}$, $M_{2,4}$ and $M_{1,5}$, which are on the same anti-diagonal, are calculated by thread 0 (T0), thread 1 (T1), thread 2 (T2), thread 3 (T3) and thread 4 (T4), respectively. These elements are then used in the next step to calculate the elements on the next anti-diagonal. Moreover, each thread is responsible to calculate the elements in one column of matrix M. For example, the elements in the second column are calculated by thread 1 (T1). The goal of the first stage is to obtain the backtracking matrix and the position of the optimal alignment score. Therefore, elements of the score matrix M, I and D are not stored. Instead, two vectors in the shared memory and three registers of each thread are used to store the intermediate results of the three score matrices. During the calculation of elements of the last column and the last row of matrix M, the optimal alignment score and its position are obtained. However, the drawback of the implementation is that some threads remain idle at the beginning or at the end of the calculation procedure, resulting in low thread utilization. If the length of R2 is smaller than the number of threads in a block, the execution is similar to Figure

6 while some threads would remain idle during the whole calculation procedure. If the length of R2 is bigger than the number of threads in a block, there are two solutions to deal with it. One is to increase the size of a block until the number of threads in a block is equal to or bigger than the length of R2. The other is to divide the calculation into several passes. In each pass, the execution is similar to Figure 6. Three vectors in the global memory are used to store the intermediate results produced by the last thread of each pass, which would be used in the next pass. The advantage of the second solution is that it increases efficiency by reducing idle percentage of threads during the calculation procedure while its disadvantage is that it increases global memory accesses.

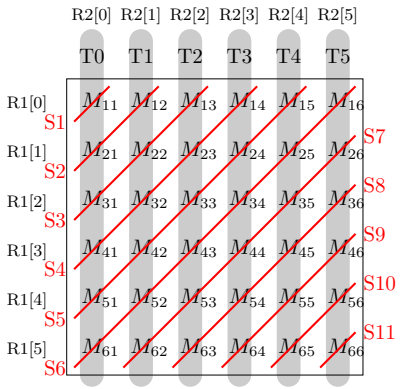


Figure 6 Calculation of matrix M on GPUs.

Recording the length of consecutive match(es)/mismatch(es) Besides recording the length of the consecutive deletion(es)/insertion(es), we also record the length of the consecutive match(es)/mismatch(es) in the first stage. The backtracking matrix on GPUs is stored in a *short*2 matrix. Each element of the matrix has two values, which are x and y. The value of x and y are defined as follows:

- $x > 0$ - indicates a deletion and the length of the consecutive deletion(s) is the value of the element
- $x = 0$ - indicates a match or mismatch and the length of the consecutive match(es)/mismatch(es) is y.
- $x < 0$ - indicates an insertion and the length of the consecutive insertion(s) is the absolute value of the element

The data type of x and y is *short*, of which the minimum value and maximum value are -32768 and 32767, respectively. The absolute values of the minimum value and maximum value are bigger than the theoretical maximum length of the consecutive operations, which is the length of R1 or R2. In order to calculate the backtracking matrix, a *short*2 vector in the shared memory and two registers of each thread are used.

Moreover, Since the backtracking matrix will be used in the next stage and the shared memory is not big enough to store it, the backtracking matrix is stored

in the global memory. Similar to calculation of the matrix M shown in Figure 6, elements of the backtracking matrix are calculated in anti-diagonal order. Thus, the backtracking matrices are stored in the diagonal-major data format (Figure 7 (b)), which is proposed in [20], instead of the row-major data format (Figure 7 (a)) to avoid non-coalesced global memory accesses of 32 threads in a warp and reduce global memory accesses.

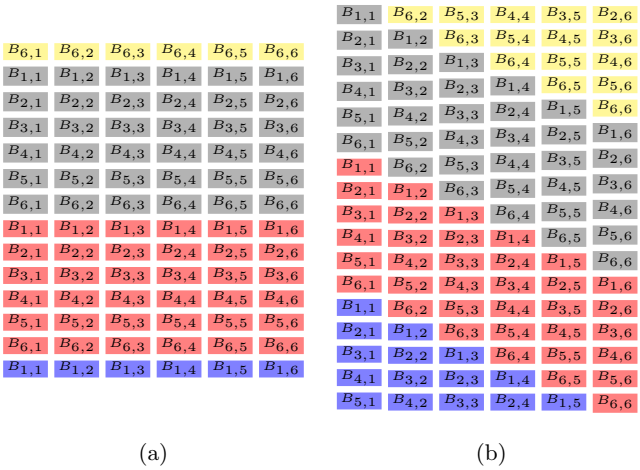


Figure 7 Layout of the backtracking matrix in the global memory. Elements of different backtracking matrices are marked with different colors.

Second stage implementation

In the second stage, we use the backtracking matrix *btrack* to obtain the optimal alignment and POS. Algorithm 1 presents the pseudo code of the optimal alignment retrieval on GPUs. $P1$ and $P2$ describe the position of the optimal alignment score. Algorithm 1 first checks whether there are soft clippings at the end of R2, and then computes the optimal alignment in a *while* loop. At the end, it checks whether there are soft clippings at the beginning of R2. The backtracking starts from $(P1, P2)$ and finishes when $i \leqslant 0$ or $j \leqslant 0$, which is calculated in linear time. POS is the value of $(i-1)$ at the end of the *while* loop. In addition, the position of each element in the backtracking matrix is calculated by i, j and max_col, as shown in the 9th line in Algorithm 1. max_col is the column size of the maximum backtracking matrix of all sequence pairs.

The length of the deletion, insertion and match/mismatch is given by the value of an element of the backtracking matrix, as shown in the 13th, 18th and 23rd line, respectively. This reduces the global memory accesses used to calculate the length of the operations.

Algorithm 1 Pseudo code of optimal alignment retrieval

```
1: function CalculateCigar(btrack[ ][ ], Cigar[ ], m, n, POS, P1, P2, max_col)
2:     state ←′ N′                                              ▷ Initialization of state
3:     length ← 0
4:     if P2 > 0&&n − P2 > 0 then                    ▷ Check soft clippings at the end of R2
5:         Cigar[index].num = n − P2
6:         Cigar[index + +].c =′ S′
7:     end if
8:     i = P1, j = P2
9:     while i > 0&&j > 0 do                                  ▷ Compute optimal alignment
10:        tp = B[(i − 1 + j − 1) ∗ max_col + j − 1]
11:        if tp.x > 0 then
12:            new_state =′ D′
13:            i = i − tp.x
14:            step_length = tp.x
15:        else
16:            if tp.x < 0 then
17:                new_state =′ I′
18:                j = j − abs(tp.x)
19:                step_length = abs(tp.x)
20:            else
21:                new_state =′ M′
22:                i = i − tp.y
23:                j = j − tp.y
24:                step_length = tp.y
25:            end if
26:        end if
27:        if state ==′ N′ then
28:            state = new_state
29:        end if
30:        if new_state == state then
31:            length+ = step_length
32:        else
33:            Cigar[index].num = length
34:            Cigar[index + +].c = state
35:            length = step_length
36:            state = new_state
37:        end if
38:    end while
39:    Cigar[index].num = length
40:    Cigar[index + +].c = state
41:    POS = i − 1
42:    if j > 0 then                                 ▷ Check soft clippings at the beginning of R2
43:        Cigar[index].num = j
44:        Cigar[index + +].c =′ S′
45:    end if
46: end function
```

Results

All the experiments are performed on IBM Power System S823L (82478-42L), which has 2 IBM Power8 processors (10 cores each) running at 3.6 GHz, 256 GB of DDR3 memory, and an NVIDIA Tesla K40 card. The NVIDIA Tesla K40 card has 2880 cores that run at up to 745 MHz and has a CUDA compute capability of 3.5.

We first compare the performance of the GPU-based semi-global alignment implementation with different techniques using the synthetic datasets. The synthetic datasets are created based on the output of Wgsim [21] with default parameters. We then compare the performance of GPU-based semi-global alignment implementation with gpu-pairAlign implementation using synthetic datasets. Next, we compare the performance of GPU-based semi-global alignment implementation with CPU-based implementation using synthetic and real datasets. We last integrate the GPU-based semi-global alignment implementation into GATK HC 3.7 and compare the overall performance.

Throughput is used as a performance metric of the first stage of the GPU-based implementation, which is measured by giga cell updates per second (GCUPS). Note that it is not fair to compare the throughput of the first stage of the semi-global alignment with traceback with that of the score-only alignments since the former needs to store backtracking matrices in the global memory.

Performance comparison of multi-pass

There are two solutions to implement the first stage of the semi-global alignment on GPUs if the length of R2 is bigger than the number of threads in a block. We realized these two solutions and used different synthetic datasets to compare the performance of the two solutions.

Figure 8 shows the performance of the two solutions with different synthetic datasets. There are 9 datasets each with a different length of R1/R2, namely: 64, 128, 192, 256, 320, 384, 448, 512 and 576. In each dataset, the lengths of R1 and R2 are the same. The number of sequence pairs in the 9 datasets is 25, 100 and 1000.

For the first solution, which is to increase the block size, there are in total 9 implementations for 9 datasets. The differences of these implementations are the block size and the sizes of vectors in the shared memory which store the intermediate results. For the second solution, which employs multi-pass, there is 1 implementation with block size of 128.

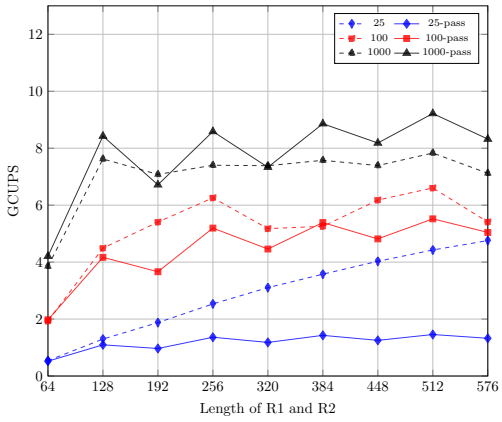


Figure 8 Performance comparison of implementations for two solutions on synthetic datasets.

As shown by Figure 8, the throughput of the first solution is higher than that of the second solution when the number of sequence pairs of the datasets is 25 and 100. However, when the number of sequence pairs of the datasets is 1000, the throughput of the second solution is higher in most cases. This is because the efficiency of the implementations for the first solution is smaller than that of the implementation for the second solution and the advantage of the second solution overweighs its disadvantage when the number of sequence pairs of the dataset is big. Thus, we can choose the implementation of these two solutions based on the number of sequence pairs of the dataset.

Performance comparison of recording match/mismatch lengths

In this section, we analyze the impact of recording the length of consecutive matches/mismatches on the performance of the second stage of the alignment on GPUs. We realized two implementations. The first implementation is our approach shown in Algorithm 1 in which the length of consecutive matches/mismatches is recorded in the backtrack matrix. The second implementation is similar to Algorithm 1 except that the length of M is increased by one and the coordinates (i, j) of M are decreased by 1. The backtracking matrices are produced by 9 implementations for the first solution using 9 synthetic datasets. Here, the synthetic datasets are not based on the output of Wgsim since we consider the best case, in which only many M operations exist in the optimal alignment. The lengths of R1/R2 in the 9 synthetic datasets are 64, 128, 192, 256, 320, 384, 448, 512 and 576. The number of sequence pairs in the 9 datasets is 100.

Figure 9 shows the execution time of the two implementations. The implementation which records match/mismatch lengths is faster. Moreover, its execution time remains nearly constant with increasing length of R1 and R2 as it only requires a single global memory access per R1 and R2 pair. The execution time of the implementation without recording match/mismatch lengths increases linearly with the length of R1 and R2. This is because the number of global memory accesses increases linearly with the number of M operations in the optimal alignment, which in turn increases linearly with the length of R1 and R2.

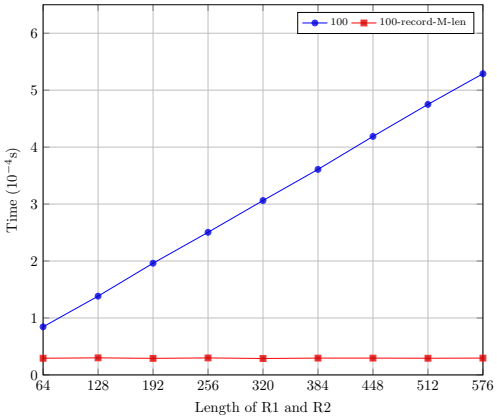


Figure 9 Execution time of GPU-based optimal alignment backtracking implementations (not) recording match/mismatch lengths on synthetic datasets.

Performance comparison with gpu-pairAlign

As mentioned in Section "Background", there are two methods to implement the second stage on GPUs: the method based on the Myers-Miller algorithm and the method based on backtracking matrices. The method based on the Myers-Miller algorithm is only suitable for the pairwise alignment of very long DNA and protein sequences. Thus, we compared our implementation with gpu-pairAlign [12],

which uses backtracking matrices to obtain the optimal alignments. gpu-pairAlign is designed to perform alignment of every given sequence pair on GPUs, especially for protein sequence pairs. It includes algorithms for global alignment, semi-global alignment and local alignment. We compare with its semi-global alignment algorithm. The semi-global alignment algorithm of gpu-pairAlign is also performed in two stages: the optimal alignment score and the backtracking matrices are computed in the first stage; the backtracking is performed in the second stage.

There are two main differences between the gpu-pairAlign implementation and our implementation: (1) In the first stage, our implementation employs the intra-task parallelization model, while the gpu-pairAlign implementation employs the inter-task parallelization model; (2) The backtracking matrix of our implementation is a short2 matrix, elements of which are the length of consecutive deletion(es), insertion(es) and match(es)/mismatch(es), while the backtracking matrices of the gpu-pairAlign implementation are four Boolean matrices, elements of which indicate the proper direction of backtracking moves.

We modified the gpu-pairAlign implementation to make it to deal with the data produced by GATK HC: (1) Since the input data of our implementation is a set of sequence pairs instead of a set of sequences, the way in which the gpu-pairAlign implementation handles input data is modified; (2) Integer arrays are used to store the intermediate results instead of short arrays since the intermediate results are bigger than the maximum value of the short data type; (3) The alignments are modified to be represented using the CIGAR format and POS.

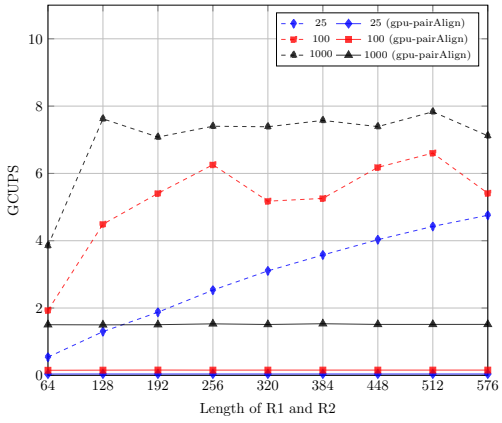


Figure 10 Performance comparison of the first stage of two implementations on synthetic datasets.

We first used the synthetic datasets described in Subsection "Performance comparison of multi-pass" to compare the performance of the first stage of the two implementations, which is shown in Figure 10. The performance of gpu-pairAlign implementation is much smaller than our implementation. The main reason is that when the size of the synthetic datasets is small, the resource on the GPU cannot be

fully utilized for the inter-task parallelization model. The second reason is that the intermediate results are stored in integer arrays, which increases the size of shared memory of each block and the number of global memory accesses.

We then compared the performance of the second stage of the two implementations using the synthetic datasets described in Subsection "Performance comparison of recording match/mismatch lengths", which is shown in Figure 11. The execution time of the second stage of our implementation remains nearly constant when the length of R1 and R2 increases, while the execution time of the second stage of the gpu-pairAlign implementation increases linearly with the length of R1 and R2. Although the gpu-pairAlign implementation reduces the global memory space by using four Boolean matrices, it still needs to calculate each move one by one, which is avoided in our implementation through storing the length of consecutive deletion(es), insertion(es) and match(es)/mismatch(es).

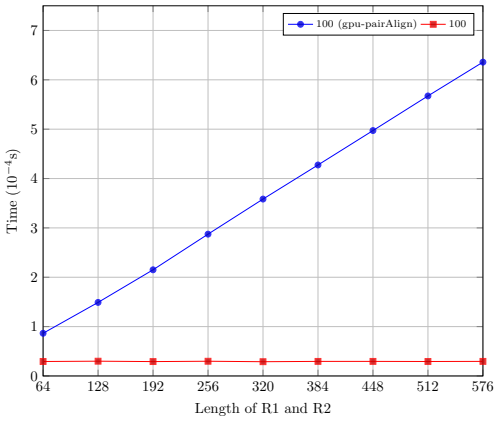


Figure 11 Execution time of the second stage of two implementations on synthetic datasets.

Performance comparison with CPU-based implementation

In this section, we compare the performance of our GPU-based semi-global alignment with traceback implementation with the CPU-based implementation using synthetic and real datasets. We used the first solution which increases the block size when the length of R2 is bigger than the block size and records the length of consecutive matches/mismatches. The CPU-based implementation is written in the C++ programming language and compiled with gcc O3 optimization, running on one Power8 core. The real datasets are produced by using a typical workload (Chromosome 10 of the whole human genome dataset G15512.HCC1954.1).

Figure 12 shows the speedup of the GPU-based implementations compared with the CPU-based implementation using the synthetic datasets described in Subsection "Performance comparison of multi-pass". There are in total 9 GPU-based implementations for 9 datasets, block size of which are 64, 128, 192, 256, 320, 384, 448,

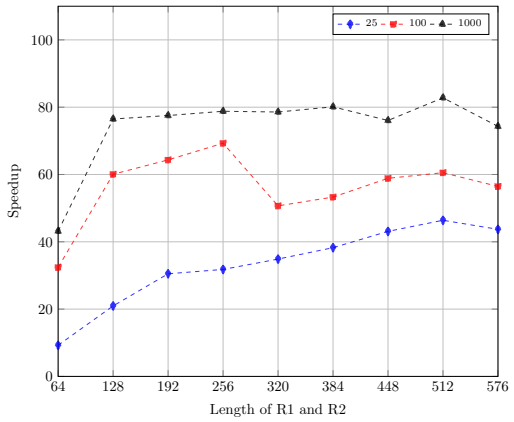


Figure 12 Speedup of the GPU-based implementations compared with CPU-based implementation on synthetic datasets.

512 and 576. The GPU-based implementations is up to 80x faster than the CPU-based implementation. Moreover, the speedup of the datasets with 1000 sequence pairs is bigger than the speedup of the datasets with 25 and 100 sequence pairs.

Table 3 shows the execution time of GPU-based implementations with the real datasets. As shown by Figure 3, the length of R2 in situation 2 is 40∼120 and the length of R2 in situation 3 is 300∼520. Thus, we used two GPU-based implementations with block size of 128 and 576 to execute the real datasets produced in situation 2 and 3, respectively. The GPU-based implementation of situation 2 is 14.14x faster than the CPU-based implementation, while the GPU-based implementation of situation 3 is 4.89x faster than the CPU-based implementation. The throughput of the first stage of the GPU-based implementation for situation 2 is 1.86 GCUPS, while that for situation 3 is 0.64 GCUPS. The throughput of situation 3 is much smaller than the throughput for the synthetic datasets with size 25. This is because the number of sequence pairs of batches in situation 3 is extremely small ($1 \sim 8$ in most cases).

Table 3 Performance of GPU-based implementations on real datasets. S2 and S3 stand for situation 2 and 3, respectively.

	Throughput (GCUPS)	GPU (sec)	CPU (sec)	Speedup
Stage 1 of S2	1.86	2.32	43.93	18.94x
Stage 2 of S2	-	0.14	0.62	4.43x
Overall of S2	-	3.15	44.55	14.14x
Stage 1 of S3	0.64	10.20	53.29	5.22x
Stage 2 of S3	-	0.09	0.17	1.89x
Overall of S3	-	10.93	53.46	4.89x

Integration into GATK HC

The two GPU-based implementations with block size of 128 and 576 are integrated into GATK 3.7 to accelerate the semi-global alignment with traceback of situation 2 and situation 3, respectively. The GATK HC implementation with both GPU-based pair-HMMs forward algorithm and GPU-based semi-global alignment with traceback is compared with other two GATK HC implementations: GATK HC (referred to as baseline), which is downloaded from the GATK website, and GATK HC with only GPU-based pair-HMMs forward algorithm. The dataset is Chromosome 10 of the whole human genome dataset (G15512.HCC1954.1). All the GATK HC implementations are performed in single thread mode.

Table 4 shows the overall execution time of these three implementations. The implementation with both GPU-based pair-HMMs forward algorithm and GPU-based semi-global alignment with traceback is 2.30x faster than the baseline implementation. Moreover, it is 1.34x faster than the implementation with only GPU-based pair-HMMs forward algorithm.

Table 4 Execution time of GATK HC implementations

	Total time (s)	Speedup
Baseline	8034.05	-
GPU (only pair-HMMs)	4687.08	1.71x
GPU (pair-HMMs + semi-global alignment with traceback)	3490.70	2.30x

Note that the number of sequence pairs of each batch produced by GATK HC is small, leading to under utilization of the GPU resources. It is better to launch multiple GATK HC processes at the same time to fully utilize the GPU resources.

Conclusion

This paper presents an implementation of the semi-global alignment with traceback on GPUs to improve the performance of GATK HC. Semi-global alignment with traceback has two stages: in the first stage, a backtracking matrix is computed; in the second stage, the optimal alignment is calculated using the backtracking matrix. Based on the characteristics of the semi-global alignment with traceback in GATK HC, the intra-task parallelization model is chosen. The first stage of our GPU implementation is up to 18.94x faster than CPU. Moreover, our GPU implementation also records the length of consecutive matches/mismatches in addition to lengths of consecutive insertions and deletions as in the CPU implementation. This helps to reduce global memory accesses and provides a speedup of up to 4.43x in the second stage. Experimental results show that our alignment kernel with traceback is up to 80x and 14.14x faster than its CPU counterpart with synthetic datasets and real datasets, respectively. The GATK HC implementation with both GPU-based pair-HMMs forward algorithm and GPU-based semi-global alignment with traceback is 2.30x faster than the baseline GATK HC. It is 1.34x faster than the GATK HC implementation with only GPU-based pair-HMMs forward algorithm.

Abbreviations
HC: HaplotypeCaller
NGS: Next Generation Sequencing

Declarations
Acknowledgements
The authors wish to thank the Texas Advanced Computing Center (TACC) at the University of Texas at Austin and IBM for the giving access to the IBM Power8 machines used in this paper.

Ethics approval and consent to participate
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Author's contributions
SR designed and performed the experiments, analyzed the data, and wrote the manuscript. All the authors jointly developed the structure and arguments for the paper, made critical revisions and approved final version.

Funding
This work was supported by CSC (Chinese Scholarship Council) grant and Delft University of Technology.
Publication of this article was sponsored by Delft University of Technology.

Availability of data and materials
The algorithm generated in this manuscript as well as all input datasets are publicly available on a publicly available repository: https://github.com/ShanshanRen/semi-global-alignment-with-traceback

Consent for publication
Not Applicable.

References
1. VdAuwera, G.: Speed up HaplotypeCaller on IBM POWER8 systems. `https://software.broadinstitute.org/gatk/blog?id=4833`
2. Proffitt, A.: Broad, Intel Announce Speed Improvements to GATK Powered by Intel Optimizations. `http://www.bio-itworld.com/2014/3/20/broad-intel-announce-speed-improvements-gatk-powered-by-intel-optimizations.html`
3. Ren, S., Bertels, K., Al-Ars, Z.: Gpu-accelerated gatk haplotypecaller with load-balanced multi-process optimization. In: IEEE International Conference on Bioinformatics and Bioengineering, pp. 497–502 (2017)
4. Ren, S., Bertels, K., Al-Ars, Z.: Efficient Acceleration of the Pair-HMMs Forward Algorithm for GATK HaplotypeCaller on Graphics Processing Units. Evol. Bioinform. Online **14**, 1176934318760543 (2018)
5. Li, I.T., Shum, W., Truong, A.K.: 160-fold acceleration of the smith-waterman algorithm using a field programmable gate array (fpga). Bmc Bioinformatics **8**(1), 1–7 (2007)
6. Benkrid, K., Liu, Y., Benkrid, A.S.: A highly parameterized and efficient fpga-based skeleton for pairwise biological sequence alignment. IEEE Transactions on Very Large Scale Integration Systems **17**(4), 561–570 (2009)
7. Hasan, L., Kentie, M., Al-Ars, Z.: Dopa: Gpu-based protein alignment using database and memory access optimizations. Bmc Res Notes **4**(1), 261 (2011)
8. Ahmed, N., Mushtaq, H., Bertels, K., Al-Ars, Z.: Gpu accelerated api for alignment of genomics sequencing data. In: IEEE International Conference on Bioinformatics and Biomedicine, pp. 510–515 (2017)
9. Maskell, D.L., Liu, Y., Bertil, S.: Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units. Bmc Research Notes **2**(1), 73 (2009)
10. Liu, Y., Schmidt, B., Maskell, D.L.: Cudasw++2.0: enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions. Bmc Research Notes **3**(1), 93 (2010)
11. Liu, Y., Huang, W., Johnson, J., Vaidya, S.: Gpu accelerated smith-waterman. In: International Conference on Computational Science, pp. 188–195 (2006)
12. Blazewicz, J., Frohmberg, W., Kierzynka, M., Pesch, E., Wojciechowski, P.: Protein alignment algorithms with an efficient backtracking routine on multiple gpus. Bmc Bioinformatics **12**(1), 181 (2011)
13. Liu, Y., Schmidt, B., Maskell, D.L.: Msa-cuda: Multiple sequence alignment on graphics processing units with cuda. In: IEEE International Conference on Application-Specific Systems, Architectures and Processors, pp. 121–128 (2009)
14. Korpar, M., Sikic, M.: Sw#-gpu-enabled exact alignments on genome scale. Bioinformatics **29**(19), 2494–5 (2013)
15. de O. Sandes, E.F., de Melo, A.C.M.A.: Smith-waterman alignment of huge sequences with gpu in linear space. In: 2011 IEEE International Parallel Distributed Processing Symposium, pp. 1199–1211 (2011). doi:10.1109/IPDPS.2011.114
16. Sandes, E.F.O., Miranda, G., Martorell, X., Ayguade, E., Teodoro, G., Melo, A.C.M.A.: Cudalign 4.0: Incremental speculative traceback for exact chromosome-wide alignment in gpu clusters. IEEE Transactions on Parallel & Distributed Systems **27**(10), 2838–2850 (2016)

17. Myers, E.W., Miller, W.: Optimal alignments in linear space. Computer Applications in the Biosciences Cabios **4**(1), 11–7 (1988)
18. Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., Subgroup, .G.P.D.P.: The sequence alignment/map format and samtools. Bioinformatics **25**(16), 2078–2079 (2009). doi:10.1093/bioinformatics/btp352
19. TCGA Mutation Calling Benchmark 4 Files. `https://gdc.cancer.gov/resources-tcga-users/tcga-mutation-calling-benchmark-4-files`
20. Xiao, S., Aji, A.M., Feng, W.: On the robust mapping of dynamic programming onto a graphics processing unit. In: International Conference on Parallel and Distributed Systems, pp. 26–33 (2009)
21. Wgsim. `https://github.com/lh3/wgsim`

6

CONCLUSIONS

This chapter outlines the main contributions of this thesis and discusses the limitations and interesting directions for future research.

6.1. MAIN CONTRIBUTIONS

This thesis aims to accelerate three computational intensive algorithms of the GATK HaplotypeCaller (HC), including de Bruijn Graph (DBG) construction algorithm, the pair-HMMs forward algorithm and the semi-global pairwise sequence alignment algorithm, on GPUs and FPGAs to improve the performance of GATK HC. In Chapters 2 to 5, we showed the proposed implementations of the three algorithms and presented the speedup achieved by comparing with the benchmark implementation using synthetic datasets and real datasets. In addition, in Chapters 4 and 5, we also presented the implementations of GATK HC integrated with GPU-based algorithms and compare the performance with the baseline GATK HC implementation. Below, we discuss the conclusions that we drew from Chapters 2 to 5.

CHAPTER 2

In this chapter, we presented a novel GPU-based algorithm of DBG construction for micro-assembly in GATK HC. We compared the performance of the GPU-based implementation with the baseline implementation. The following conclusions can be drawn from this chapter.

1. We introduced the DBG construction algorithm implemented in GATK HC and used a simple example to explain the main difference between the DBG construction in GATK HC and the general DBG construction used in genome assembly.

2. The idea of the proposed algorithm is: (1) it assumes that there are no repeat k-mers in the dataset and first calculate the occurrences of (k+1)-mers in parallel on the GPU, thereby achieving high speedup; (2) the dataset is inspected for repeat k-mers, and only these repeats are re-evaluated on the CPU.

3. Experimental results showed that a speedup of up to 3x for the proposed implementation compared with the software-only implementation with various synthetic datasets, while the speedup for real human genome datasets is up to 2.66x.

CHAPTER 3

In this chapter, we first showed an FPGA-based design of the pair-HMMs forward algorithm and compared the performance with the benchmark implementation. We then analyzed the performance characteristics of this design and presented a design utilizing the computing resources on FPGAs, which was compared with the state-of-the-art implementation. The following conclusions can be drawn from this chapter.

1. We introduced the pair-HMMs forward algorithm and investigated its inherent parallelism.

2. We proposed a novel systolic array design to accelerate the pair-HMMs forward algorithm on FPGAs, which was implemented on the Convey supercomputing platform including four FPGA co-processors. Experimental results showed that this FPGA-based implementation is up to 67x faster, compared to the software-only implementation.

3. We presented a model to calculate the utilization of a systolic array on FPGAs. We analyzed architectural alternatives and implemented one such architecture. Experimental results showed the implementation achieves up to 90% of the theoretical throughput for a real dataset and is 2.5x faster than the state-of-the-art implementation on a similar contemporary platform.

CHAPTER 4

In this chapter, we presented several GPU-based implementations of the pair-HMMs forward algorithm and compared their performance with various datasets. We showed a solution to integrate the GPU-based implementation of the pair-HMMs forward algorithm into GATK HC. At last, we proposed a load-balanced multi-process optimization for the GPU-based GATK HC implementation. The following conclusions can be drawn from this chapter.

1. Based on the characteristics of GPU architecture and the inherent parallelism of the pair-HMMs forward algorithm, we presented several GPU-based implementations of this algorithm.

2. We showed the performance comparison of these implementations with various datasets and real datasets. Experimental results showed that the proposed implementations achieve a speedup of up to 5.47x over existing GPU-based implementations.

3. We integrated the GPU-based implementation of the pair-HMMs forward algorithm into GATK HC. In single-threaded mode, the GPU-based GATK HC is 1.71x faster than the baseline GATK HC implementation.

4. We proposed a load-balanced multi-process optimization that divides the genome into regions of different sizes to ensure a more equal distribution of computation load between different processes. The GPU-based implementation in load-balanced multi-process mode achieves up to 2.04x over the baseline GATK HC implementation in non-load-balanced multi-process mode.

CHAPTER 5

In this chapter, we presented a GPU-based implementation of the semi-global pairwise sequence alignment with traceback. Moreover, we integrated this GPU-based implementation into GATK HC. The following conclusions can be drawn from this chapter.

1. We introduced the semi-global alignment algorithm in GATK HC and discussed the characteristics of the input datasets and output datasets of this algorithm.

2. We presented a GPU-based implementation of semi-global alignment with traceback. Based on the characteristics of the semi-global alignment algorithm in GATK HC, the intra-task parallelization model was chosen. Further, the GPU-based implementation records the length of consecutive matches/mismatches in addition to lengths of consecutive insertions and deletions as in the CPU implementation. Experimental results showed that the GPU-based implementation is up to 14.14x faster than the software-only implementation.

3. We integrated the GPU-based implementation of the semi-global alignment with traceback into GATK HC alongside the GPU-based implementation of the pair-HMMs forward algorithm. This GPU-based implementation is 2.3x faster than the baseline GATK HC implementation. Moreover, it is 1.34x faster than the GATK HC implementation only with the integrated GPU-based pair-HMMs forward algorithm.

6.2. LIMITATIONS AND FUTURE WORK

We conclude this thesis by discussing the limitations and the possible directions for future work.

First, we have not integrated the FPGA-based implementation of the pair-HMMs forward algorithm into GATK HC. As mentioned in Section 1.2, the Java program cannot directly launch the programming code of FPGAs and GPUs. In order to solve this limitation, we need to search solutions provided by packages and interfaces. For the GPU-based implementations, we used JCuda and JNI (Java Native Interface) to solve this limitation. For FPGA-based implementations, we have tried to employ IPC (Inter-Process Communication) techniques of shared memory to allow the FPGA-based implementation of the pair-HMMs algorithm and GATK HC to communicate with each other on the Convey supercomputing platform. However, the performance was not as good as expected. In 2014, IBM released systems accelerated both by FPGAs and GPUs. Moreover, IBM proposed CAPI (Coherent Accelerator Processor Interface) technology, which makes accelerating workloads on FPGAs faster and easier than earlier approaches. Thus, it would be possible to use the CAPI technology and other techniques together to integrate the FPGA-based implementation into GATK HC on a CAPI-enabled IBM machine.

Second, we have not integrated the GPU-based DBG construction algorithm for micro-assembly into GATK HC. As mentioned in Section 1.2, the size of input datasets of the algorithms should be large enough to fully utilize the computing sources on GPUs and FPGAs. For the GPU-based implementations of pair-HMMs forward algorithm and semi-global alignment algorithm, it is not difficult to solve this limitation by modifying the source code of GATK HC. However, for the GPU-based implementation of DBG construction algorithm for micro-assembly, it requires a major modification of the source code of GATK HC. In early 2018, GATK 4.0 was released, which modified the engine framework of the non-Spark implementations of many tools, including GATK HC, making it easier to modify the source code. It is possible to integrate the GPU-based DBG construction algorithm for micro-assembly into GATK 4.0.

Third, the implementation of GATK HC with integrated GPU-based implementations of the pair-HMMs forward algorithm and the semi-global alignment algorithm only ran on one compute node. It would be interesting to run the GPU-based implementation of GATK HC on all the compute nodes in a cluster or on a cloud computing environment.

Fourth, although the acceleration of the three algorithms is intended to improve the performance of GATK HC, the techniques and methods presented in this thesis can be extended to other DNA analysis tools. It would be interesting to use another DNA analysis tool to test these techniques and methods.

Finally, it took great effort to read the source code of GATK HC and undertake modification of the source code to integrate the accelerated implementations into GATK HC. Therefore, we recommend starting an effort to develop new DNA analysis tools and take ease of integration into consideration. The framework of the tools should be made friendly enough for other researchers to accelerate the computational intensive kernels on FPGAs and GPUs.

LIST OF PUBLICATIONS

INTERNATIONAL JOURNALS

1. **S. Ren**, N. Ahmed, K.L.M. Bertels, Z. Al-Ars, *GPU Accelerated Sequence Alignment with Trace-back for GATK HaplotypeCaller*. Accepted for publication in BMC Genomics, 2019

2. **S. Ren**, K.L.M. Bertels, Z. Al-Ars, *Efficient Acceleration of the Pair-HMMs Forward Algorithm for GATK HaplotypeCaller on Graphics Processing Units*, Evolutionary Bioinformatics, 14:1176934318760543, 2018

INTERNATIONAL CONFERENCES

1. **S. Ren**, N. Ahmed, K.L.M. Bertels, Z. Al-Ars, *An Efficient GPU-based de Bruijn Graph Construction Algorithm for Micro-Assembly*, 18th annual IEEE International Conference on BioInformatics and BioEngineering (BIBE 2018), October 29-31, 2018

2. **S. Ren**, K.L.M. Bertels, Z. Al-Ars, *GPU-Accelerated GATK HaplotypeCaller with Load-Balanced Multi-Process Optimization*, 17th annual IEEE International Conference on BioInformatics and BioEngineering (BIBE 2017), October 23-25, 2017

3. **S. Ren**, K.L.M. Bertels, Z. Al-Ars, *Exploration of Alternative GPU Implementations of the Pair-HMMs Forward Algorithm*, 3rd International Workshop on High Performance Computing on Bioinformatics (HPCB 2016), December 15-18, 2016

4. J.W. Peltenburg, **S. Ren**, Z. Al-Ars, *Maximizing Systolic Array Efficiency to Accelerate the PairHMM Forward Algorithm*, IEEE International Conference on Bioinformatics and Biomedicine (BIBM 2016), December 15-18, 2016

5. **S. Ren**, V.M. Sima, Z. Al-Ars, *FPGA Acceleration of the Pair-HMMs Forward Algorithm for DNA Sequence Analysis*, International Workshop on High Performance Computing on Bioinformatics (HPCB 2015), November 9-12, 2015

CURRICULUM VITÆ

Shanshan REN

Shanshan Ren was born in SiChuan, China in 1988. She received her Bachelor's degree in Network Engineering and Master's degree in Computer Science and Technology from National University of Defense Technology, Changsha, China. In September 2013, she started her PhD at the Quantum and Computer Engineering Department of the Delft University of Technology, where she was working under the supervision of Prof. Koen Bertels and Dr. Zaid Al-Ars. Her research interests include high performance computing and optimization.