Delft University of Technology

High Performance Seed-and-Extend Algorithms for Genomics

Ahmed, Nauman

DOI
10.4233/uuid:7e916f03-09cc-4510-9914-03a44b339462

Publication date
2020

Document Version
Final published version

Citation (APA)
Ahmed, N. (2020). *High Performance Seed-and-Extend Algorithms for Genomics*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:7e916f03-09cc-4510-9914-03a44b339462

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

High Performance Seed-and-Extend Algorithms for Genomics

High Performance Seed-and-Extend Algorithms for Genomics

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

Thursday 20 February 2020 at 10:00 o'clock

by

Nauman AHMED

Master of Science in Electrical Engineering
University of Engineering and Technology Lahore

born in Lahore, Pakistan

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

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

Independent members:

Prof.dr.ir. C. Vuik	Delft University of Technology
Prof.dr.ir. B. De Schutter	Delft University of Technology
Prof.dr. F. Silla	Polytechnic University of Valencia (UPV), Spain
Dr. G.W.E. Santen	Leiden University Medical Center (LUMC)
Dr. C. Hauff	Delft University of Technology

Printed by: Gildeprint
Front & Back: Nucleic acid double helix by Jack Moreh

Copyright © 2020 by N. Ahmed
All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the author. In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of Delft University of Technology products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to `http://www.ieee.org/publications_standards/publications/rights/rights_link.html` to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

ISBN 978-94-6384-108-5

An electronic version of this dissertation is available at
`http://repository.tudelft.nl`.

To my father

Acknowledgements

Doing a PhD, a research work spanned over many years, is not possible without constant support and help of the supervisors, family and friends. During my PhD, I came across many wonderful people that have contributed in some way in the successful completion of my PhD research.

I am indebted to Zaid Al-Ars; my supervisor, promotor and mentor; for his gracious support. It was Zaid who showed me this exciting research area of high performance genomics and the challenges that are required to be addressed. The biggest quality of Zaid is his positivity. In PhD research, you have to face many setbacks. Zaid was always able to find a positive side of these setbacks. He allowed me to work freely and explore new ideas which enabled me to produce some really good results. He was more than helpful in my research, starting from giving me a new idea, guiding the research and finally correcting my paper draft. In the course of my PhD, I learned a lot from Zaid about research methodology, presenting your research work, dealing with collaborators and research lifecycle in general.

I am thankful to Koen Bertels, my promotor, for his full support during my PhD. Koen was the head of the department and the group when I started my PhD. He performed his role exceptionally well by providing a healthy environment where everyone was comfortable in performing the research work. I am also thankful to my PhD committee for taking the time to assess my PhD thesis and providing me useful comments.

The secretariat staff of the Computer Engineering group which includes Lidwina, Joyce, Trisha and Laura, were all very helpful to me. I especially thank Erik, the network administrator of the group. He ensured that the machines I use are always up and running. He installed software and answer my emails even during the weekends. I am also thankful to Robbert Eggermont for providing me access to the INSY cluster that helped me to generate results for my first journal publication.

I am also thankful to my PhD research collaborators: Ernst, Giacomo, Hamid, Johan, Jonathan, Petros, Shanshan, Tanveer, Tongdong and Vlad for their help and support. I had many lively discussions with them which helped me to improve my work and learn more about computer engineering. Working at the Computer Engineering group was made enjoyable by my colleagues: Anh, Baozhou, Innocent, Jian, Johan, Joost, Li, Mota, Nader, Nicoleta, Shanshan and many others. Baozhou was very positive about my research work, which was very encouraging for me.

I have been lucky to have very nice office mates. My old office mates Imran Ashraf and Hamid Mushtaq were supportive throughout the time I spent with them. Imran helped me even in the smallest things like installing software on my desktop and laptop. Whenever I was stuck in my work he was always there to help me. Hamid helped me to understand other research works due to his vast knowledge of programming languages. My new office mate Tanveer motivated me to take on interesting research ideas. We had many exciting discussions on the latest computer memory technologies.

Living in another country away from your home is not always easy. However, I always felt at home by the company of my Pakistani friends: Aftab, Ahmed, Akram, Aleem, Fahim, Hamza, Hassan, Hamid, Hanan, Hussam, Huzaif, Imran, Irfan, Junaid, Mohsin, Muneeb, Nauman, Qasim, Saad, Salman, Samad, Shakeel, Tabish, Tanveer, Usama, Usman, and many others. I spent enjoyable weekends with Fahim, Imran, Irfan, Hanan, Hussam, Saad, Shakeel and Tabish. Imran and Qasim always hosted us with a big heart. Hanan and Hussam cooked delicious food for us while Irfan and Saad provided me with useful tips about my research work. I will miss the funny incidents that Shakeel use to narrate us. I am also thankful to my colleagues Frahan and Adeem at UET Lahore for taking care of my official matters.

I am extremely grateful to my sisters Monazza and Shazia for their affection and constant support. They performed all the duties of a son while I was away from my parents. Remembering my playful niblings Haaniah, Areej and Momin helped me to relax in tough hours of work.

I cannot express my gratitude to my wife Madiha for her cooperation and support during my PhD studies. We went through difficult times during my PhD but she always remained steadfast. I spent many weekends and had to work till late in the evenings. She was very understanding and took good care of the children and home while I was at work. My children Yumna and Musa are the source of happiness for me. Playing with them or just watching them would relieve me from all my PhD worries.

The PhD research would have not been possible without prays and sacrifice of Ammi and Abbu. Ammi had to bear my absence. She missed me a lot but always instructed me to focus on my PhD research. My heart is filled with the pain while remembering Abbu. He is the one who motivated me to do a PhD from abroad even knowing that his health is not in a good state. He used to get an update on my PhD work every day and was eagerly waiting for its completion. Alas! it does not happen in his lifetime. Abbu! I will always miss you. I know that you will be watching me from the heavens and will be very happy about the completion of my PhD.

Nauman Ahmed
January 23, 2020
Schiedam, Netherlands

Contents

1

Introduction

DNA, the *book of life*, controls the growth and appearance of all living beings. A *genome* is the complete set of DNA in a living organism. *Genomics* is the study of genomes. In genomics, genome analysis or in other words DNA analysis is performed to extract useful information about the characteristics of an organism. This chapter will present the motivation for the work done in this PhD thesis by outlining various computational challenges in genome analysis. The chapter is organized as follows: Section 1.1 discusses DNA sequences technologies and their applications. Section 1.2 describes the various DNA analysis algorithms. Section 1.3 give details of the computing platforms for DNA analysis. Section 1.4 explains the current challenges in the fast analysis of DNA data, followed by Section 1.5 which outlines possible solutions for these challenges. Section 1.6 describes the research questions addressed in this thesis, while Section 1.7 lists the contributions of the thesis. Section 1.8 presents the dissertation outline.

1.1. DNA sequencing methods and applications

A genome or DNA sample can be regarded as a string made up of only four types of characters ′a′, ′c′, ′g′ and ′t′ representing the four nucleotide bases: Adenine, Cytosine, Guanine and Thymine, respectively. For DNA analysis, the order of the bases in the DNA sample must be known. One way to identify the DNA content of a sample is by using microarrays. A DNA microarray has a solid surface containing a collection of ″spots″. Each spot has a set of unique DNA sequences attached to it. To determine whether the target DNA consists of sequences identical to those on the DNA microarray, the DNA is shattered into small *fragments* and then a solution of these small DNA fragments is sprinkled over the DNA microarray. The

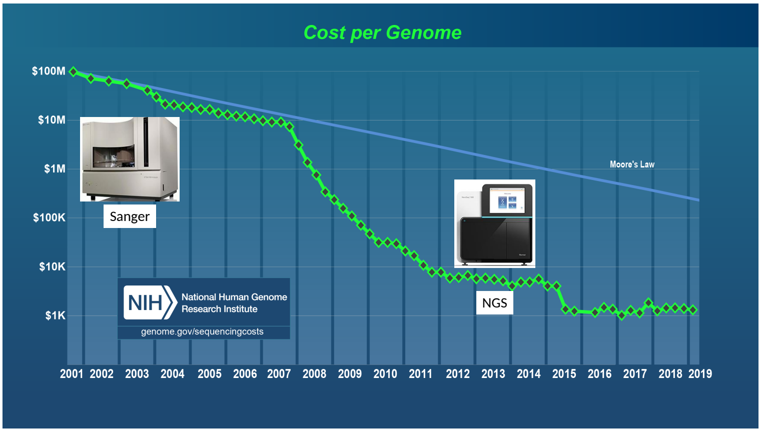


Figure 1.1: Cost reduction due to Next Generation Sequencing. The figure is reproduced from [1].

small DNA fragments bind to their complementary DNA sequences present in the DNA microarray. In this way, scientists can detect the relative concentration of the DNA sequences in the target DNA sample. In [3], Bumgarner describes various limitations of DNA microarrays regarding their resolution. The microrarrays are used to determine the relative concentration of the known DNA sequences in the solution and cannot be used to find the order of bases in a sample. The solution of the target has to be prepared very carefully. A solution containing a very high or very low concentration of the DNA sequences may give no information about the relative concentration of the DNA sequences in the target DNA sample. Furthermore, if different DNA fragments share a common sequence they will bind to the same spot, despite being different. DNA microarray can only be designed for known DNA sequences. Sequences that are unknown or vary greatly from the known sequences cannot be detected with DNA microarrays. Hence, DNA microarrays may not be suitable for certain critical applications that require high resolution DNA sequence analysis.

An alternative approach to find the order of bases in a DNA sample is *DNA sequencing*. DNA sequencing machines are capable of reading a DNA sample to find the exact order of bases in it. As a DNA sequencing machine reads a given sequence at a per-base level, it does not have the limitations faced by DNA microarrays. Therefore, DNA sequencing is replacing DNA microarrays for high resolution DNA analysis. The machine used for DNA sequencing is known as a DNA sequencer. As in the case of DNA microarrays, the DNA sample is shattered into fragments before sequencing. The DNA sequencing machines scan through the fragments to read the bases. The output of the machine is the sequence of bases known as *DNA reads*. The *read length* is the number of bases in a read, specified in *base pairs* (bp). First-generation DNA sequencers use Sanger sequencing [4] producing reads up to 1000 bp. However, Sanger sequencing is only suitable for sequencing smaller

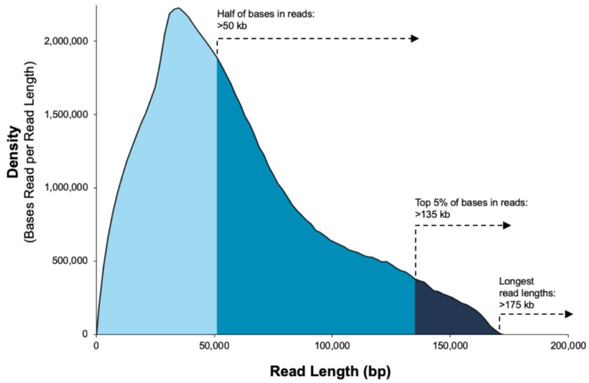


Figure 1.2: Read length distribution for Pacbio sequencing. Reproduced from [2].

genomes due to its low throughput.

As the number of applications grows, scientists faced the need to reduce the cost of high-throughput DNA sequencing of large genomes. This resulted in the development of second-generation massively parallel sequencing technologies more commonly known as *Next Generation Sequencing* (NGS). Figure 1.1 shows that with the advent of NGS in 2007, the cost of sequencing begins to decrease sharply bringing down the cost of sequencing a human genome to around US $1000 in 2015. In this period, sequencing cost dropped by around 10,000x, from US $10 million in 2007 to US $1000 in 2015. The drop in cost is accompanied with an increase in throughput, where current NGS machines can produce 6 terabytes of data in a single machine run of up to 3 days [5]. NGS reads have lengths of up to 300 bp, and therefore they are referred to as *short DNA reads*. Illumina is the leading manufacturer of NGS sequencers.

Third generation sequencers manufactured by Pacific Biosciences (Pacbio) and Oxford Nanopore Technologies (ONT), can produce reads up to hundreds of kilobases long. Figure 1.2 shows the read length distribution of Pacbio sequencers. The average read length is around 50 kilobases with the longest read being more than 175 kilobases. ONT sequencers produce ultra-long reads having a similar read length distribution as shown in Figure 1.2 but with even higher average length.

DNA sequencing has a wide range of applications from nutrition to life sciences. According to an economic survey, the global sequencing market will grow from around 6 billion US dollars in 2017 to more than 25 billion dollars in 2025 with a cumulative annual growth rate of 19% [6]. Examples of the applications of DNA sequencing are listed below:

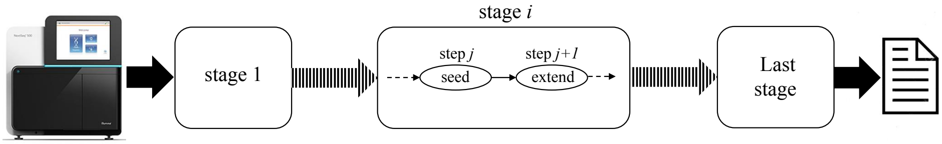


Figure 1.3: A multistage DNA analysis pipeline. Each stage is a DNA analysis application. Input is the DNA reads generated by a sequencer.

- prenatal testing to find abnormalities in an unborn baby [7]
- determining the risk of developing a certain genetic disease [8]
- diagnosing an existing genetic disease [9]
- breeding improved varieties of crops [10]
- improving the efficiency of livestock production [11]
- finding bacteria and viruses in blood and food samples [12] [13]
- etc.

Hence, genome sequencing is becoming the norm rather than the exception in many fields, which will play an increasingly important role in our wellness and quality of life.

1.2. DNA analysis algorithms

Reading the DNA bases in a sample is the first step in the process of DNA analysis. Sequencing data is stored as DNA reads in a computer readable data format, e.g. FASTQ format. DNA analysis is performed by computer programs that process the DNA sequencing data to perform the desired analysis. Many DNA analysis tools have to match two DNA strings P and T. A straight forward application of string matching algorithms is not feasible for DNA sequencing data due to the large volume of data to be analyzed. Therefore, string matching in modern DNA analysis applications is performed in two steps:

1. Finding substrings of P and T that highly match with each other. This step is known as *seeding* and these substrings are known as *seeds*. Different kinds of seeds can be computed, e.g. k-mer matches [14], maximal exact matches [15], etc.
2. Comparing P and T around the highly matching substrings found in the seeding step. This step is known as *seed-extension*. The operation performed in step-2 is known as *Sequence Alignment*. Sequence alignment is the process

of transforming two or more sequences by introducing gaps or substitutions so that parts of the sequences become identical to each other. Various sequence alignment algorithms exist, e.g. local alignment [16], global alignment [17] etc.

This two-step technique is known as the *seed-and-extend* heuristic, pioneered by BLAST [18]. If the length of P and T is small, the seeding step can be skipped and P is directly compared against T using sequence alignment algorithms.

The DNA analysis may be performed by a single application or a pipeline of applications as shown in Figure 1.3. A pipeline stage is a single DNA analysis application. The output of a pipeline stage acts as the input of the next stage. Each pipeline stage consists of one or more *processing steps*. Seeding and extension are one of the processing steps of a pipeline stage. *DNA analysis algorithm* is a general term that we can use to refer to the whole DNA analysis pipeline, an analysis stage in the pipeline or a processing step within a pipeline stage. Below is a list of some DNA analysis applications having seeding and extension (or sequence alignment) as processing steps.

- Modern DNA read mappers (also known as read aligners) like BWA-MEM [19], Bowtie2 [20] and Minimap2 [21] follow seed-and-extend paradigm.
- Overlap computation in OLC (Overlap Consensus Layout) based assembly of long DNA reads is also performed using seed-and-extend. Examples are Darwin [22] and Daligner [23].
- Whole genome alignment tools like MUMmer4 [24] and LASTZ [25] are based upon seed-and-extend.
- DNA read error correction applications like Jabba [26] use seed-and-extend.
- DNA analysis applications like HaplotypeCaller [27], DeepVariant [28] and Strelka [29] etc. form the last stage in the variant calling pipeline. Sequence alignment is an important processing step in these applications
- DNA database search engines like BLAST [18] select a subset of all the database sequences by performing seeding. The selected sequences are aligned to the query. Other tools like DOPA [30] align all the sequences in the database to the query sequences.

1.3. Computing platform for DNA analysis

DNA analysis algorithms can be executed on high performance computing platforms. A typical high performance computing platform is shown in Figure 1.4,

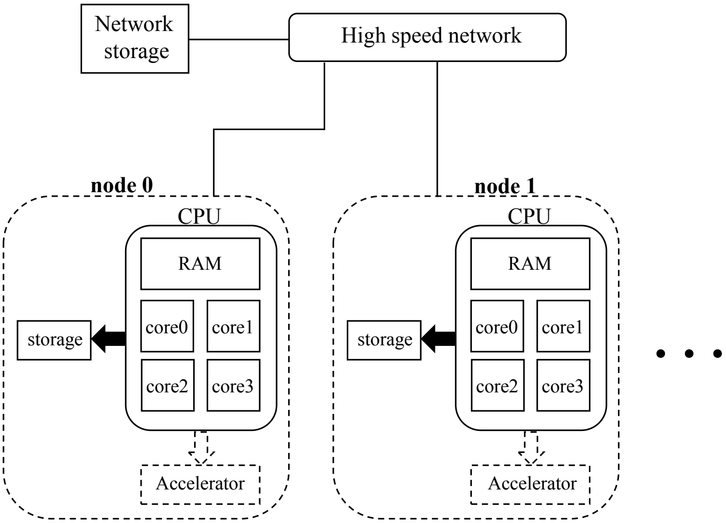
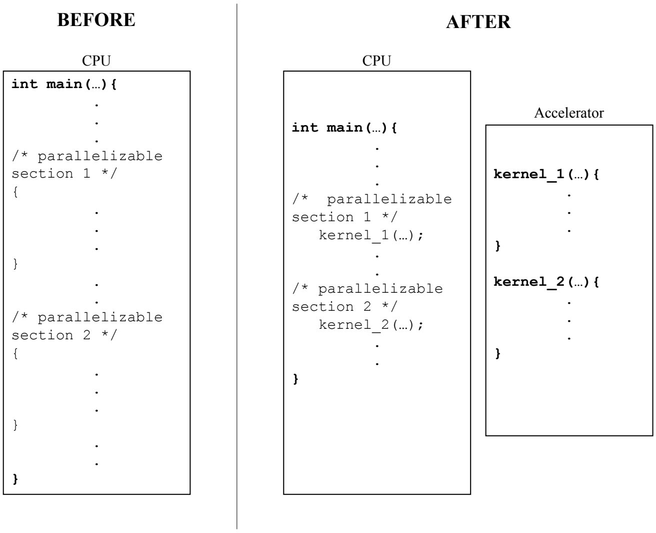


Figure 1.4: A typical computing platform for DNA analysis. The CPU may physically consist of one more multicore processors known as *sockets*.



```
int main(…){
          .
          .
          .
/* parallelizable
section 1 */
{
          .
          .
          .
}
          .
          .
/* parallelizable
section 2 */
{
          .
          .
          .
}
          .
          .
}
```

```
int main(…){
          .
          .
          .
/*  parallelizable
section 1 */
   kernel_1(…);
          .
          .
/* parallelizable
section 2 */
   kernel_2(…);
          .
          .
}
```

```
kernel_1(…){
          .
          .
          .
}
kernel_2(…){
          .
          .
          .
}
```

Figure 1.5: The DNA analysis algorithm execution before and after offloading highly parallelizable sections to an accelerator.

Metric	**CPU**	**FPGA**	**GPU**
Amount of parallelism	★★	★★★★	★★★★
SIMD parallelism	★★★★★	★★★	★★★★★
MIMD parallelism	★★★★★	★★★★★	★
Sequential performance	★★★★★	★★★	★
Memory bandwidth	★★★★	★	★★★
Power efficiency	★	★★★★★	★★
Multiple kernels	NA	★★	★★★★★

Table 1.1: A comparison of various metrics of CPU with FPGA and GPU based accelerators for high performance computing. The metric performance is directly proportional to the number of stars.

although the configuration may vary. It consists of one or more *compute nodes*. A *cluster* is a computing platform with at least two compute nodes. Such a cluster has shared network storage. The compute nodes and shared storage are interconnected via a high speed network. Each compute node contains a CPU with multiple cores connected to shared memory. Each compute node also has a local storage space. Besides, an accelerator may also be attached to the CPU. A node consisting of an accelerator is known as a *heterogeneous computing system*.

The accelerator consists of a processing device and memory. Mainly two types of accelerators are in use today: FPGA-based and GPU-based. The main advantage of the accelerators over the CPU is the large amount of parallelism offered by these systems. In a heterogeneous compute system containing an accelerator, the CPU acts as a *host* which offloads highly parallelizable sections of the algorithm to an accelerator. The programmer has to write an implementation of the highly parallelizable section that can be executed on the accelerator device (GPU or FPGA). The implementation is known as a *kernel*. Figure 1.5 shows the DNA analysis algorithm before and after using an accelerator. The algorithm running on the CPU consists of highly parallelizable sections that are offloaded to the accelerator. The kernels contain the implementation of the parallelizable sections which execute on the accelerator.

Table 1.1 shows a comparison of various metrics of CPU with FPGA (Field Programmable Gate Array) and GPU (Graphics Processing Unit) based accelerators that are required for high performance computing. CPUs offers both SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data) parallelism

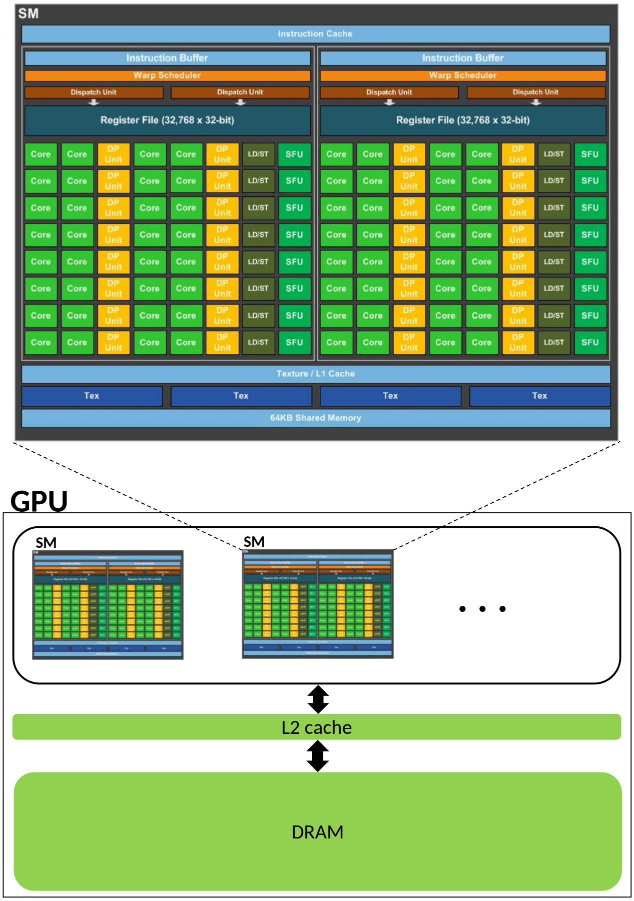


Figure 1.6: A simplified view of NVIDIA's Pascal architecture with the magnified view of the streaming multiprocessor (SM) shown on the top. *DP units* are double precision cores; *LD/STs* are load-store units;*SFUs* are special function units; *TEXs* are texture units. The figure is reproduced from [31].

with very high sequential performance. FPGA offers much higher parallelism than CPU. It also has much less power consumption than CPU and GPU. MIMD as well as SIMD parallelism can be achieved with FPGAs. GPU is well placed in the list due its massive amount of SIMD parallelism and high memory bandwidth. Multiple sections of an algorithm can be parallelized by writing a separate kernel for each section. FPGAs can also be used for parallelizing multiple sections of the algorithm, but it requires reconfiguration of the FPGA which involves some overhead. GPUs were originally designed for graphic rendering. But its many-core design created an opportunity to parallelize many other applications. The GPU architecture varies from one vendor to the other and even across different GPU generations from the same vendor. Here we give a general overview of state-of-the-art NVIDIA GPUs. Figure 1.6 presents a simplified view of the internals of NVIDIA GPUs based on the

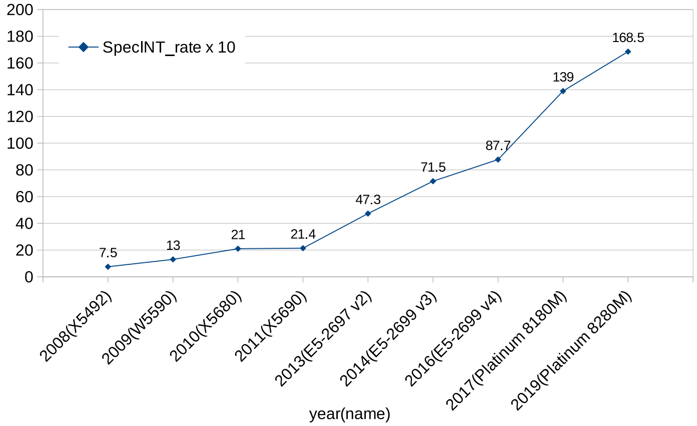


Figure 1.7: The SpecINT_rate performance of various generations of Intel Xeon server-class processors. The fields on the x-axis are the launching year of the processor with the processor name in the brackets.

Pascal architecture. The cores of a GPU, known as streaming processors (SPs), groups of which are organized into a number of streaming multiprocessors (SMs). Each SM has a set of SPs, a register file, warp schedulers, a read only memory (not shown in figure), L1 cache, shared memory, and some other hardware units. In NVIDIA GPUs, a *warp* is a set of 32 threads that execute in lockstep by sharing the program counter. All SMs access the DRAM (known as global memory) through a shared L2 cache. The programming language for developing kernels for NVIDIA GPUs is known as *CUDA* which is an extension of C/C++. The data to be processed by the kernel is first copied from the host CPU memory into the global memory of the GPU. The CPU then launches the kernel. Once the kernel is finished the results are copied from the global memory back into CPU memory.

1.4. Challenges in fast DNA analysis

Low cost sequencing and analysis promises to make DNA analysis affordable enough to be used by a larger public in many sectors ranging from healthcare to nutrition. However, several challenges need to be overcome to achieve this goal. Second and third generation sequencing platforms produce massive amounts of data that has to be processed by DNA analysis algorithms. Therefore, this makes DNA analysis a time consuming task. Moreover, in many critical DNA analysis applications like clinical diagnostic, the time consumed in DNA analysis may become a matter of life and death. In the following, we will present the current challenges in fast DNA analysis.

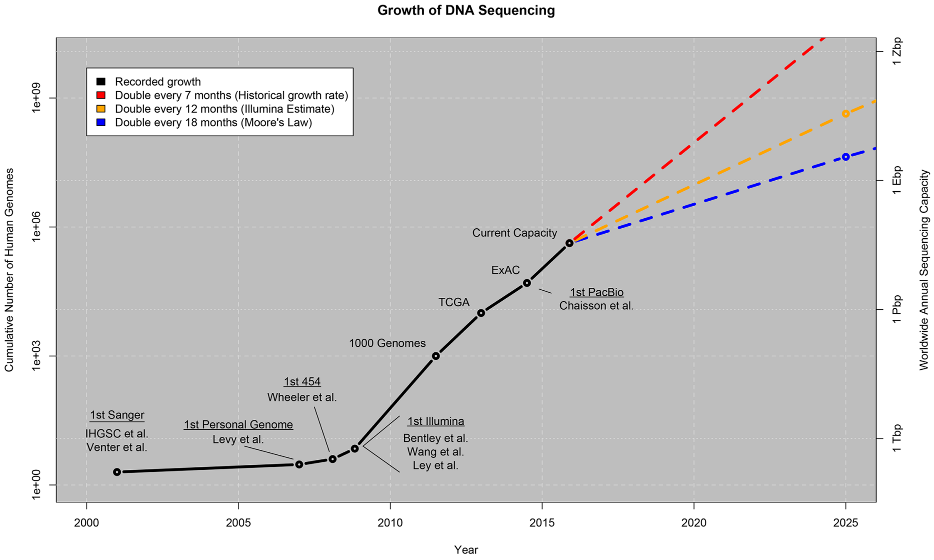


Figure 1.8: The growth of DNA sequencing data. The dotted lines show the projected growth. Reproduced from [32].

1.4.1. Slow CPU performance scaling

Moore's law accompanied with Dennard scaling resulted in an exponential growth in CPU performance until it hit several barriers: the power wall, the memory bandwidth wall, which resulted in limiting CPU performance scaling. Figure 1.7 plots the throughput performance of Intel Xeon server-class processors on the SPEC CPU2006 integer benchmarks. The graph in the figure shows CPU performance scaling in the last 11 years. The performance in the first 5 years of the figure (2008 to 2013) has increased by 6.3x (i.e., 1.44x per year). However, in the last 6 years from 2013 to 2019, it only increased 3.56x (i.e., 1.23x per year). Hence, the performance gain is reduced by nearly a factor of 2 in the last 6 years.

1.4.2. DNA sequencing data growth

Figure 1.8, reproduced from [32], shows that the amount of DNA sequencing data generated is doubling every 7 months. The authors predicted that if the current trend continues (historical growth rate in Figure 1.8) then by 2025 the DNA sequencing data will reach 5 Zetta bases per year with 2 billion human genomes being sequenced every year. *Variant calling* is a DNA analysis to detect mutations in a genome. In [33], a cluster implementation of a GATK genome analysis pipeline for variant calling [34] is presented. The implementation can perform the analysis of 200 Gbps sequencing data in 90 minutes. But it requires 19 IBM Power8 compute nodes to do so. Each node has 20 cores of IBM Power8 and 512 gigabytes of

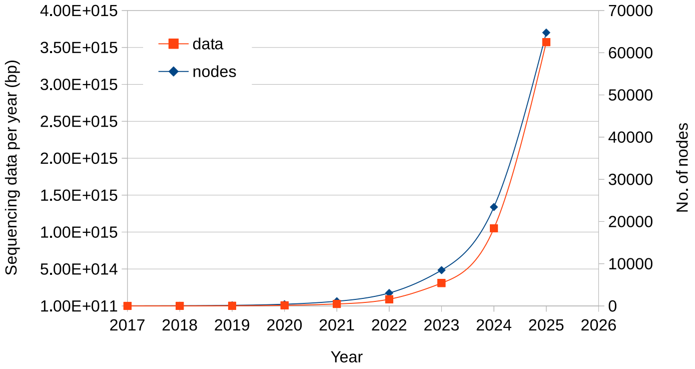
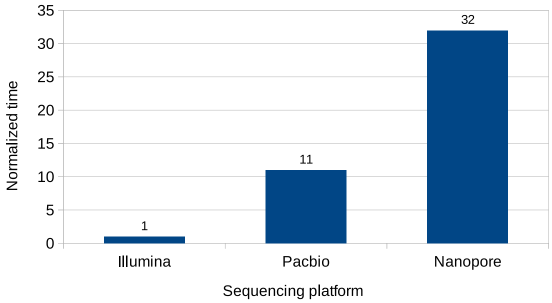


Figure 1.9: Using the DNA data growth prediction in [32] we can extrapolate the computational needs required to perform the variant calling based on the results published in [33]. The left Y-axis and the line labeled as ”data” shows the sequencing data per year. The right Y-axis and the line labeled as ”nodes” shows the number of compute nodes.

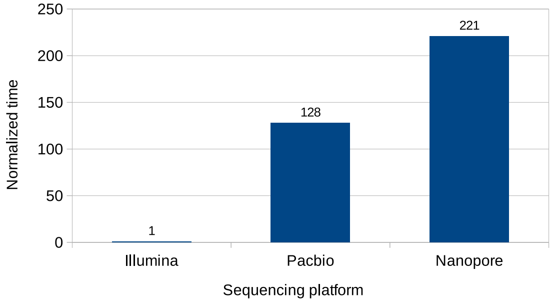
RAM. This shows that sufficient computing resources are required to process the sequencing data in a reasonable amount of time. A close inspection of the historical growth rate in Figure 1.8 reveals that the sequencing data is growing by 3.4x every year. Based on this prediction in [32], about DNA sequencing data growth and the results of the cluster implementation in [33], we projected the number of IBM Power8 nodes required to perform variant calling, if the input sequencing data of [33] (i.e., 200Gbp) grow by 3.4x every year. Figure 1.9 shows our projection. For predicting the number of nodes, we assume computational power of the CPUs will increase at a rate of 23% per year as calculated from Figure 1.7 in Section 1.4.1. The graph shows exponential growth for both the sequencing data and the required number of nodes to perform the analysis. The sequencing data is 3.4x more than the previous year while the required number of nodes is increased by around 2.75x every year. By 2024, nearly 23 thousand Power8 nodes will be required to perform the analysis on 1 Peta bases. In 2025, the data grows to 3.5 Peta bases and the required number of nodes become nearly 65 thousand. Hence, to keep the compute time constant, old nodes have to be replaced by newer 1.23x faster nodes and at the same time use more than double (2.76x) the number of nodes. In this analysis, we have neglected the amount by which the storage and interconnect network have to be scaled up to avoid degradation in performance. Such a huge financial investment is not sustainable.

1.4.3. Long DNA reads

State-of-the-art third generation DNA sequencers produce long reads with lengths ranging up to hundreds of kilobases. We measured the execution time of seeding and extension for the same amount of sequencing data generated by Illumina,



(a) Seeding



(b) Sequence alignment (seed-extension)

Figure 1.10: Relative execution times of seeding and extension for the reads generated by three second and third generation sequencing platforms.

Pacbio and Oxford Nanopore sequencers. Seeding and extension for long DNA reads take considerably more time than for Illumina short reads as shown in Figure 1.10a and Figure 1.10b. Seeding of Pacbio reads takes 11x more time than short Illumina reads. For Oxford Nanopore, which produces even longer reads than Pacbio, the seeding time is 32x slower. In the case of sequence alignment, the gap in the execution time for short and long reads widens even more where extension with Pacbio and Nanopore reads takes 128x and 221x more time than Illumina reads, respectively. Hence, the seed-and-extend of long DNA reads is a bottleneck in DNA analysis applications.

1.4.4. DNA sequencing time

The DNA sequencing time of the Illumina sequencers is a major bottleneck in the DNA analysis. The latest NovaSeq sequencing platform produces a maximum of 2.5 Tera bases in 36 hours [5]. However, the cluster implementation of variant calling described in [33] can process this amount of data in about 18.5 hours, which is nearly half the time needed for sequencing. Moreover, the implementation is highly scalable, where increasing the number of nodes in the cluster will reduce the computation time. Further accelerations and optimizations are also possible that will further reduce the execution time of different pipeline stages, such as reducing the variant calling stage by tools like elprep [35]. Hence, a highly optimized and accelerated DNA analysis algorithm will not help to reduce the overall analysis time if the sequencing itself is taking much more time than the computation.

1.5. Overcoming the challenges of DNA analysis

In the previous section, we described that the challenges faced in fast DNA analysis are due to the massive amount of sequencing data available, its exponential growth rate, as well as the increasing DNA read length. These challenges result in the continued increase in DNA analysis time, a problem that can be addressed in the following ways.

1.5.1. Acceleration of DNA analysis algorithms

In many DNA analysis algorithms, there are several independent tasks and operations. The most common way of reducing execution time is to accelerate the analysis algorithm by running these independent tasks in parallel. A CPU core on a computing platform (see Figure 1.4) can execute an independent task. Moreover, CPU cores have SIMD (Single Instruction Multiple Data) units that offer parallelism at an even finer granularity within a task. A large number of previous research efforts are based on the parallelization of DNA analysis algorithms. In some researches, the parallelization is performed at the cluster level by efficiently utilizing the cluster resources. Alternatively, the focus of the research may be on the parallelization of DNA analysis algorithms on a single node or even on a single core using the SIMD units. For example, [33] and [36] present a cluster implementation of a DNA analysis algorithm; [37] and [38] are parallelization of sequence alignment algorithms on a CPU using multiple cores and SIMD units. As heterogeneous computing is gaining immense significance in the era of high-performance computing, many researchers have used GPU and FPGA based accelerators to increase the performance of DNA analysis algorithms by accelerating various steps of the application [39]. For example, [40] and [41] present the GPU and FPGA acceleration of the BWA-MEM seed-extension algorithm, respectively. Similarly, in [42] and [43] the PairHMM computation step of the GATK variant calling algorithm is accelerated on GPU and FPGA, respectively.

1.5.2. Optimization of DNA analysis algorithms

In this approach, the sequential performance of DNA analysis algorithms is improved via software optimizations. In the following, some optimizations are described that can be applied to reduce the execution time:

- ***Improving cache locality***: Increasing cache hits helps to gain more performance on a CPU. Techniques presented in [44] and [45] optimize seeding algorithms by increasing the cache hit rate.

- ***Reducing memory references***: Fewer memory references reduce the execution time of memory-bound applications. Work in [46] shows a seeding algorithm that is faster than the original implementation due to less memory load instructions. Other techniques involve exchanging computation for memory accesses, e.g. building a so-called ”query profile” reduces the memory accesses in sequence alignment [47].

- ***Reducing computation***: Using methods to avoid redundant computation helps to reduce the execution time. For example, in banded sequence alignment, proposed in [48], less computation is required to align two sequences without affecting the final result.

- ***Reducing I/O***: In many DNA analysis algorithms, I/O is a major bottleneck that can be reduced by various methods, such as using in-memory data formats presented by ArrowSAM [49].

1.5.3. Designing efficient algorithms

The performance of a highly optimized and accelerated algorithm to perform a specific type of analysis can only be further boosted by designing a new algorithm that is faster than the existing algorithm without losing accuracy. For example, [50] proposes a better approach for computing maximal exact matching seeds. Similarly, the algorithm in [22] can be used for fast local alignment of long DNA sequences with the same accuracy as that of the Smith-Waterman algorithm [16] which is the standard method for computing the local alignment. For DNA read mapping of Illumina reads, BWA-MEM is faster and more accurate than BWA-SW [51] as shown in [19]. The developers of GATK claim that the new algorithm version 4 is faster than version 3 without sacrificing accuracy.

1.6. Research questions

As described in Section 1.5, many solutions exist to meet the challenges of fast DNA analysis and researchers are spending their effort to employ these solutions to speed up the analysis. However, many areas need improvement since there are still some unaddressed challenges. In this thesis, we answer the following research questions:

1. How can the performance of the algorithm for computation of maximal exact matching seeds be improved?

   Maximal exact matching seeds are used in many DNA analysis applications. The algorithm for computing maximal exact matching seeds is memory bound and cache unfriendly.

2. How can the computation of maximal exact matching seeds for long DNA reads/sequences be accelerated on a GPU?

   State-of-the-art third generation sequencers produce long DNA reads. The computation of maximal exact matching seeds for long reads/sequences is a time expensive task. Seed computation is a highly parallel process which makes it suitable for GPUs. As heterogeneous computing systems are becoming commonplace, there is a need for a high performance GPU acceleration for maximal exact matching seeds computation of long DNA sequences.

3. How can a GPU accelerated library for the sequence alignment of high-throughput NGS reads be developed?

   Sequence alignment is performed in many DNA analysis algorithms. These algorithms have to process large amounts of DNA sequencing data. In many applications, a large number of sequence alignments can be computed in parallel. GPUs offer massive parallelism which is highly suitable for this purpose. However, there is only one GPU accelerated sequence alignment API, NVBIO [52]. It has limited performance and does not use the latest features offered by NVIDIA's GPUs and their programming models. Therefore, a fast GPU accelerated library that contains various sequence alignment algorithms is useful for the developers of DNA analysis pipelines.

4. How does the seeding and extension algorithm affect the speed and accuracy of DNA read mappers?

   Modern DNA read mappers are based on the seed-and-extend paradigm. There are different types of seeding algorithms. Similarly, various sequence alignment algorithms can be used in the extension step. The alternatives available for performing seeding and extension in read mappers call for the need to compare the speed and accuracy of different combinations of seeding and extension algorithms.

5. How to reduce the impact of DNA sequencing delay in the DNA analysis time? The time spent in DNA sequencing is a major bottleneck in the overall DNA analysis time as described in Section 1.4.4. Therefore, developing methods and techniques to overcome this bottleneck is important to reduce the DNA analysis time.

1.7. Thesis contributions

In this section, we present our contributions by addressing the research questions posed in Section 1.6. We provide solutions by accelerating and optimizing DNA analysis algorithms as well as for designing efficient algorithms. Most of the existing accelerated kernels for DNA analysis are designed to fit for a particular application and hence are deeply integrated. Therefore, the solutions cannot be used to accelerate a similar analysis in other applications. Our accelerated algorithms are faster than existing implementations and at the same time reusable i.e. they can be easily integrated into DNA analysis algorithms, which makes them unique among many other optimized and accelerated solutions. We also employ our optimized and accelerated algorithms in real DNA analysis applications to showcase the efficacy of our approach. In the following, we list the contributions of the thesis.

Contributions to algorithm optimization

1. We present an improved algorithm for computing the maximal exact matching seeds which is 1.7x faster than the original algorithm. This contribution is presented in Chapter 2.
2. We used our software optimized algorithm for computing maximal exact matches in the BWA-MEM DNA read mapper and showed that it raises the speedup of FPGA accelerated BWA-MEM from up to 2.6x. This is presented in Chapter 2.

Contributions to algorithm acceleration

1. We accelerated the computation of maximal exact matches on GPUs. Our GPU acceleration is much faster than CPU based implementations. The GPU implementation is available in the form of a CUDA API and can be integrated into any DNA analysis algorithm. Our implementation is up to 9x faster for computing maximal exact matching seeds as compared to the fastest CPU implementation running on a server-grade machine with 24 threads. This contribution is described in Chapter 2.
2. We present GASAL2, a GPU accelerated API for the sequence alignment of high throughput NGS sequencing data. NVBIO [52], NVIDIA's library, is the only other library available for this purpose. GASAL2 is up to 21x faster than high-performance CPU libraries running on an Intel Xeon system with 28 cores and up to 13x faster than NVBIO. This contribution is presented in Chapter 3.
3. Using the GASAL2 library, we accelerated the seed-extension step of the BWA-MEM DNA read mapper. GASAL2 seed-extension is up to 20x faster, which speeds up the whole BWA-MEM application by 1.3x as compared to 12 Xeon threads. This contribution is described in Chapter 3.

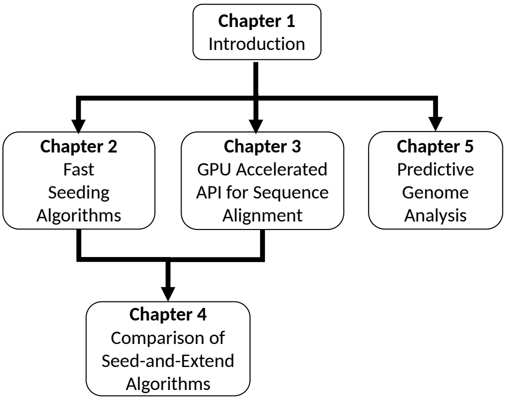

Chapter 1
Introduction
Chapter 2
Fast Seeding Algorithms
Chapter 3
GPU Accelerated API for Sequence Alignment
Chapter 5
Predictive Genome Analysis
Chapter 4
Comparison of Seed-and-Extend Algorithms

Figure 1.11: Recommended reading order of the dissertation.

4. We accelerated the Darwin read overlapper, used in the assembly of long DNA reads. Our GPU acceleration is 109x faster versus 8 CPU threads of an Intel Xeon machine. This contribution is discussed in Chapter 3.

Contributions to designing efficient algorithms

1. We developed GASE, a generic read mapper based upon seed-and-extend. It contains many optimized and accelerated seeding and extension algorithms. In a given study, GASE can be configured with any combination of seeding and extension algorithm to meet the requirements of accuracy and execution time.

2. We explored the design choices for DNA read mappers for NGS data. We employed different algorithms in the seeding and extension step of the mapper and compared the accuracy and execution time of the alternatives. This contribution is discussed in Chapter 4.

3. We present a novel method to overcome the long DNA sequencing time bottleneck for Illumina sequencers. We showed that our approach can help to further reduce the end-to-end analysis time by starting the analysis before sequencing is completely finished without much loss in the accuracy of the DNA analysis. This contribution is discussed in Chapter 5

1.8. Dissertation outline

In this chapter, we have discussed the importance of DNA sequencing and the current challenges we are facing in the fast analysis of the sequencing data. We

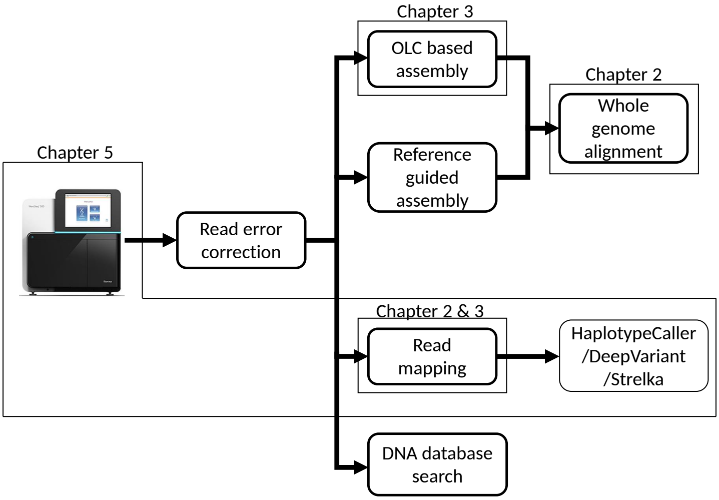


Figure 1.12: Various possible DNA analysis workflows with stages based upon seed-and-extend or at least use sequence alignment in a computational step. Also showing the applications optimized and/or accelerated in the thesis.

outline the solutions to overcome these challenges and our contributions to fast DNA analysis. The remaining dissertation describes the algorithms and methods that we developed to speed up the DNA analysis in detail. The outline of the dissertation is as follows

Chapter 2 focuses on improving the performance of seeding algorithms. It first presents a software optimized seeding approach, followed by a GPU accelerated version of the algorithm.

Chapter 3 focuses on improving the performance of sequence alignment algorithms. It describes the GPU acceleration of a sequence alignment API. The API is then applied to accelerate the seed-extension stage of BWA-MEM. Furthermore, it also presents the GPU acceleration for Darwin, a read overlapper for long DNA reads.

Chapter 4 presents a comparison of seeding and extension algorithms, identifying their speed and accuracy. It also describes a read mapper, known as GASE, that we developed to perform the comparison.

Chapter 5 describes our approach to reduce the DNA sequencing delay.

Chapter 6 draws conclusions form the dissertation and proposes future research directions.

Figure 1.11 shows the relationship between different chapters and the recommended order for the reader. After reading the introduction chapter (Chapter 1), the reader can move on to Chapter 2, Chapter 3 or Chapter 5. It is recommended that Chapter 4 should be read after reading Chapter 2 or Chapter 3.

We also applied our optimized and accelerated seeding and extension algorithms to real DNA analysis applications to show the performance improvement achieved. Figure 1.12 shows various possible workflows for analyzing DNA sequencing data. All the DNA analysis applications shown in various stages in the figure commonly use the seed-and-extend paradigm or at least perform sequence alignment in one of their computational steps. The figure also shows the stages that we accelerated using our optimized and accelerated seeding and extension algorithms. The reads generated by the DNA sequencer are corrected to remove the sequencing errors in the ”Read error correction” stage. The corrected reads can be assembled using different assembly algorithms. OLC based assemblers and reference guided assembly algorithms contain seed-and-extend processing steps. The GPU accelerated sequence alignment algorithms in Chapter 3 are used to accelerate the read overlap computation in OLC based read assemblers. Assembled genomes are compared with each other using ”Whole genome alignment” algorithms. Chapter 2 describes the GPU accelerated maximal exact match seeding algorithm that can be used to accelerate whole genome alignment applications (such as MUMmer). Alternatively, DNA reads are mapped. Modern DNA read mappers are based on the seed-and-extend approach. Chapter 2 also presents a CPU-optimized seeding algorithm used to speed up the BWA-MEM DNA read mapper. In Chapter 3, we have shown our GPU accelerated sequence alignment API that can be used to accelerate the seed-extension stage of BWA-MEM. Mapped DNA reads are used for variant calling using the HaplotypeCaller, DeepVariant, Strelka, etc. All these tools contain sequence alignment as an important processing step. The techniques in Chapter 5 decrease the end-to-end time by reducing the sequencing delay in a variant calling pipeline using Illumina reads. Short DNA reads generated by Illumina sequencing platforms have a very low sequencing error rate, and hence do not require read correction (as represented by the block in the figure labeled as Chapter 5). Reads are also searched in DNA databases using search engines to find a matching pattern. Most of the search engines use sequence alignment algorithms while many others follow the seed-and-extend paradigm.

2

Fast Seeding Algorithms

In seed-and-extend DNA analysis algorithms, the first step is to find seeds (which are exactly matching substrings) of the two DNA sequences to be compared as described in Section 1.2. In this chapter, we describe our accelerated and optimized seeding algorithms for genome analysis. The chapter focuses on the computation of maximal exact matching seeds which is a compute intensive task that may become a bottleneck in many DNA analysis applications. We first describe *GPUseed*, which is our GPU library for accelerating the computation of maximal exact matching seeds. We show how the seeds can be computed using space efficient FM-index [53]. Then the GPU implementation is described in detail. The results show that with real Pacbio data, GPUseed is up to 9x faster than a highly optimized CPU based implementation running with 24 threads.

We also described our software optimized algorithm for computing maximal exact matching seeds used in the BWA-MEM read mapper. Our optimization helps to speed up the FPGA accelerated BWA-MEM from 1.9x to 2.6x.

This chapter consists of the following articles:

- N. Ahmed, K. Bertels, and Z. Al-Ars, *GPUseed: Fast Computation of Maximal Exact Matches for Genome Analysis*, (2019), Submitted to *MDPI Genes*.
- © 2015 IEEE. Reprinted, with permission, from "N. Ahmed, V. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars, *Heterogeneous Hardware/Software Acceleration of the BWA-MEM DNA Alignment Algorithm*, in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2015) pp. 240–246".

Article

GPUseed: Fast computation of Maximal Exact Matches for Genome Analysis

Nauman Ahmed [1,2,*], Koen Bertels [1] and Zaid Al-Ars [1]

[1] Delft University of Technology, Delft, Netherlands
[2] University of Engineering and Technology Lahore, Lahore, Pakistan
* Correspondence: n.ahmed@tudelft.nl

Received: date; Accepted: date; Published: date

Abstract: The seeding heuristic is widely used in many DNA analysis applications to speed up the analysis. In many applications, seeding takes a substantial amount of total execution time. In this paper, we present a CUDA API for computing maximal exact matching seeds on GPU. It can be used to compute maximal exact matching as well as super-maximal exact matching seeds. We applied optimization to reduce the number of GPU global memory accesses and to avoid redundant computation. Our implementation also tries to extract maximum parallelism from the maximal exact match computation tasks. We tested our library using the data from the state-of-the-art third generation Pacbio and Oxford Nanopore DNA sequencers generating hundreds of kilobases long DNA reads. Our results show that it is up to 9x faster for computing maximal exact matching seeds as compared to optimized CPU implementation running on 24 Intel Xeon threads. For computing super-maximal exact matches, our API is up to 5.5x faster. The implementation is open source and can be integrated into any DNA analysis application. The results of our implementation show that GPU provides substantial performance gain for massively parallelizable algorithms even if they are memory-bound.

Keywords: maximal exact matches, GPU, genome analysis

---

1. Introduction

Current DNA analysis programs have to process massive amounts of data. In many applications this data is in the form of *DNA reads*. A DNA string is a single read or an assembly of many reads. It is made up of only four characters: "a", "c", "t" and "g". These four characters represent the four types of nucleotide bases in the DNA and are also known as *base pairs*(bp). Many of the DNA analysis algorithms require to solve an approximate string matching problem. Therefore if the number of the DNA strings to be matched is large or the strings are long, the direct application of Smith-Waterman [1] or similar algorithms is too slow for practical purposes. To overcome this problem, BLAST [2] pioneered the approach of applying a *seeding* heuristic. The reason behind applying the seeding heuristic is the observation that for a good match to be found between two strings, these strings must share a highly matching substring. Therefore, to match two strings, first, a common substring should be found. This common substring is known as a *seed*. An approximate match between the strings is then found by applying Smith-Waterman around the seed. This two-step method is known as seed-and-extend.

Computing a seed refers to finding the pattern of the common substring as well as its location in the two strings. In a typical DNA analysis application, millions of independent DNA strings have to be processed, each of which may contain many seeds. Therefore, computing seeds is a highly parallel problem, that can be accelerated on massively parallel Graphics Processing Unit (GPU) devices.

Maximal exact match seeds are computed in a variety of bioinformatics applications which include: BWA-MEM [3] and CUSHAW2 [4] DNA read mappers; Jabba [5], a DNA sequencing error correction tool; and MUMmer [6] a whole genome alignment application. Therefore, optimizing or accelerating the computation of the maximal exact matches could be beneficial for a large number of DNA analysis programs. Some seeding algorithms, like slaMEM [7], extend the FM-index for faster computation of MEMs, while others, like essaMEM [8], propose other types of indexes to speed up the computation. CUSHAW2-GPU [9] is the GPU implementation of the CUSHAW2 read mapper. It computes the MEMs on the GPU. But it is designed only for short reads with a maximum allowed read length of only 320 bases. Moreover, it stores the suffix array intervals of the MEMs in an intermediate file which is subsequently loaded for locating the MEM seeds on the reference genome. This makes the computation extremely slow. In [10] the authors present the computation of exact matches using suffix trees. Suffix trees are large data structures that consume a huge amount of memory. The suffix tree used in the paper would take 152 gigabytes of memory for the human reference genome. Hence, the approach is impractical for large genomes. GPUmem [11] is a GPU implementation for computing maximal exact matches between only two very long DNA sequences, e.g. between chromosome 1 and chromosome 2 of the human genome.

NVBIO [12] is an open source library developed by NVIDIA. It contains the functions for computing the maximal exact matching seeds on GPU using the FM-index. But it can only be used to find maximal exact matching seeds for short DNA reads. For long DNA reads, NVBIO terminates due to illegal memory access error. This may be due to the sizes of some data structures that are not sufficient for long DNA reads. Moreover, NVBIO uses bidirectional BWT of [13] which limits the amount of parallelism (See Section 2.2.3, first paragraph). This is especially true for long DNA reads since less number of long DNA reads can be loaded in the GPU memory as compared to short DNA reads for MEM computation.

Maximal exact matching seeds are widely used for DNA analysis but in the past, there were limited efforts to parallelize their computation. State of the art third generation sequencing platforms produce long DNA reads with lengths up to hundreds of kilobases. In this paper, we present *GPUseed*, a CUDA API to accelerate the computation of various *maximal exact matching* seeds on GPUs for DNA analysis in long DNA sequences. The contributions of the paper are as follows:

- We present the fastest GPU implementation in the form of an API for computing the maximal exact matches in long DNA sequences.
- We present unique optimizations that include pre-calculated suffix array intervals and early detection of redundant MEMs for fast computation of maximal exact matches on GPU.
- Experiments show that our implementation provides 7-9x speedup over the fastest CPU implementation for the third generation Pacbio DNA reads.

This paper is organized as follows. Section 2 discusses the required background and the implementation details of GPUseed. Section 3 describes the experimental results. Section 4 presents the conclusions drawn from the paper.

2. Materials and Methods

2.1. Background

As described in the introduction, seeding is used to speed up the process of finding an approximate match between two DNA strings. More formally, let one string be called as text T and the other be called pattern P.

In practice, T is one long DNA string assembled from many DNA reads. To find an approximate match between T and P, we first need to find substrings of P that are exactly matching at one or more locations in T. A *maximal exact match* (MEM) between two strings is an exact match that cannot be further extended in either direction without incurring a mismatch. The concept of maximal exact matches was first proposed in [6]. A MEM is said to be an *SMEM* (super-maximal exact match) if it is not contained in any other MEM between T and P [14].

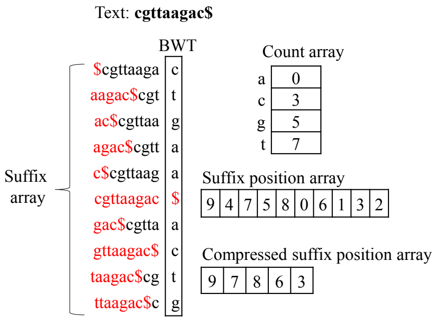


Figure 1. The FM-index. Strings in red color form a list of suffixes of text sorted in lexicographically order.

2.1.1. The FM-index

The algorithm for computing the seeds depends upon the underlying *index*. The index is a pre-built data structure that is used to compute seeds. For example, to find substrings of pattern P that exactly match in text T, an index of T is built. In DNA analysis applications, mainly two types of indexes are used: i) Hash tables ii) Indexes based on suffix/prefix trie.

Hash table indexes are fast and require $\mathcal{O}(1)$ time to compute a seed. The disadvantage of using hash tables is that they can only be used to find fixed-length seeds. Moreover, the length of the seed should be kept low (e.g., below 15 [15]). For longer seeds, it requires a huge amount of RAM. For example, the SNAP read mapper hash table requires 39 gigabytes of RAM for seeds of length 20 [16].

There are various types of indexes based on the suffix/prefix trie index type, such as suffix arrays [17], enhanced suffix arrays, [18], and FM-index [19]. *FM-index* is widely used in many DNA analysis applications due to its speed and small memory footprint. In our implementation, we have used the seed searching algorithm based on the FM-index

The FM-index is a data structure based on the Burrows-Wheelers transform [20] of the text. In Figure 1, the FM-index is a set of three arrays: i) Count array C ii) Burrows-Wheeler transform array BWT and iii) Suffix position array SP. The C array has only four elements, one for each DNA base. The C array element for a base x holds the number of DNA bases in the text T that is lexicographically smaller than x. The BWT array holds the Burrows-Wheeler transform of T. The Burrows-Wheeler transform of T is computed by lexicographically sorting all the rotations of T and then storing the last column. The SP holds the starting position of the suffixes of T. For a very large T, storing the starting position of all the suffixes will require a huge amount of RAM. For example, if T is the human reference genome, the number of suffixes are over 3 billion (length of the reference genome). 4 bytes are required to store each position. Hence, over 12 gigabytes of memory is required to store the full suffix array. Usually, to save RAM, SP is stored in compressed form and the starting position of only those suffixes is stored for which suffix array index is a multiple of a certain number, known as *compression ratio* in this paper. For example, the compression ratio in Figure 1 is 2.

2.1.2. Computing seeds using FM-index

As described before, computing a seed means that we attempt to find a common substring between T and P and find its location in T and P. Assume $P[i:j]$, a substring of P. i and j are the starting and ending positions of $P[i:j]$ in P. $|.|$ denotes the length of a string. Seed computation using the FM-index is completed in two steps: a) Computing suffix array intervals of the seed, and b)

Algorithm 1: Computing Suffix array intervals of all the MEMs between P and T

```
Input: Pattern $P$ and minimum required MEM length $min\_mem\_len$
Output: Array $M$ containing all the MEMs in $P$
Function MemSAInterval($P, min\_mem\_len$) begin
    Initialize M as an empty array
    $[l, u] \leftarrow [0, |T| - 1]$
    for $j \leftarrow |P| - 1$ down to $min\_mem\_len - 1$ do
        $q \leftarrow j$
        while $q >= 0$ do
            $prev\_l \leftarrow l$
            $prev\_u \leftarrow u$
            $l \leftarrow C[P[q]] + Occ(P[q], l - 1) + 1$
            $u \leftarrow C[P[q]] + Occ(P[q], u)$
            if $l > u$ then
                break
            $q \leftarrow q - 1$
        // $q = -1$
        if $l \leq u$ and $j - (q + 1) + 1 \geq min\_mem\_len$ then
            $start \leftarrow q + 1$
            $end \leftarrow j$
            Append $([l, u], start, end)$ to $M$
        // otherwise
        else if $j - q + 1 \geq min\_mem\_len$ then
            $start \leftarrow q + 1$
            $end \leftarrow j$
            Append $([prev\_l, prev\_u], start, end)$ to $M$
```

Locating the seed in T using suffix array intervals. Suffix array interval $[l(P[i:j]), u(P[i:j])]$ of $P[i:j]$ is defined as:

$$\begin{aligned} l(P[i:j]) &= \min\{k : P[i:j] \text{ is the prefix of } SA[k]\} \\ u(P[i:j]) &= \max\{k : P[i:j] \text{ is the prefix of } SA[k]\} \end{aligned}$$

where suffix array SA contains the suffixes of the text T sorted in lexicographical order, as shown in Figure 1. Suffix array interval of $P[i:j]$ can be computed using FM-index with *backward search*. In backward search we start with an empty string. The l and u of the empty string are defined as 0 and $|T| - 1$, respectively. We then add bases to the empty string from the end of $P[i:j]$, one base at a time, in the backward direction so that the string grows as $P[j:j] \rightarrow P[j-1:j] \rightarrow P[j-2, j] \cdots \rightarrow P[i:j]$. After adding every base, the suffix array interval of the new string is calculated as:

$$l(P[x:j]) = C[P[x]] + Occ(P[x], l(P[x+1:j]) - 1) + 1 \quad (1)$$

$$u(P[x:j]) = C[P[x]] + Occ(P[x], u(P[x+1:j])) \quad (2)$$

where $Occ(b, y)$ is the number of occurrences of base b in the BWT array from 0 to y. If $l(P[x:j]) \leq u(P[x:j])$, then $P[x:j]$ does exist in T and occurs at $u(P[x:j]) - l(P[x:j] + 1$ locations in T.

Algorithm 1 uses the recurrence Equations 1 and 2 to compute the suffix array intervals of all the MEMs between P and T. The algorithm returns an array M containing the MEMs in the form of tuples $([l, u], start, end)$, where $[l, u]$ is the suffix array interval of the MEM; $start$ is the starting position of the MEM in P and end is the ending position of the MEM. MEM computation starts from the base at index q and builds the MEM string in the backward direction by adding bases from P until the base at the $0th$ index of P is reached. To compute SMEMs, assume that the MEMs in the array M are sorted in ascending order with respect to the $start$. Since SMEMs are a subset of MEMs, all those MEMs M_i in the array M which satisfy the condition $start[i] == start[i+1], i = 0 \ldots |M| - 2$ are filtered out and the remaining MEMs in the array M are the SMEMs. $|M|$ is the number of elements in the array M

Locating a seed refers to computing the start position of a seed in T. Algorithm 2 shows how the FM-index is used to compute the starting position of seeds with compressed suffix position

Algorithm 2: Computing the starting position for a given suffix array index

```
Input: suffix array index of the seed $sa\_idx$
Output: starting position of the seed in the text
Function LOCATESEED($sa\_idx$) begin
    $itr \leftarrow 0$
    $i \leftarrow sa\_idx$
    while $i \% r \neq 0$ do
        $i \leftarrow C[BWT[i]] + Occ(BWT[i], i-1)$
        $itr \leftarrow itr + 1$
    return $SP[i] + itr$
```

array SP having a compression ratio of r. The suffix array interval of a seed contains $u - l + 1$ suffix array indexes. The `LOCATESEED` function accepts a suffix array index (sa_idx) of the seed and computes the corresponding seed starting position in T. Hence, the `LOCATESEED` function is called for $sa\_idx = l, l+1 \ldots, u-1, u$ to find all the locations of the seed in T.

Computing MEM seeds is a highly parallel process. The seeds for all the DNA reads can be computed in parallel. Even for a single read the iterations of the `for` loop in Algorithm 1 are independent. Moreover, each MEM occurs at $l - u + 1$ places in T and all these locations can be calculated in parallel using Algorithm 2. Hence, the computation of maximal exact matching seeds is well suited for parallel processing.

2.1.3. Graphics processing units

In heterogeneous computing era, Graphics Processing Units (GPUs) are used as accelerators for high performance applications due to there massively parallel architecture. In the following, we will briefly describe the architecture and programming of NVIDIA GPUs

In NVIDIA GPUs, there are numerous streaming multiprocessors (SMs). Each SM has many cores known as streaming processors (SPs). An SP is the basic computational unit that executes a GPU thread. GPU also has its own DRAM known as *global memory*. GPU threads are grouped into *grids*. Each grid contains many GPU thread *blocks*. The threads in a block are assigned to the same SM. The number of blocks within a grid and the number of GPU threads in a block are configured by the programmer. All the GPU threads can communicate with each other via global memory. However, the threads in a block can also communicate through a fast *shared memory*. In a block, there is a set of 32 threads known as a *warp* that share the program counter. The threads in a warp can communicate with the help of *shuffle* instructions.

The programming language for NVIDIA GPUs is known as *CUDA* which is an extension of C/C++. The GPU programmer writes a *kernel* which executes on the GPU. The GPU is attached to a *host* CPU which initiates all the tasks related to the GPU. The data to be processed by the kernel is copied from the host memory to the global memory of the GPU. The CPU then launches the kernel. Once the kernel is finished, the results are copied from the global memory back into the host CPU memory.

2.2. GPUseed Implementation

GPUseed exploits the massive parallelism in seed computation. In many situations, there is a large number of patterns that need to be matched with the text T. Moreover, several seeds for each pattern P need to be computed. Therefore, in total, a large number of seeds can be computed in parallel. GPUseed contains several stages for computing (S)MEM seeds. Figure 4 shows the different computational stages of GPUseed. The array shown after each stage is the output of the stage. Each stage launches one or more GPU kernels. The API is capable of computing MEMs and SMEMs not only on DNA text T but also on its reverse complement $\overline{T}$. To simplify the discussion, we focus on the case when the (S)MEMs are to be computed only on T.

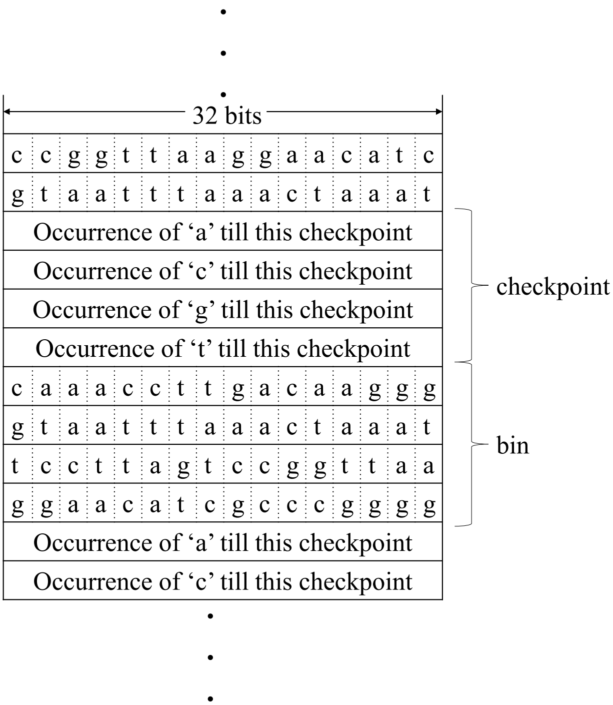

32 bits
c c g g t t a a g g a a c a t c
g t a a t t t a a a c t a a a t
Occurrence of ‘a’ till this checkpoint
Occurrence of ‘c’ till this checkpoint
Occurrence of ‘g’ till this checkpoint
Occurrence of ‘t’ till this checkpoint
checkpoint
c a a a c c t t g a c a a g g g
g t a a t t t a a a c t a a a t
t c c t t a g t c c g g t t a a
g g a a c a t c g c c c g g g g
bin
Occurrence of ‘a’ till this checkpoint
Occurrence of ‘c’ till this checkpoint

Figure 2. The *BWT* array used in GPUseed. The BWT bases shown here are arbitrary

2.2.1. The index

To compute the seeds using GPUseed, a pre-built index is loaded in the GPU memory. The index consists of *BWT* array, count array *C*, compressed suffix position array *SP* and a pre-calculated suffix interval array. The use of the pre-calculated suffix interval array is described in Section 2.2.3. As described in Section 2.1.2 the seed search algorithm using FM-index needs to compute $Occ(b, y)$, which is the number of occurrences of base b in the *BWT* array from 0 to y. In the case of a large BWT array, counting the number of occurrences becomes very slow. Therefore, in practice, the *BWT* array is interleaved with checkpoints that contain the number of occurrences of all the four types of bases till that checkpoint. Figure 2 shows the *BWT* array used in GPUseed. To find $Occ(b, y)$, first, the bin containing the index y is found. Then the bases of type ′b′ in the bin till index ′y′ are counted and the count is added to the preceding checkpoint value of the base ′b′. The checkpoints occur after a fixed number of BWT bases. This number is known as *bin_size*. In GPUseed, each *BWT* array entry is a 32-bit integer. Since each base is encoded in 2 bits, the *bin_size* must be a multiple of 16. We found that $bin\_size = 64$ is the most suitable choice as it allows to load the checkpoint and the bin using only 2 `uint4` CUDA vector load instructions. CUDA *popcount* instructions are used to count the number of bases in a bin. We also tested other bin sizes. Smaller bin sizes help to decrease the number of popcount instructions but cannot decrease the number of load instructions as the minimum bin size is 16. Moreover, smaller bin sizes can increase the number of memory accesses as described in the first optimization of Section 2.2.3.

2.2.2. Stage-0: Pre-processing

GPUseed accepts input data from the user and returns the results. The user passes patterns for which the seeds are to be computed on the GPU along with the length of each pattern. The user also

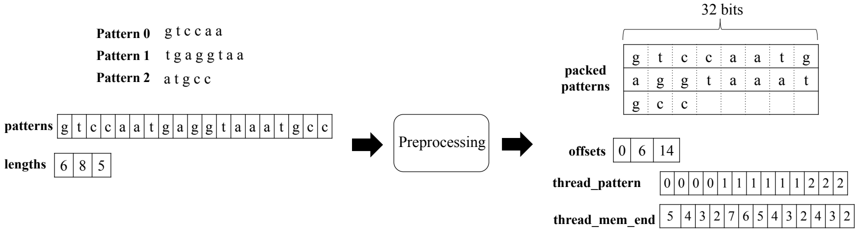

Pattern 0 g t c c a a
Pattern 1 t g a g g t a a
Pattern 2 a t g c c
patterns g t c c a a t g a g g t a a a t g c c
lengths 6 8 5
Preprocessing
32 bits
packed patterns
g t c c a a t g
a g g t a a a t
g c c
offsets 0 6 14
thread_pattern 0 0 0 0 1 1 1 1 1 1 2 2 2
thread_mem_end 5 4 3 2 7 6 5 4 3 2 4 3 2

Figure 3. An example of the input to GPUseed and the result of prepossessing for (S)MEM computation on GPU. Minimum required seed length is 3

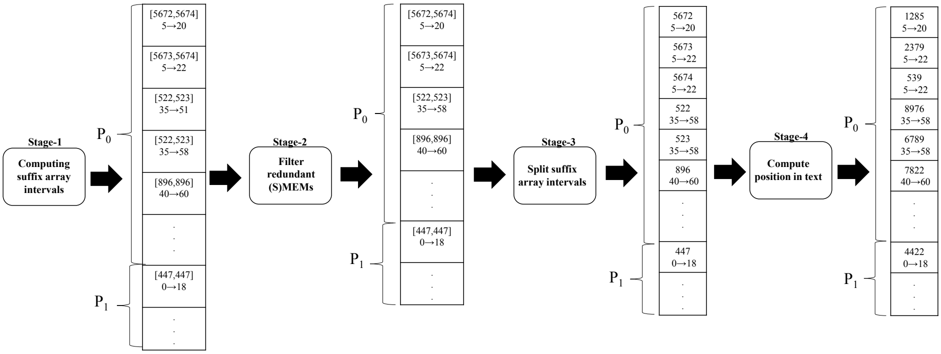


Figure 4. Different computational stages in the GPUseed

passes the pointers to the FM-index arrays. GPUseed also contains the functions for constructing an FM-index for a given text T. These functions are extracted from BWA read mapper version 0.5.9 [21]. The index is copied to the GPU global memory. In the rest of the paper, we will assume that a pre-built FM-index exists in the GPU global memory. GPUseed pre-processes the input to generate 4 arrays. Figure 3 shows an example of the input (*pattern* and *lengths* array) and the result of prepossessing for (S)MEM seed computation on GPU: i) A 32 bit integer array which contains *packed patterns*. The bases of the input patterns are converted from ASCII to 4-bit representation. 4-bit representation is required to accommodate the fifth type of base ’n’, known as an ambiguous base. DNA sequencers assign value ’n’ to those bases which it finds difficult to identify. 4 bits instead of 3 bits are chosen for the ease of post-processing. 3-bit representation allows to pack 10 bases in an integer instead of 8 and hence provide a significant advantage over 4-bit representation. The packing is performed on the GPU using the same method as described in [22]. ii) *offsets*: an integer array containing the starting index of the patterns in the input. This array is computed on the GPU by performing a prefix sum of the input *lengths* array. These offset values are used to calculate the starting position of a pattern in the *packed patterns* array. iii) *thread_pattern*: an integer array that assigns a pattern to a thread. iv) *thread_mem_end*: this array assigns a (S)MEM within a pattern to a thread. The values in this array specify the index of the pattern at which the (S)MEM is going to end. Since GPUseed uses the backward search algorithm, the thread starts from the base at the assigned pattern index and extends it in the backward direction. The maximum number of (S)MEMs in a pattern equals $|P| - min\_mem\_len + 1$. There is a large variation in the length of the long DNA reads produced by the third generation DNA sequencers. Therefore, The computation of *thread_pattern* and *thread_mem_end* arrays is essential for efficient utilization of the GPU resources. Both the *thread_pattern* and *thread_mem_end* are also computed on the GPU. Since all the output arrays of the pre-processing stage are computed on GPU,

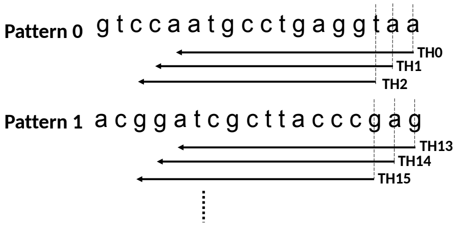

Pattern 0
g t c c a a t g c c t g a g g t a a
TH0
TH1
TH2
Pattern 1
a c g g a t c g c t t a c c c g a g
TH13
TH14
TH15

Figure 5. An example of GPU thread assignment for (S)MEM seed computation on GPU. Minimum required seed length is 6

the overhead of the pre-processing is negligible. We found that the total time spent in prepossessing is less than 1% of the total execution time of GPUseed in all the experiments.

2.2.3. Stage-1: Finding suffix array intervals

This first stage computes the suffix array intervals of MEMs. To compute SMEMs, some MEMs are filtered out the second stage as described in the next section (Section 2.2.4).

The software optimized algorithms for computing MEMs are different from the one shown in Algorithm 1. For example, the bidirectional-BWT proposed in [13] allows adding bases both in the forward and backward direction. Also in [14] and in BWA-MEM [3] a bidirectional FM-index (called the FMD-index) is implemented. In such algorithms, the MEMs covering an index v of pattern P are computed by first adding the bases in the forward direction and after adding every base the corresponding suffix array interval is stored. Hence, the stored suffix array intervals are for the strings: $P[v]$, $P[v]P[v+1]$, $P[v]P[v+1]P[v+2], \ldots$. Then each of these strings is extended in the backward direction to compute all the MEMs covering base at index v. The bases added in the forward direction are common to all the MEMs and hence added only once. This reduces the accesses to the FM-index. One disadvantage of such an approach is that the forward addition of bases has to be sequential and cannot be parallelized. Therefore, in our GPU implementation, we will use Algorithm 1 as it allows all the MEMs in pattern to be computed in parallel, exploiting the massive parallelism of GPUs.

The main GPU kernel in this stage is used to compute the suffix array intervals of MEMs. One MEM is assigned to a GPU thread. Figure 5 shows an example of the GPU thread assignment. THx are the threads, where x is the thread number. Each GPU thread starts from a different base of the pattern as shown in Figure 5 and then extends it in the backward direction. Algorithm 3 shows the GPU kernel used in GPUseed for computing the suffix array intervals of MEMs. The algorithm is similar to Algorithm 1 with each iteration of the `for` loop being computed by a separate GPU thread. We further applied two optimizations which are explained below:

First optimization – Pre-calculated suffix array intervals: As a part of the index building, we pre-calculate the suffix array intervals of all the possible $4^{PRE\_CALC\_LEN}$ sequences of length PRE_CALC_LEN. Therefore, the suffix array interval of the last PRE_CALC_LEN bases of the MEM is already known. The GPU thread first loads the pre-calculated suffix array intervals of the last PRE_CALC_LEN and then extends the MEM in the backward direction. Using pre-calculated suffix array intervals has two advantages

1. The pre-calculated suffix intervals reduce the number of backward search steps required for the computation of the suffix array interval of a MEM.
2. As shown in Algorithm 1 the calculation of a suffix array interval $[l, u]$ requires two accesses to the BWT array to compute the Occ. At the start of the backward search these accesses are in different bins of the BWT array, but as the backward search proceeds the difference between l and u decreases [23] and at some point may become less than the bin size. From here only one BWT array access is required to compute the suffix array interval. Using the pre-calculated suffix array intervals may allow us to skip that initial phase of two BWT array accesses.

Algorithm 3: Computation of suffix array intervals of (S)MEMs in GPUseed

Input: The Pattern P_{tid} assigned to the thread with thread ID *tid* in *thread_pattern* array; The index *j* of the pattern assigned to the thread in *thread_mem_end* array; the suffix array interval $[l_{precalc}, u_{precalc}]$ of the last *PRE_CALC_LEN* bases of the MEM and minimum required MEM length *min_mem_len*

Output: Array *M* containing all the MEMs in *P*

```
1  Function GPUMEMSAInterval($P_{tid}, j, [l_{precalc}, u_{precalc}]$,min_mem_len) begin
2      if $u_{precalc} \geq l_{precalc}$ then
3          Initialize prev_intv_size array having length of WARP_SIZE with 0's
4          $[l, u] \leftarrow [l_{init}, u_{init}]$
5          $q \leftarrow j - PRE\_CALC\_LEN$
6          $curr\_intv\_size_{tid} \leftarrow u_{init} - l_{init} + 1$
7          $redundant \leftarrow 0$
8          while $q >= 0$ do
9              $prev\_l \leftarrow l$
10             $prev\_u \leftarrow u$
11             $l \leftarrow C[P_{t}id[q]] + Occ(P_{t}id[q], l - 1) + 1$
12             $u \leftarrow C[P_{t}id[q]] + Occ(P_{t}id[q], u)$
13             if $l > u$ then
14                 break
15             for $i \leftarrow 1$ up to $tid\%WARP\_SIZE$ do
16                 if thread tid - i is active and $prev\_intv\_size[i] = curr\_intv\_size_{tid-i}$ and $P_{tid} = P_{tid-i}$ then
17                     $redundant \leftarrow 1$
18                     break
19             for $i \leftarrow WARP\_SIZE - 1$ down to 1 do
20                 $prev\_intv\_size[i] \leftarrow prev\_intv\_size[i - 1]$
21             $prev\_intv\_size[0] \leftarrow curr\_intv\_size_{tid}$
22             $curr\_intv\_size_{tid} \leftarrow u - l + 1$
23             $q \leftarrow q - 1$
24         if $redundant = 0$ then
25             // $q = -1$
26             if $l \leq u$ and $j - (q + 1) + 1 \geq min\_mem\_len$ then
27                 return $([l, u], q + 1, j)$
28             // otherwise
29             else if $j - (q + 1) \geq min\_mem\_len$ then
30                 return $([prev\_l, prev\_u], q + 1, j)$
31             else
32                 // return NULL
33                 return $\emptyset$
34         else
35             // return NULL
36             return $\emptyset$
37     else
38         // return NULL
39         return $\emptyset$
```

We found that $PRE\_CALC\_LEN = 13$, requiring 512 megabytes of GPU memory, provides a good speed-memory tradeoff. In case the minimum required MEM length is less than 13, the value of PRE_CALC_LEN is reduced.

As described previously, the MEM finding algorithm optimized for CPUs use the bidirectional index and the algorithm is different for the one used in our GPU implementation (Algorithm 1). The downside of using such an index is that the optimization of pre-calculated suffix array intervals cannot be applied. Hence, this optimization is unique in the sense that the MEM finding algorithm of our GPU implementation allows pre-calculated suffix array intervals to speed up the MEM computation.

Second optimization – Early detection of redundant MEMs: Two overlapping MEMs may have the same suffix intervals, and hence the smaller one is redundant. For example in Figure 5, $TH1$ and $TH22$ may have the same suffix intervals at the same index of Pattern 0 and will be backward extended till the same base of the pattern. In this case, the MEM found by $TH2$ is redundant. In GPUseed, the kernel to find the suffix array intervals of the MEMs tries to detect such redundant MEMs as shown from Line 15 to Line 22 in Algorithm 3. $WARP\_SIZE = 32$ in current NVIDIA GPUs. A thread keeps a record of the previously computed suffix array interval sizes $(u - l + 1)$ in the *prev_intv_size* array during the backward search. A MEM computed by a GPU thread is redundant if a value in its *prev_intv_size* array is the same as the current interval size of a thread with lesser ID, which is in the same warp and is also assigned the same pattern. If the MEM assigned to GPU is found to be redundant by the early detection mechanism, the thread exits returning NULL. To access the values of the current interval size of the other threads, we use CUDA *warp shuffle* instruction, which allows the rapid exchange of variables between threads in the same warp without involving shared memory.

Finally, each thread writes the suffix array interval and start and end position of the MEMs in the output array as shown in Figure 4. Each entry in the output array of Stage-1 contains two values. The top value is the suffix array interval and the bottom value is $start \rightarrow end$, where $start$ and end are the starting and ending positions of the MEM in the pattern, respectively. Some entries will be NULL. Note that the values in the output array are in the ascending order with respect to the $start$. The output of Stage-1 contains example values of suffix array intervals and $start \rightarrow end$

2.2.4. Stage-2: Filtering redundant MEMs

A warp contains only $WARP\_SIZE$ number of threads and a warp shuffle instruction only exchanges variables within a warp. Therefore, the output of Stage-1 still contains some redundant MEMs. Moreover, in the case of SMEM computation, we also need to filter out some non-redundant MEMs. The GPU kernel of Stage-2 is used for this purpose. First, the NULL entries in the output array of Stage-1 are eliminated. We used the `DeviceSelect` kernel from *CUB* CUDA library [24] to eliminate these NULL entries and compact the output array of Stage-1. After compaction, each thread is assigned one entry of the output array of Stage-1. The thread applies a test to know whether to filter out this entry or not. The condition for filtering out an entry slightly differs for MEM and SMEM computation. For MEM the condition is:

```
if (start[i] = start[i−1]
    and u[i] − l[i] + 1 = u[i−1] − l[i−1] + 1)
{
    /*filter out M[i]*/
   M[i] = NULL
}
```

where M is the array of MEMs. For SMEMs the second condition i.e. `u[i] - l[i] + 1 = u[i-1] - l[i-1] + 1`, is not required. Since each thread has to work on only one entry, the filtering stage is very fast. The output of this filtering kernel is once again compacted to remove the filtered out (S)MEMs using CUB `DeviceSelect` kernel. The output of filtering stage (Stage-2) after compaction is shown in Figure 4. It has the same format as for Stage-1. Note that the third value in the output array of Stage-1 is filtered out. The first value in the array should have also been filtered out if it is an SMEM computation.

2.2.5. Stage-3: Splitting suffix array intervals

This GPU kernel splits up the suffix interval to its constituent suffix array indexes. The splitting is done so that each GPU thread in the locate kernel (Section 2.2.6) computes only one position in the DNA text corresponding to the suffix array index assigned to it. In the case of two overlapping MEMs with the same starting position, the suffix interval of the longer MEM is the subset of the shorter MEM. Therefore, the splitting up performed by this stage makes sure that a suffix array index does not appear twice and, hence two GPU threads in the locate kernel are never assigned the same suffix array index. This will reduce the number of locations to be computed by the locate kernel (Section 2.2.6). The splitting process is very fast as each GPU thread is assigned only one suffix interval to split. This stage takes less than 1% of the total execution time in all the experiments. The output after this stage is shown in Figure 4. Each entry has two values. The top value is the suffix array index, while the bottom value is $start \rightarrow end$. Note that the suffix array index 5673 is present in both the first and second entries of the output of Stage-2, but it is not repeated in the output of Stage-3

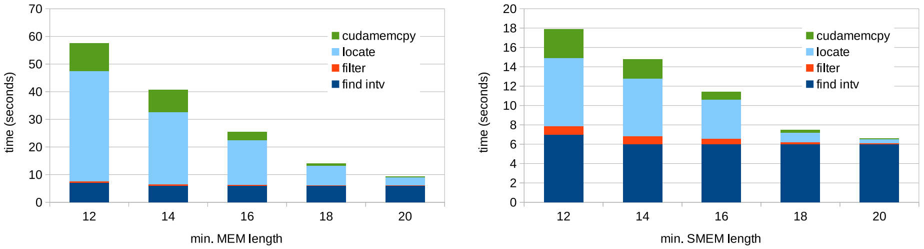


Figure 6. Time spent in different stages for computing (S)MEMs on GPUseed for Pacbio reads with different required minimum seed lengths.

2.2.6. Stage-4: Locating (S)MEMs in text

This is the final stage and generates the locations of the (S)MEM seeds in the DNA text. Each GPU thread has to compute only one location using Algorithm 2. Figure 4 shows the example values of the seed locations in the text.

2.2.7. Capabilities of the API

The CUDA API is suitable for computing both MEMs and SMEMs. The API can also be used in multithreaded applications with one or multiple GPUs. (S)MEMs for forward as well reverse complement DNA text can be computed. To do so we reverse complement the pattern P to obtain $\overline{P}$ and find the (S)MEMs for $\overline{P}$. To reverse complement a DNA sequence it is first reversed and then base 'a' is replaced with base 't' and vice verse; base 'c' is replaced with base 'g' and vice verse. The packing kernel in the pre-processing stage (Stage-0) can perform the reverse complement operation if required. In case of reverse complement the entries in output arrays of Figure 4 are arranged as $P_0\overline{P_0}P_1\overline{P_1}\ldots$. The entries in $P_i\overline{P_i}$ of final output array are further sorted with respect to start position on the pattern using CUB `DeviceSegmentedSort` kernel. The API can be used with patterns up to a length of 2^{31} bases.

3. Results and discussions

For comparison purposes, we considered the problem of computing (S)MEMs between a set of DNA reads and the human reference genome, UCSC hg19 (GRCh37 Genome Reference Consortium Human Reference 37 (GCA_000001405.1)). Hence, in this case the text T is the reference genome. (S)MEMs on both the T as well as $\overline{T}$ are computed. We used a subset of long DNA reads generated by state-of-the-art third generation DNA sequencing of the whole human genome. Experiments are performed with two datasets: *Pacbio reads* and *Oxford Nanopore reads*. The Pacbio reads are downloaded from [25]. The total number of reads is 25249 with the average read length of 7 kilobases and the longest read is 35 kilobases. The Oxford Nanopore read dataset is downloaded from [26] which contains 53723 reads, but we only used the top 8954 reads to reduce execution time. The average read length in this dataset is 25 kilobases and the longest read is 475 kilobases. The execution time reported in this Section is the time required to find the (S)MEMs for all the reads in the dataset.

We compared our CUDA API against the fastest CPU implementation available to compute (S)MEMs using FM-index. This implementation is available in BWA-MEM [27]. We used version 0.7.13 which has a `fastmap` command to generate (S)MEMs. Originally the command only produces SMEMs. We slightly modified the SMEM function to produce MEMs. Moreover, we extended the `fastmap` command to make it multithreaded using OpenMP. The reads are distributed over CPU threads for computing (S)MEMs. Our extended version of `fastmap` command is available at [28].

Furthermore, we compared the `fastmap` command of BWA-MEM to other MEM seeding tools available publicly to ensure that `fastmap` is the fastest CPU implementation for computing MEM

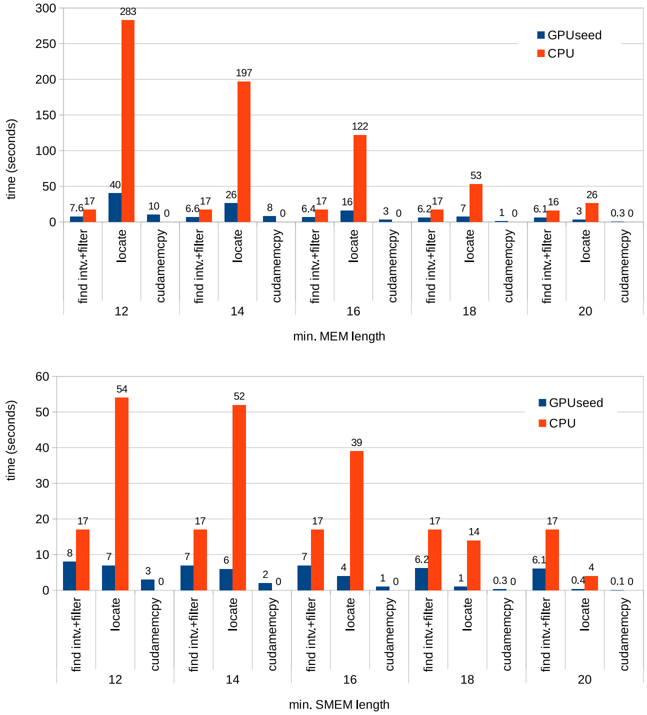


Figure 7. Comparison of time spent in different stages to compute (S)MEMs on CPU and GPUseed for Pacbio reads with different required minimum seed lengths. GPUseed is executed with one CPU thread, whereas the CPU implementation is executed with 24 threads.

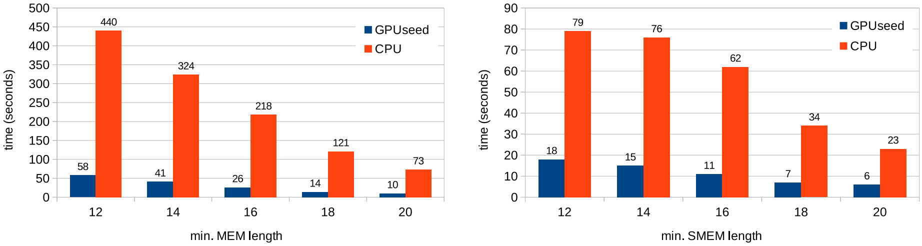


Figure 8. Comparison of total execution time to compute (S)MEMs on CPU and GPUseed for Pacbio reads with different required minimum seed lengths. GPUseed is executed with one CPU thread, whereas the CPU implementation is executed with 24 threads.

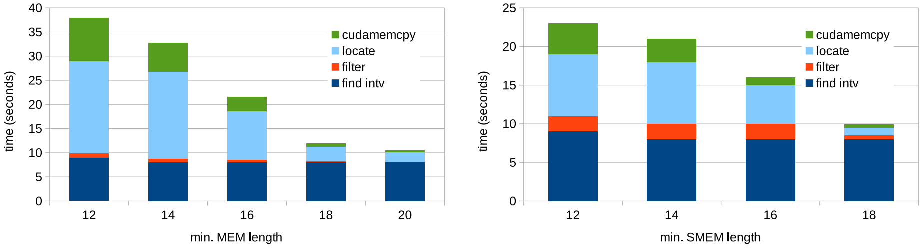


Figure 9. Time spent in different stages for computing (S)MEMs on GPU for Oxford Nanopore reads with different required minimum seed lengths.

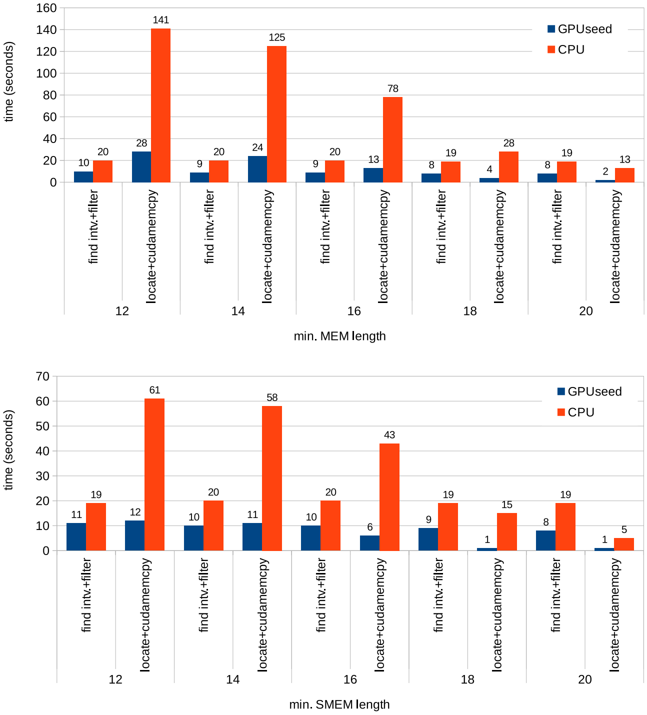


Figure 10. Comparison of time spent in different stages to compute (S)MEMs on CPU and GPU for Oxford Nanopore reads with different required minimum seed lengths. GPUseed is executed with one CPU thread, whereas the CPU implementation is executed with 24 threads.

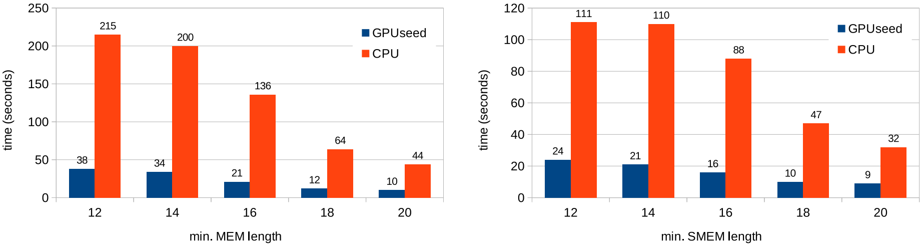


Figure 11. Comparison of total execution time to compute (S)MEMs on CPU and GPU for Oxford Nanopore with different required minimum seed lengths. GPUseed is executed with one CPU thread, whereas the CPU implementation is executed with 24 threads.

seeds. We tested CPU implementation of sparse-MEM [29] and essaMEM [30], both of which support multithreading. With the same index size of 4.8 GB and 24 CPU threads with minimum MEM length of 20, sparseMEM and essaMEM are 8.5x and 6.8x slower than `fastmap`, respectively. The measurements do not include the time required to build the index and I/O time.

Our GPU implementation is executed on NVIDIA Tesla K40c installed in a 2-socket Intel Xeon E5-2620 v3 CPU with two way hyper-threading (12 physical cores, 24 logical cores) and 32 gigabytes of RAM. The CPU implementation is executed on the same machine using 24 threads. We tried different block sizes i.e. the number of GPU threads per block. We found a block size of 128 to be a suitable choice. In all the experiments GPUseed is executed with one CPU thread, whereas the CPU implementation is executed with 24 threads.

The index is the same as described in Section 2.2.1 consisting of a BWT array (1.6 gigabytes) with $bin\_size = 64$, count array (20 bytes), compressed suffix position array (1.8 gigabytes) with compression ratio of 7 and a pre-calculated suffix intervals array with $PRE\_CALC\_LEN = 13$ (512 megabytes). $PRE\_CALC\_LEN = 12$ for minimum MEM length of 12. Hence the total FM-index size is around 4 gigabytes. For the CPU implementation, the FM-index size is 4.8 gigabytes.

3.1. Results for Pacbio reads

Figure 6 shows the time spent in different stages for computing (S)MEMs on the GPU for different required minimum seed lengths. *find intv* is the first stage described in Section 2.2.3. The stage for splitting suffix array intervals and locating the (S)MEMs are shown as single stage *locate* in Figure 6. The time spent in splitting (not shown) is negligible as compared to locating the (S)MEMs using Algorithm 2. *cudamemcpy* represents the time spent in transferring data from CPU to GPU and vice versa. We have not included the time to copy the FM-index and pre-calculated suffix array intervals to the GPU as it is done only once at the beginning of the program. The *cudamemcpy* time is mainly due to the copying of the final output array containing the positions of the (S)MEMs in the text and pattern $start \rightarrow end$ from GPU to CPU. Time spent in copying the pattern (reads) sequences and their lengths and offsets is negligible. The *find intv* stage is same for both MEMs and SMEMs, Therefore, the time for finding intervals is the same (around 6 seconds) for both MEMs and SMEMs. Moreover, the *find intervals* time remains constant irrespective of the minimum required (S)MEM length. The filtration of intervals takes negligible time for all minimum required seed lengths in both MEMs and SMEMs and is around 1% of the total execution time. In the case of MEMs, the *locate* time is small for the bigger minimum required MEM lengths but become dominant for smaller values. For a minimum required MEM length of 12 *locate* time is nearly 6x more than the *find intv* time. This happens because the number of MEMs increase with decreasing minimum required lengths. For the same reason, time spent in memory transfers between CPU and GPU is small for the longer minimum required lengths, but becomes larger than *find intv* for minimum MEM lengths of 14 and 12. In case of SMEMs *locate* time is small for the bigger minimum required lengths but becomes equal to *find intv* time for smaller values.

Figure 7 shows the time spent in different stages of the computation. The CPU implementation has only two stages: finding (S)MEM suffix array intervals (*find intv.* in the Figure 7) and locating the (S)MEMs on the text. There is no separate filtering stage as it is performed during interval computation. Obviously, there is no *cudamemcpy* in CPU implementation. The figure shows that the main reason for the speedup of GPUseed over CPU is due to a much faster *locate* stage on GPU. The *locate* stage of GPUseed is 6x-9x faster than CPU implementation. The *find intv* stage is around 2-3x faster on GPU.

Figure 8 shows a comparison of the total execution time of GPUseed and CPU implementation for (S)MEM computation. Since 24 threads are used to execute the CPU implementation, the time spent in *find intv* and *locate* stage of the CPU implementation (shown in Figure 7) is computed by taking the *average* across the stage time for all threads. Therefore, the sum of the time spent in the CPU stages do not add up to the total CPU execution time. For MEMs the speedup of GPUseed over CPU varies from 7x to 9x. The maximum speedup is achieved around the minimum required MEM lengths of 16-18. For SMEMs the speedup is from 4x to 5.6x with a maximum speedup achieved with minimum seed length of 16.

3.2. Results for Oxford Nanopore reads

The results for Oxford Nanopore reads show the same trend as for Pacbio reads. Figure 9 shows that the *find intv* time remains nearly same for all minimum required (S)MEM lengths (around 8 seconds). The *locate* time is smaller than *find intv* time for longer minimum lengths but becomes larger for smaller minimum lengths. But the difference between *locate* and *find intv* for smaller minimum lengths is not as large as for Pacbio reads. This is because Nanopore reads generate less (S)MEMs as compared to Pacbio reads due to a higher sequencing error rate.

The trend in Figure 10 is same as for Pacbio reads. *find intv* is around 2x faster on GPU and the*locate* stage is around 9x-5x faster.

Figure 11 shows the overall speedup. The overall speedup is 4x-6x and 3.5x-5x for MEMS and SMEMs, respectively. Hence, the speedup is less for Nanopore reads as compared to Pacbio reads due to less number of (S)MEMs.

4. Conclusions

Computation of maximal exact matching seeds is performed in a variety of DNA analysis applications. In this paper, we presented GPUseed, a CUDA API for computing maximal and super-maximal exact matching seeds on GPU. The API computes the maximal exact matching seeds using FM-index. We parallelize the computation on the GPU and extracted maximum parallelism from the computation task. Optimizations were applied to reduce the GPU memory accesses and to reduce redundant computation. Tests were performed using the latest Pacbio and Oxford Nanopore DNA sequencing data containing up to 475 kilobases long reads. Different minimum match lengths were selected. The results showed that our API is up to 9x faster for computing maximal exact matches and up to 5.5x faster for computing super-maximal exact matches as compared to an optimized CPU implementation running on 24 threads of Intel Xeon machine. Computing suffix array intervals is up to 3x faster whereas calculating the location of the match is up to 14x faster. The implementation is open source and can be integrated into any DNA analysis application (https://github.com/nahmedraja/GPUseed).

Author Contributions:

Funding: This research received no external funding.

Acknowledgments:

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Smith, T.; Waterman, M. Identification of common molecular subsequences. *Journal of Molecular Biology* **1981**, *147*, 195 – 197.
2. Altschul, S.F.; Gish, W.; Miller, W.; Myers, E.W.; Lipman, D.J. Basic local alignment search tool. *Journal of Molecular Biology* **1990**, *215*, 403 – 410.
3. Li, H. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv* **2013**.
4. Liu, Y.; Schmidt, B. Long read alignment based on maximal exact match seeds. *Bioinformatics* **2012**, *28*, i318–i324.
5. Miclotte, G.; Heydari, M.; Demeester, P.; Rombauts, S.; Van de Peer, Y.; Audenaert, P.; Fostier, J. Jabba: hybrid error correction for long sequencing reads. *Algorithms for Molecular Biology* **2016**, *11*, 10.
6. Delcher, A.L.; Kasif, S.; Fleischmann, R.D.; Peterson, J.; White, O.; Salzberg, S.L. Alignment of whole genomes. *Nucleic Acids Research* **1999**, *27*, 2369–2376. doi:10.1093/nar/27.11.2369.
7. Fernandes, F.; Freitas, A.T. slaMEM: efficient retrieval of maximal exact matches using a sampled LCP array. *Bioinformatics* **2014**, *30*, 464–471.
8. Vyverman, M.; De Baets, B.; Fack, V.; Dawyndt, P. essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics* **2013**, *29*, 802–804.
9. Liu, Y.; Schmidt, B. CUSHAW2-GPU: Empowering Faster Gapped Short-Read Alignment Using GPU Computing. *Design Test, IEEE* **2014**, *31*, 31–39.
10. Schatz, M.C.; Trapnell, C. Fast Exact String Matching on the GPU. Technical report, University of Maryland, 2007.
11. Abu-Doleh, A.; Kaya, K.; Abouelhoda, M.; Çatalyürek, V. Extracting Maximal Exact Matches on GPU. 2014 IEEE International Parallel Distributed Processing Symposium Workshops, 2014, pp. 1417–1426.
12. Pantaleoni, J.; Subtil, N. NVBIO. nvlabs.github.io/nvbio. Accessed 1 October, 2017.
13. Lam, T.W.; others. High Throughput Short Read Alignment via Bi-directional BWT. IEEE BIBM 2009, 2009, pp. 31–36.
14. Li, H. Exploring Single-sample SNP and INDEL Calling with Whole-genome De Novo Assembly. *Bioinformatics* **2012**, *28*, 1838–1844.
15. Li, H.; Homer, N. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics* **2010**, *11*, 473–483.
16. Zaharia, M.; others. Faster and More Accurate Sequence Alignment with SNAP. *arXiv [q-bio.GN]* **2011**.
17. Manber, U.; Myers, G. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing* **1993**, *22*, 935–948.
18. Abouelhoda, M.I.; Kurtz, S.; Ohlebusch, E. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms* **2004**, *2*, 53 – 86.
19. Ferragina, P.; Manzini, G. Opportunistic data structures with applications. Proceedings 41st Annual Symposium on Foundations of Computer Science, 2000, pp. 390–398.
20. Burrows, M.; Wheeler, D.J. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.
21. Li, Heng. Burrows Wheeler Aligner version 0.5.9. github.com/lh3/bwa/releases/tag/bwa-0.5.9. Accessed 1 January, 2019.
22. Ahmed, N.; Mushtaq, H.; Bertels, K.; Al-Ars, Z. GPU accelerated API for alignment of genomics sequencing data. 2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), 2017, pp. 510–515.
23. Zhang, J.; Lin, H.; Balaji, P.; Feng, W. Optimizing Burrows-Wheeler Transform-Based Sequence Alignment on Multicore Architectures. 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, 2013, pp. 377–384.
24. Merrill, D. CUB - a configurable C++ template library of high-performance CUDA primitives. nvlabs.github.io/cub. Accessed 2 January, 2019.
25. Pacbio. datasets.pacb.com/2013/Human10x/READS/2530572/0001/Analysis_Results/m130929_024849_42213_c100518541910000001823079209281311_s1_p0.1.subreads.fasta, 2014.
26. Nanopore. s3.amazonaws.com/nanopore-human-wgs/rel4-nanopore-wgs-17958431-FAF13748.fastq.gz, 2018.
27. Li, H. Burrows-Wheeler Aligner. github.com/lh3/bwa. Accessed 2 January, 2019.

28. Ahmed, N. GASE - Generic Aligner for Seed-and-Extend. github.com/nahmedraja/GASE. Accessed 2 January, 2019.
29. Khan, Z. sparseMEM. github.com/zia1138/sparseMEM. Accessed 2 January, 2019.
30. Vyverman, M. essaMEM. github.com/readmapping/essaMEM. Accessed 2 January, 2019.

Heterogeneous Hardware/Software Acceleration of the BWA-MEM DNA Alignment Algorithm

Nauman Ahmed*, Vlad-Mihai Sima†, Ernst Houtgast†, Koen Bertels* and Zaid Al-Ars*

*Computer Engineering Lab, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
†Bluebee, Molengraaffsingel 12–14, 2629 JD Delft, The Netherlands
E-mail: *{n.ahmed, k.l.m.bertels, z.al-ars}@tudelft.nl, †{vlad.sima, ernst.houtgast}@bluebee.com

Abstract—**The fast decrease in cost of DNA sequencing has resulted in an enormous growth in available genome data, and hence led to an increasing demand for fast DNA analysis algorithms used for diagnostics of genetic disorders, such as cancer. One of the most computationally intensive steps in the analysis is represented by the DNA read alignment. In this paper, we present an accelerated version of BWA-MEM, one of the most popular read alignment algorithms, by implementing a heterogeneous hardware/software optimized version on the Convey HC2ex platform. A challenging factor of the BWA-MEM algorithm is the fact that it consists of not one, but three computationally intensive kernels: SMEM generation, suffix array lookup and local Smith-Waterman. Obtaining substantial speedup is hence contingent on accelerating all of these three kernels at once. The paper shows an architecture containing two hardware-accelerated kernels and one kernel optimized in software. The two hardware kernels of suffix array lookup and local Smith-Waterman are able to reach speedups of 2.8x and 5.7x, respectively. The software optimization of the SMEM generation kernel is able to achieve a speedup of 1.7x. This enables a total application acceleration of 2.6x compared to the original software version.**

I. INTRODUCTION

With the emergence of low cost high throughput next-generation DNA sequencing methods, it is now possible to diagnose many genetic disorders, e.g. cancer, with very high resolution. In DNA sequencing, DNA is broken down into small fragments which are then sequenced using sequencing machines that produce hundreds of millions of short DNA reads. These short reads are then processed to identify the differences between the DNA under test and a reference genome. Processing all these millions of short reads is a very time consuming process. Therefore, acceleration and optimization of the analysis time is needed to make DNA diagnostics feasible for a large population.

Read alignment is a core step in DNA analysis, which is the process of comparing two DNA strings to identify the amount of similarity between them. In read alignment, a short read (with a typical length of 100 bases) is aligned against a huge reference genome of for example nearly 3 billion bases for the human genome. The Smith-Waterman algorithm [1], used as a standard for sequence alignment, has $O(n \cdot m)$ complexity to align two sequences of length n and m. However, using Smith-Waterman for the alignment process is too time consuming for any practical purposes [2]. For this reason, many new read aligners have emerged in the past few years.

With the continued improvements in sequencing technologies, sequencing read lengths continue to increase gradually. BWA-MEM has been designed to perform efficient and accurate alignment of longer reads, making it one of the most popular alignment algorithms available [3]. However, BWA-MEM is one of the most computationally intensive algorithms needed for DNA analysis. In this work, we discuss the acceleration of BWA-MEM using both hardware and software techniques. This is the first acceleration of BWA-MEM reported in the literature. In this work, we accelerate BWA-MEM on the Convey HC2ex platform, consisting of an 8-core Intel Xeon based host processor connected to a co-processor with four Xilinx Virtex-6 FPGAs.

No work has been reported in the literature about the acceleration of BWA-MEM. Numerous works are published on the acceleration of other read alignment algorithms. In [4] is shown to be accelerated on Convey HC2ex platform. BWA-ALN also has an accelerated version on Convey HC1 [5]. Both algorithms are different from BWA-MEM and are not suitable for longer reads. The seed generation stage of the CUSHAW2 [6] is similar to BWA-MEM. It computes maximal exact matches instead of super maximal exact matches and uses a different kind of index for which it requires two passes to find all the seeds as compared to only one pass in BWA-MEM. CUSHAW2 has a GPU accelerated version called CUSHAW2-GPU [7].

The paper proposes to split BWA-MEM in different execution stages, that are then optimally mapped to CPU or FPGA. The algorithm has a lot of DRAM accesses causing large memory waits. We circumvent this problem in software by improving the algorithm to reduce the number of DRAM accesses. We also address this problem in hardware by coalescing the memory accesses. The compute intensive parts of the application are accelerated on the FPGA by exploiting the available parallelism. BWA-MEM is segmented in such a way that enables the parallel execution of host and coprocessor to maximize throughput. With the help of these hardware and software optimizations we are able to achieve a 2.6x speedup for the whole application. A large part of the application is

---

Acknowledgement—This work is supported by the Faculty Development Program of the University of Engineering and Technology Lahore, Pakistan

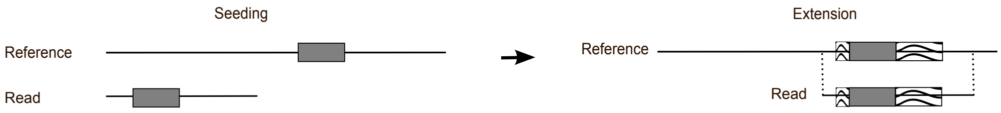


Fig. 1. A seed is first found (rectangular box) and then extended (shaded box).

TABLE I
PERCENTAGE EXECUTION TIME OF BWA-MEM STAGES

Execution stage	% Execution time	Bound
SMEM generation	31.78 - 38.88%	Memory
Suffix array lookup	6.65 - 14.15%	Memory
Seed extension	34.57 - 42.87%	Computation
Output	7.42 - 11.16%	Memory
Miscellaneous	1.33 - 6.04%	–

written in Xilinx Vivado HLS to reduce the design time.

The rest of the paper is organized as follows. Section II summarizes the BWA-MEM algorithm and the profiling results. Section III describes the implementation details of the software optimizations of the SMEM generation. Section IV and V present the hardware acceleration of the suffix array lookup and the local Smith-Waterman stages, respectively. Results are presented in Section VI. Conclusions and future work is discussed in Section VII.

II. BWA-MEM ALGORITHM

Most of the current read aligners (including BWA-MEM) are based on the observation that two DNA sequences of the same species are likely to contain short highly matched substrings. Such kind of aligners follow a *seed-and-extend* strategy, which consists of two steps (1) seeding and (2) extension. The seeding step is to first locate the regions within the reference genome where a substring of the short read is highly matched. This substring is known as a *seed*. A seed could be an exact match or an inexact match with certain allowed number of differences. After seeding the remaining read is aligned to the reference genome around the seed in the extension step using Smith-Waterman or a Smith-Waterman like algorithm. BWA-MEM only finds exact matches while seeding.

A. Execution stages

In BWA-MEM, before starting the read alignment, an index of the reference genome is created. This is a one time step and hence not on the critical execution path. In our discussion we assume that an index is already present. The different execution stages of BWA-MEM read alignment algorithm are described below. The first two stages belong to seeding.

SMEM generation: BWA-MEM first computes the so-called *super-maximal exact matches* (SMEMs). An SMEM is a substring of the read that is exactly matching in the reference DNA and cannot be further extended in either directions. Moreover, it must not be contained in another match.

Suffix array lookup: The suffix array lookup stage is responsible for locating the actual starting position of the SMEM in the reference genome. An SMEM with its known starting position(s) in the reference genome forms seed(s) in the reference.

Seed extension: Seeds are substrings of the read that are exactly matching in the reference genome. As shown in Figure 1, to align the whole read against the reference genome, these seeds are extended in both directions. This extension is performed using a dynamic programming algorithm based on Smith-Waterman.

Output: The read alignment information is written to a file in the SAM (sequence alignment/map) format [8].

B. Profiling results

We have profiled the BWA-MEM software version 0.7.8 to find the percentage execution times of different stages using `gprof`. Table I shows the division of total execution time among the different execution stages, taking a number of input data sets representing different read lengths into consideration. These datasets are acquired from GCAT (Genome Comparison and Analytic Testing) and aligned against the UCSC hg19 human reference genome [9]. The results show that there is no dominant execution stage which means that it is essential to accelerate all the stages to achieve a good overall speedup. The table also shows that the most of the stages of the application are memory bound.

In the following sections, we show the software optimization of the SMEM generation step and the hardware acceleration of the suffix array lookup and seed extension. Compared with the software all these stages can execute in parallel, which allows us to hide some of the computation time by pipelining the queries.

III. OPTIMIZATION OF SMEM GENERATION

A. Theoretical background

This is the first execution stage of BWA-MEM. As described above, the task is to find SMEMs. BWA-MEM uses the *FMD-index* of the reference genome to find the SMEMs. Let T and Y be two sequences of symbols. In DNA analysis, T is the reference DNA while Y is a substring of the read, and the symbols are drawn from an alphabet set Σ consisting of only four symbols i.e. $\Sigma = \{a, t, c, g\}$. The FMD-index is a set of data structures based on the Burrows-Wheeler transform of $T \oplus \overline{T}$ where $\oplus$ is the concatenation operator. $\overline{T}$ is Watson-Crick reverse complement of T. It is formed by first reversing the string T and then replacing symbol a with t and vice versa, and replacing g with c and vice versa. The FMD-index can locate all occurrences of Y in the reference genome T in time proportional to the length of Y. Let SA be the *lexicographically sorted* suffix array of $T \oplus \overline{T}$, where $SA(i)$ represents the ith suffix. Then the *suffix array interval* of Y is defined as $[I_l(Y), I_u(Y)]$, where

$$\begin{aligned} I_l(Y) &= \min\{i : Y \text{ is the prefix of } SA(i)\} \\ I_u(Y) &= \max\{i : Y \text{ is the prefix of } SA(i)\} \end{aligned}$$

$I_l(Y)$ and $I_u(Y)$ are known as the lower and upper limit of the interval, respectively. In other words, the suffix array interval is the set of all those indices of the sorted suffix array in which Y is the prefix. As the suffix array is lexicographically sorted, all these indices will occur together and we only need to know the first and the last index, i.e. $I_l(Y)$ and $I_u(Y)$, respectively. If Y is not present in the reference genome then $[I_l(Y), I_u(Y)]$ is an empty set and if Y is an empty string then $[I_l(Y), I_u(Y)] = [1, 2|T|]$. The size of the interval can be defined as

$$I_s(Y) = I_u(Y) - I_l(Y) + 1 \qquad (1)$$

If Y is not present in the reference genome $I_s(Y)$ is less than or equal to zero. In [10], it is shown that FMD-index can be used to find the *bi-interval* of a given DNA string Y. The bi-interval is defined as $[I_l(Y), I_l(\overline{Y}), I_s(Y)]$. $\overline{Y}$ is Watson-Crick reverse complement of Y. Once the bi-interval of Y is known, the suffix array interval can be found using equation 1. These suffix array intervals are used in the next execution stage (the suffix array lookup stage) to find the exact starting position of Y in the reference genome.

B. SMEM computation in BWA-MEM

An SMEM satisfies three conditions 1) it is an exact match, 2) the match cannot be extended in either directions, and 3) it is not contained in any other match. Algorithm 1 shows the method used in BWA-MEM to find all the SMEMs that include the base at position i_0 of the read P. The complete SMEM computation algorithm can be found in the literature [10].

The algorithm starts from the base at i_0 and first calculates the bi-interval of the string $P[i_0]$ and stores it in Temp array. This calculation is performed by calling the FMDINDEX function. The first `while` loop it moves in the forward direction and adds the next base to the previous string $P[i_0]$, and calculates the bi-interval of the string $P[i_0]P[i_0+1]$. The $forward$ parameter passed to the FMDINDEX indicates the direction of adding the base to the previous string. If this string exists in the reference genome T and the bi-interval is not the same as for $P[i_0]$, its bi-interval is saved in Temp. In this way the algorithm keeps on adding the bases to the previous string and storing the resulting bi-interval in Temp until the string does not exist in the reference genome, i.e. $s \leq 0$. Thus the Temp array contains the bi-intervals of a set of overlapping strings, all starting from i_0. In the second `while` loop the algorithm picks these strings one by one and enlarges them in the reverse direction by adding bases that are behind i_0. For each string its keeps on adding the bases in the backward direction until the resulting string no more exists in the reference genome. The bi-interval of the last hit string is then added to SMEMs array if the length of the string is at least a minimum required. Hence, SMEMs keeps the bi-intervals of the SMEMs found.

One limitation of this algorithm is the accesses to the FMD-index. Every time a base is added in either the forward or backward direction, the FMD-index is accessed. FMD-index is a large data structure having a default total size of 1.5 GB. These accesses to the FMD-index are not local and the difference between the memory addresses of consecutive accesses is huge as studied in [11]. This causes a large amount of data cache and data TLB misses. Most of the accesses to FMD-index end up in the DRAM. Algorithm 1 has a large execution time because it is mostly waiting for the memory request to complete. In our work we speed up the algorithm by reducing the number of FMD-index accesses.

Our improvement is based upon two observations. 1) The algorithm first computes an SMEM and then checks whether its length is greater than the minimum required. In this way, it backward enlarges even those strings which cannot have the total final length greater than the minimum required. 2) All the computed SMEMs have to include the base at position i_0 which means that they overlap with each other and share a common substring. If the bi-interval of this common substring is already known then it can be enlarged to find the SMEMs with less number of FMD-index accesses.

Algorithm 1: SMEM computation

```
Input: String $P$, start position $i_0$, length of reference genome $|T|$, and minimum
    required SMEM length $min\_smem\_len$
Output: Set of bi-intervals $[k, l, s]$ of the SMEMs covering the base at $i_0$
Function CompSMEM($P, i_0, |T|, min\_smem\_len$) begin
    Initialize $[k, l, s]$ as $[1, 1, 2|T|]$
    Initialize Temp, Fwd_len and SMEMs as empty arrays
    $[k, l, s] \leftarrow$ FMDIndex($[k, l, s], P[i_0], forward$)
    Append $[k, l, s]$ to Temp
    $i \leftarrow i_0 + 1$
    $x \leftarrow 1$
    Fwd_len[x] $\leftarrow 1$
    $x \leftarrow x + 1$
    while $i \leq |P|$ and $s > 0$ do
        $[k, l, s] \leftarrow$ FMDIndex($[k, l, s], P[i], forward$)
        if $s > 0$ and $[k, l, s] \neq$ Temp[$x - 1$] then
            Append $[k, l, s]$ to Temp
            Fwd_len[x] $\leftarrow i - i_0 + 1$
            $x \leftarrow x + 1$
        $i \leftarrow i + 1$
    $x \leftarrow 1$
    while $x \leq$ |Temp| do
        $[k, l, s] \leftarrow$ Temp[$x$]
        for $i \leftarrow i_0 - 1$ to 1 do
            $[k, l, s] \leftarrow$ FMDIndex($[k, l, s], P[i], backward$)
            if $s \leq 0$ then
                $back\_len \leftarrow (i_0 - i - 1)$
                $smem\_len \leftarrow$ Fwd_len[$x$] $+ back\_len$
                if $smem\_len \geq min\_smem\_len$ then
                    Append $[k, l, s]$ to SMEMs
                break
        $x \leftarrow x + 1$
    return SMEMs
```

Algorithm 2: Optimized SMEM computation

```
Input: String $P$, start position $i_0$, length of reference genome $|T|$, minimum
       required SMEM length $min\_smem\_len$, a parameter to turn on-off the
       optimization $max\_fwd\_distance$
Output: Set of bi-intervals $[k, l, s]$ of the SMEMs covering the base at $i_0$
1  Function
   CompSMEMOpt($P, i_0, |T|, min\_smem\_len, max\_fwd\_distance$)
   begin
2      Initialize $[k, l, s]$ as $[1, 1, 2|T|]$
3      Initialize Temp, Fwd_len, Back_intv and SMEMs as empty arrays
4      $[k, l, s] \leftarrow$ FMDIndex($[k, l, s], P[i_0], forward$)
5      Append $[k, l, s]$ to Temp
6      $i \leftarrow i_0 + 1$
7      $x \leftarrow 1$
8      Fwd_len[x] $\leftarrow 1$
9      $x \leftarrow x + 1$
10     while $i \leq |P|$ and $s > 0$ do
11         $[k, l, s] \leftarrow$ FMDIndex($[k, l, s], P[i], forward$)
12         if $s > 0$ and $[k, l, s] \neq Temp[x - 1]$ then
13             Append $[k, l, s]$ to Temp
14             Fwd_len[i] $\leftarrow i - i_0 + 1$
15             $x \leftarrow x + 1$
16         $i \leftarrow i + 1$
17     $x \leftarrow 1$
18     $start \leftarrow i_0$
19     $stop \leftarrow i_0$
20     while $x \leq$ |Temp| do
21         $[k, l, s] \leftarrow$ Temp[$x$]
22         if Back_intv is empty or $stop - start \geq max\_fwd\_dist$
           then
23             ($[k, l, s], back\_len$, Back_intv) $\leftarrow$
               BackEnlarge($[k, l, s], P, i_0$)
24             $start \leftarrow i_0 +$ Fwd_len[$x$]
25             $stop \leftarrow i_0 +$ Fwd_len[$x + 1$]
26         else
27             $stop \leftarrow$ Fwd_len[$x$]
28             ($[k, l, s], back\_len$) $\leftarrow$
               FwdEnlarge($[k, l, s], P, start, stop$, Back_intv)
29         $smem\_len \leftarrow$ Fwd_len[$x$] $+ back\_len$
30         if $smem\_len \geq min\_smem\_len$ then
31             Append $[k, l, s]$ to SMEMs
32         $x \leftarrow x + 1$
33         $max\_len \leftarrow$ Fwd_len[$x$] $+ back\_len$
34         while $max\_len < min\_smem\_len$ do
35             $x \leftarrow x + 1$
36             $max\_len \leftarrow$ Fwd_len[$x$] $+ back\_len$
37     return SMEMs
```

Algorithm 3: Backward enlargement

```
Input: The bi-interval $[k, l, s]$, string $P$, start postion $i_0$, array to store
       intermediate intervals Back_intv
Output: Bi-interval $[k, l, s]$, length of backward string $back\_len$
Function BackEnlarge($[k, l, s], P, i_0$) begin
    Initialize Back_intv as empty array
    $back\_len \leftarrow 0$
    for $i \leftarrow i_0 - 1$ to $1$ do
        $[k, l, s] \leftarrow$ FMDIndex($[k, l, s], P[i], backward$)
        if $s > 0$ then
            Append $[k, l, s]$ to Back_intv
            $back\_len \leftarrow back\_len + 1$
        else
            return ($[k, l, s], back\_len$, Back_intv)
```

Algorithm 4: Forward enlargement

```
Input: The bi-interval $[k, l, s]$, string $P$, start and stop index for enlargement
       $start$ $stop$ respectively, array of intermediate intervals Back_intv
Output: Bi-interval $[k, l, s]$, length of backward string $back\_len$
Function FwdEnlarge($[k, l, s], P, start, stop$, Back_intv) begin
    $back\_len \leftarrow$ |Back_intv|
    for $x \leftarrow$ |Back_intv| to $1$ do
        for $i \leftarrow start$ to $stop$ do
            $[k, l, s] \leftarrow$ FMDIndex($[k, l, s], P[i], forward$)
            if $s < 0$ then
                break
        else
            if $i = stop$ then
                return ($[k, l, s], x$)
```

C. *Our optimization*

Algorithm 2 to 4 present our optimized implementation of the SMEM generation. We have improved the algorithm of the second `while` loop shown in a box in Algorithm 2. Calculation of the Temp array is same as in the original algorithm. Based on the observations discussed in the previous section we have made two improvements in the original algorithm.

1) If we are certain that after the backward enlargement of a bi-interval, the total length of the resulting exact match cannot be larger than the minimum required, we skip the backward enlargement of the bi-interval. This will avoid wasteful accesses to the FMD-index. This is implemented in lines 33 to 36 in Algorithm 2.

2) The SMEMs have to include the base at position i_0. All the strings covering the base at position i_0 can be visualized as a concatenation of two strings. One that contains the symbols at read positions greater or equal to i_0 and the other containing the symbols at read positions less than i_0. We may call them as *forward string* and *backward string* respectively. The forward string of an SMEM is found in the first `while` loop of Algorithm 1 or 2 (both are the same). Therefore, before starting the second loop we know the forward strings of all possible SMEMs. If the SMEMs corresponding to the bi-intervals in SMEMs array are numbered as SMEM_i, SMEM_{i+1} … SMEM_{i+r} …, where SMEM_{i+1} is computed after SMEM_i, then

$$\text{SMEM}_i = P[i_0 - m_i] \dots P[i_0 - 1]P[i_0]P[i_0 + 1] \dots P[i_0 + n_i]$$

$$\text{SMEM}_{i+r} = P[i_0 - m_{i+r}] \dots P[i_0 - 1]P[i_0]P[i_0 + 1] \dots P[i_0 + n_{i+r}]$$

Now we will show that how our optimized method calculates SMEM_{i+r} using SMEM_i. The first `while` loop mandates that $n_{i+r} > n_i$. Due to this and the reason that both SMEMs must not contain each other (condition 3 for an SMEM), $m_{i+r} < m_i$. In our technique, we compute SMEM_i using the original method but while doing that we store the bi-intervals of all the intermediate strings formed during the backward enlargement in the Back_intv array. All these intermediate strings have the same forward string, but their

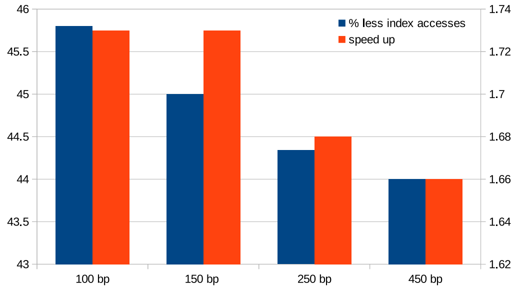


Fig. 2. Percentage reduction in FMD-index accesses (left Y-axis) and the corresponding speedup in SMEM generation (right Y-axis).

backward strings are extended by only one symbol. Hence, while computing SMEM$_i$ we keep track of the bi-intervals of all the strings: from $P[i_0-1]P[i_0]P[i_0+1]\dots P[i_0+n_i]$ to $P[i_0-m_i]\dots P[i_0-1]P[i_0]P[i_0+1]\dots P[i_0+n_i]$. As $m_{i+r} < m_i$, one of these strings is always $P[i_0-m_{i+r}]\dots P[i_0-1]P[i_0]P[i_0+1]\dots P[i_0+n_i]$ which is a substring of SMEM$_{i+r}$ and contains the full backward string of SMEM$_{i+r}$ but partial forward string ($n_{i+r} > n_i$). In our optimized version, this substring is enlarged in the forward direction by $(n_{i+r}-n_i)$ bases to find SMEM$_{i+r}$. To compute SMEM$_{i+r}$ using this method, total number of accesses to the FMD-index are $n_{i+r}+[(m_i-m_{i+r})\cdot(n_{i+r}-n_i)]$ as compared to $n_{i+r}+m_{i+r}$ in the original algorithm. The first term for both these methods is the same and it is observed that on the average, for small r, the second term of our technique is smaller than the second term of the original method, i.e. for small r, $(m_i-m_{i+r})\cdot(n_{i+r}-n_i) < m_{i+r}$. In our implementation we apply our optimization if $n_{i+r}-n_i \leq max\_fwd\_dist$ (default value 3), otherwise the computation is completed using the original method.

The results of our optimization are shown in Figure 2. The read lengths are given in number of base pairs (bps), which is the same as number of bases. The plot shows that for different read lengths the reduction in FMD-index accesses is 44-45% and the corresponding speedup varies from 1.66-1.73x. Our optimization only relies on algorithmic improvement and has limited overhead as we only need one extra array i.e. Back_intv. The maximum possible length of this array is equal to the read length.

IV. SUFFIX ARRAY LOOKUP ACCELERATION

The suffix array lookup is taking around 15% of the computation time of the whole application. The objective is to retrieve the index in the reference string of one suffix string. In order to reduce the memory required for this, partial suffix and occurrence arrays are stored, and the inverse compressed suffix array (inverse CSA) is used [12]. The inverse CSA is given by:

$$\psi^{-1}(i) = C(B[i]) + O(B[i], i)$$

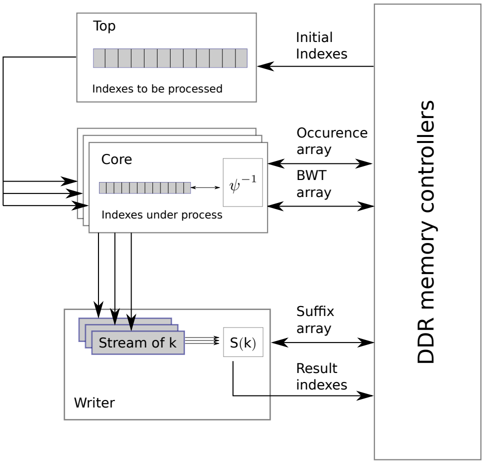


Fig. 3. Suffix array lookup architecture

where B is the BWT string, C is the count array and O is the occurrence array. The above equation is applied until the result is a factor of 32 and then the suffix can be computed using:

$$S(k) = S((\psi^{-1})^{(j)}(k)) + j$$

This implies that a series of memory reads are done to the B and O arrays. Due to their interdependency, these reads can not be prefetched by a general purpose cache. The result is that when running this function on the CPU, most of the time will be spent waiting for memory.

The hardware implementation can avoid waiting for the memory by using two main ideas: batching the input data and pipelining the memory reads. The batching will increase the number of memory reads that can be performed, while the pipelining will hide the memory latency. We implemented an architecture where indexes are sent to a number of hardware cores, and each hardware core processes a number of reads in a pipeline fashion. For the Convey platform, we implement a pipeline of 32 indexes, with 5 cores per FPGA, which gives a total of 640 indexes processed at the same time. This architecture can be see in Figure 3. All the represented blocks (Top, Core, Writer) will execute in parallel. Streams are used to avoid the need of synchronization. The number of core blocks can be adjusted depending on the FPGA area available.

V. SEED EXTENSION ACCELERATION

The purpose of the seed extension kernel is to extend the length of an exact match while allowing for small differences, such as mismatches between the read and reference, or skipping symbols on either the read or reference. To obtain the extension, a method is used that is similar to the well-known Smith-Waterman algorithm [1], which is a dynamic programming method guaranteed to find the optimum alignment between two sequences for a given scoring system. A *similarity matrix* is filled that computes the best score out of all combinations of matches, mismatches and gaps.

A natural way to map dynamic programming algorithms onto reconfigurable hardware is as a linear systolic array. Many implementations that map the Smith-Waterman algorithm onto a systolic array have been proposed, amongst others [13], [14] and [15]. A systolic array consists of processing elements (PEs) that operate in parallel. In our case, we use such an array to take advantage of the available parallelism that exists while filling the similarity matrix, by processing the cells on the anti-diagonal in parallel. We map one read symbol to one PE. Hence, the length of the PE-array determines the maximum length of a read that can be processed. Each cycle, a PE processes one cell of the matrix and passes the resulting values to the next element.

A. Key differences

The seed extension kernel used in BWA-MEM is similar to the Smith-Waterman algorithm. However, since the purpose is to extend a seed, and not to find an optimal alignment between two sequences, three key differences arise:

1. Non-zero initial values: For an extension, the match between sequences will always start from their respective first symbols. Hence, unlike normal Smith-Waterman alignment, the initial values of the dynamic programming matrix are non-zero, but depend on the alignment score of the seed found by the SMEM generation function.

2. Additional output generation: Typically, a Smith-Waterman implementation generates a local and global alignment scores, which are the highest score in the matrix and the highest score that spans the entire read, respectively. The seed extension also returns the exact location inside the similarity matrix where these scores have been found. Furthermore, a *maximum offset* is calculated that indicates the distance from the diagonal at which a maximum score has been found. Therefore, the systolic array implementation passes additional values between the PEs compared to a regular Smith-Waterman systolic array implementation

3. Partial similarity matrix calculation: To optimize for execution speed, the software BWA-MEM uses a heuristic to only calculate those cells that are expected to contribute to the final score. Profiling reveals that in practice, only about 42% of all cells are calculated. This heuristic is not needed for our implementation, as it is able to perform all calculations on the anti-diagonal in parallel, which may also result in higher quality alignments.

B. Implementation details

Before deciding upon the final hardware design of the seed extension kernel, a number of ideas and designs alternatives have been considered, varying in acceleration potential, FPGA-resource consumption, suitability for certain data sets, and complexity.

A read has to travel through the entire systolic array, regardless of its actual length. To minimize latency, ideally a read would be processed by a PE-array matching its exact length. However, in practice this is not achievable, since it requires having a PE-array exactly matching each possible read length, which is impractical given the available resources on the FPGA. Therefore, we implemented an array with multiple exit points. This ensures that shorter reads do not have to travel through the entire array, reducing latency and increasing utilization and performance.

The FPGA-accelerated seed extension kernel is 1.5 times faster when comparing one module against one Xeon core. Our design contains three identical modules per FPGA, which is fast enough to completely hide the execution of this kernel by overlapping it with SMEM generation. The Seed Extension kernel implementation and design alternatives are described in more detail in [16].

VI. RESULTS

The machine on which the implementation was tested is a Convey HC2ex. It consists of a host computer with 2 Intel Xeon CPU E5-2643 processors running at 3.30GHz (in total 8 cores) and 64 GB RAM memory. The co-processor is represented by 4 Xilinx Virtex-6 LX760 FPGA with a 64 GB memory attached.

The data set used to test the performance of the alignment algorithm is obtained from the GCAT framework [17]. The read length is 150 base pairs using pair ended reads.

The tests were run with the complete input and output in a RAM disk. A RAM disk is a virtual disk built in the DDR memory. This way, we eliminate any possible I/O limitations from the test. We note that there are options fast enough to feed and process the data output by the algorithm, either network or high performance disks, but we decided to choose the RAM disk solution to simplify as much as possible the methodology. Each test was run multiple times and the best result was selected. We measured the execution time of the original software, the hardware version without SMEM generation improvements and the version with all improvements. The results are presented in Table II.

We can notice, that due to Amdahl's law, accelerating Smith-Waterman and suffix array lookup, resulted only in a moderate speedup of 1.9x. Improving further the SMEM generation had a big impact, driving the total speedup to 2.6x

We also performed a more complete test of the behavior of the optimizations for different numbers of processors. The results are shown in Figure 4.

To show the possibilities of our design we analyzed the relation between FPGA number, core number and speedup obtained, considering the currently implemented hardware kernels. We assumed a 20% overhead for each FGPA for the memory controllers. The FPGAs and CPU cores are the same used for the implementation detailed above. The results are presented in Figure 5. We can see that to obtain a 2.5x speedup, for the current system we need to use only 2 FPGAs. For 3 FPGAs, we can get 2.5x speedup for up to 16 cores. For 4 FPGAs we can get the maximum speedup up until 20 CPU cores. It is worth mentioning that we focused our

TABLE II
EXECUTION TIMES (IN SECONDS) AND SPEEDUPS OF ACCELERATED BWA-MEM

Data set	SW exec	HW exec	Speedup vs SW	HW exec + SMEM opt	Speedup vs SW
gcat_set_041	534.00	280.00	1.91	208.00	2.57
gcat_set_042	530.50	279.00	1.90	208.00	2.55

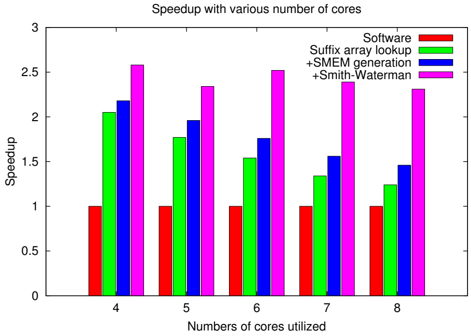


Fig. 4. Speedup for different numbers of CPU cores on the Convey HC2ex machine

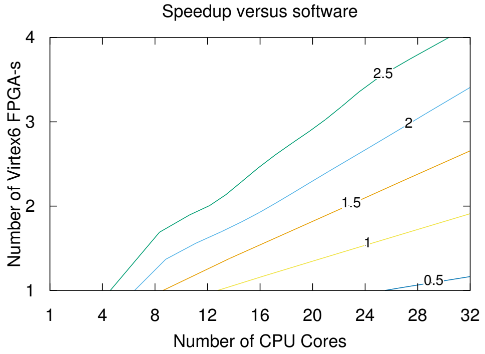


Fig. 5. Speedup in function of number of FPGAs and CPU cores.

implementation on the 8 core processor system, and we do not exclude further optimizations that would allow the area to be utilized better.

VII. CONCLUSION AND FUTURE WORK

This paper presented a software optimized and hardware accelerated implementation of the well-known BWA-MEM DNA read mapping algorithm. The implementation focused on the three highest computationally expensive execution stages of the algorithm: SMEM generation, suffix array lookup and local Smith-Waterman. A system architecture was proposed to achieve a high acceleration for these components, containing two hardware-accelerated kernels and one kernel optimized in software. The two hardware kernels of suffix array lookup and local Smith-Waterman are able to reach speedups of 2.8x and 5.7x, respectively. The software optimization of the SMEM generation kernel is able to achieve a speedup of 1.7x. This enables a total application acceleration of 2.6x compared to the original software version. Analysis shows that the implementation is bottlenecked by the software part, which indicates that further acceleration of BWA-MEM functions in hardware could achieve higher performance. This will be the focus of our future work.

REFERENCES

[1] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[2] T. W. Lam *et al.*, "Compressed indexing and local alignment of dna," *Bioinformatics*, vol. 24, no. 6, pp. 791–797, Mar. 2008.

[3] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," *arXiv [q-bio.GN]*, May 2013. [Online]. Available: http://arxiv.org/abs/1303.3997

[4] E. Fernandez *et al.*, "FHAST: FPGA-based acceleration of Bowtie in hardware," *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 2015.

[5] "Hybrid Core Computing and Bioinformatics Applications," http://www.hpcsociety.org/Resources/Documents/121212Kirby-CONVEY-SHPCP_121212.pdf.

[6] Y. Liu and B. Schmidt, "Long read alignment based on maximal exact match seeds," *Bioinformatics*, vol. 28, no. 18, pp. i318–i324, 2012.

[7] ——, "Cushaw2-gpu: Empowering faster gapped short-read alignment using gpu computing," *Design Test, IEEE*, vol. 31, no. 1, pp. 31–39, Feb 2014.

[8] H. Li *et al.*, "The sequence alignment/map format and samtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[9] "Genome Comparison and Analytic Testing," http://www.bioplanet.com/gcat.

[10] H. Li, "Exploring single-sample SNP and indel calling with whole-genome de novo assembly," *Bioinformatics*, vol. 28, no. 14, pp. 1838–1844, Jul 2012.

[11] J. Zhang *et al.*, "Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures," in *CCGrid*, Delft, Netherlands, May 2013.

[12] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[13] T. Oliver, B. Schmidt, and D. Maskell, "Hyper customized processors for bio-sequence database scanning on FPGAs," in *Proceedings of the ACM/SIGDA*. ACM, 2005, pp. 229–237.

[14] C. W. Yu *et al.*, "A Smith-Waterman systolic cell," in *New Algorithms, Architectures and Applications for Reconfigurable Computing*. Springer, 2005, pp. 291–300.

[15] P. Zhang, G. Tan, and G. R. Gao, "Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform," in *Proceedings of the HPRCTA'07*. ACM, 2007, pp. 39–48.

[16] E. Houtgast, V. Sima, K. Bertels, and Z. Al-Ars, "An FPGA-Based Systolic Array to Accelerate the BWA-MEM Genomic Mapping Algorithm," in *SAMOS XV*. IEEE, 2015.

[17] G. Highnam *et al.*, "An analytical framework for optimizing variant discovery from personal genomes," *Nature communications*, vol. 6, 2015.

3

GPU Accelerated API for Sequence Alignment

This chapter discusses our GPU acceleration of sequence alignment algorithms. These algorithms are widely used in various DNA analysis applications in practice, either standalone or in combination with other algorithms. For example, seed-and-extend DNA analysis algorithms are composed of 2 steps: the seeding step (discussed in Chapter 2), followed by seed-extension (discussed in this chapter), which is implemented using sequence alignment algorithms.

This chapter discusses GASAL2, our GPU library for sequencing alignment of high-throughput NGS data. After a brief introduction of sequence alignment algorithms, we motivate the development of GASAL2. The implementation details of GASAL2 are then presented. The performance limitations of NVBIO, NVIDIA's sequence alignment library, are also identified and overcome in GASAL2. We also list the advantages of GASAL2 over existing GPU based libraries. Performance measurement of GASAL2 shows that it is up to 20x faster than Parasail, a CPU based sequence alignment library, running on 56 Intel Xeon threads and up to 13x faster than NVBIO. GASAL2 is used to accelerate the seed extension stage of BWA-MEM read mapper which resulted in 1.3x overall application speedup.

In this chapter we also present the GPU acceleration of the Darwin read overlapper [22] used for assembly of long DNA reads. After a brief introduction on read assembly, we describe the Darwin read overlapping algorithm. The implementation details of the GPU acceleration are provided next. The results of the acceleration show that with real Pacbio data, our GPU implementation on NVIDIA Tesla K40 GPU is 109x than 8 CPU threads of Intel Xeon machine and 24x faster than 64 threads of IBM Power8 machine.

This chapter consists of the following articles:

- N. Ahmed, J. Lévy, S. Ren, H. Mushtaq, K. Bertels and Z. Al-Ars, *GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data*, BMC Bioinformatics 20, 520 (2019).
- N. Ahmed, T. D. Qiu, K. Bertels, and Z. Al-Ars, *GPU Acceleration of Darwin Read Overlapper for de Novo Assembly of Long DNA Reads*, (2020), Accepted for publication in *BMC Systems Biology*.

Ahmed *et al. BMC Bioinformatics* (2019) 20:520
https://doi.org/10.1186/s12859-019-3086-9

BMC Bioinformatics

SOFTWARE Open Access

GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data

Nauman Ahmed[1]*, Jonathan Lévy[2], Shanshan Ren[2], Hamid Mushtaq[3], Koen Bertels[2] and Zaid Al-Ars[2]

Abstract

Background: Due the computational complexity of sequence alignment algorithms, various accelerated solutions have been proposed to speedup this analysis. NVBIO is the only available GPU library that accelerates sequence alignment of high-throughput NGS data, but has limited performance. In this article we present *GASAL2*, a GPU library for aligning DNA and RNA sequences that outperforms existing CPU and GPU libraries.

Results: The GASAL2 library provides specialized, accelerated kernels for local, global and all types of semi-global alignment. Pairwise sequence alignment can be performed with and without traceback. GASAL2 outperforms the fastest CPU-optimized SIMD implementations such as SeqAn and Parasail, as well as NVIDIA's own GPU-based library known as NVBIO. GASAL2 is unique in performing sequence packing on GPU, which is up to 750x faster than NVBIO. Overall on Geforce GTX 1080 Ti GPU, GASAL2 is up to 21x faster than Parasail on a dual socket hyper-threaded Intel Xeon system with 28 cores and up to 13x faster than NVBIO with a query length of up to 300 bases and 100 bases, respectively. GASAL2 alignment functions are asynchronous/non-blocking and allow full overlap of CPU and GPU execution. The paper shows how to use GASAL2 to accelerate BWA-MEM, speeding up the local alignment by 20x, which gives an overall application speedup of 1.3x vs. CPU with up to 12 threads.

Conclusions: The library provides high performance APIs for local, global and semi-global alignment that can be easily integrated into various bioinformatics tools.

Keywords: Genomics, Sequence alignment, NGS, GPU library

Background

Many applications for processing NGS sequencing data depend heavily on sequence alignment algorithms to identify similarity between the DNA fragments in the datasets. Well known programs for DNA mapping such as BWA-MEM [1] and Bowtie2 [2], DNA assemblers such PCAP [3] and PHRAP [4], make repeated use of these alignment algorithms. Furthermore, in various practical multiple sequence alignment algorithms, many pairwise sequence alignments are performed to align sequences with each other. Also, alignment based read error correction algorithms, like Coral [5] and ECHO [6], perform a large number of pairwise sequence alignments. In addition, variant callers for NGS data e.g. GATK HaplotypeCaller [7], also make use of sequence alignment.

Sequence alignment is the process of editing two or more sequences using gaps and substitutions such that they closely match each other. It is performed using dynamic programming. There are two types of sequence alignment algorithms for biological sequences: *global alignment* and *local alignment*. The former is performed using the Needleman-Wunsch algorithm [8] (NW), while Smith-Waterman algorithm [9] (SW) is used for the latter. Both algorithms have been improved by Gotoh [10] to use affine-gap penalties. These alignment algorithms can be divided into the following classes:

- *Global alignment*: In global alignment, also known as end-to-end alignment, the goal is to align the sequences in their entirety while maximizing the alignment score.

---

*Correspondence: n.ahmed@tudelft.nl
[1] Delft University of Technology, Delft, Netherlands and University of Engineering and Technology, Lahore, Pakistan
Full list of author information is available at the end of the article

© The Author(s). 2019, corrected publication 2019 **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made. The Creative Commons Public Domain Dedication waiver (http://creativecommons.org/publicdomain/zero/1.0/) applies to the data made available in this article, unless otherwise stated.

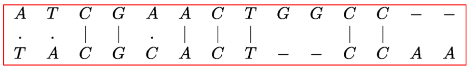
- *Semi-global alignment*: Unlike global alignment, semi-global alignment finds the overlap between the two sequences, allowing to skip the ends of a sequence without penalty. In semi-global alignment the gaps at the leading or trailing edges of the sequences can be ignored, without inducing any score penalty. Different kinds of semi-global alignments are possible depending on which sequence can have its beginning or end be skipped. GASAL2 supports all kinds of semi-global alignments where any combination of beginning or end of a pair of sequences can be ignored.
- *Local alignment*: In local alignment, the goal is to align two sequences so that the alignment score is maximized. As opposed to global alignment, the final alignment may not contain the whole of the sequences. No penalty is induced by misalignments in the beginning and end of the sequences, and the score is kept positive.

Figure 1 shows the alignment of the two sequences shown in Fig. 2. The bases enclosed in the box constitute the alignment. Match score is 3; mis-match penalty is 4; gap open and gap extension penalties are 6 and 1, respectively. For global alignment the alignment score is -5. In case of semi-global alignment the gaps at the end of S_1 are not penalized. The alignment score is 7, while the start and end positions of the alignment on S_2 are 2 and 10, respectively. For local alignment, the final alignment score is 10. The end-positions of the alignment on S_1 and S_2 are 12 and 10, respectively. The start-position is 3 on both sequences.

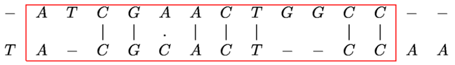
Graphical processing units

Graphical Processing Units (GPUs) were developed for rendering graphics, but are now being used to accelerate many other applications due to their massively parallel architecture. The GPU architecture varies from one vendor to the other and even across different GPU generations from the same vendor. Here we give a general overview of state-of-the-art NVIDIA GPUs. The cores of a GPU, known as streaming processors (SPs), groups of which are organized into a number of streaming multiprocessors (SMs). Each SM has a set of SPs, a register file, one or more thread schedulers, a read only memory, L1 cache, shared memory, and some other hardware units. All SMs access the DRAM (known as global memory) through a shared L2 cache. The programming language for NVIDIA GPUs is known as *CUDA* which is an extension of C/C++. The function that executes on the GPU is known as *kernel*. The data to be processed by the kernel is first copied from the CPU memory into the global memory of the GPU. The CPU (known as the *host*) then launches the kernel. Once the kernel is finished the results are copied from the global memory back into CPU memory. This copying of data back and forth between host and GPU is quite time expensive. Therefore, data is transferred between the host and GPU in the form of large batches to keep number of transfers at minimum. Moreover, the batch should be large enough to fully utilize the GPU resources.

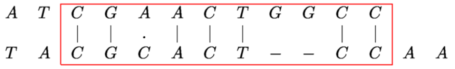
At every clock cycle each SM executes instructions from a group of threads known as a *warp*. A *warp* is a set of 32 GPU threads that execute in lock-step (i.e., they share the instruction pointer). Therefore, if one or more threads execute a different instruction, different execution paths are serialized causing performance loss. This phenomenon is known as *divergent execution* and should be avoided as much as possible. Moreover, to achieve good memory throughput the memory accesses should be coalesced (i.e., all the threads in a warp should access consecutive memory locations).



(a) Global alignment example



(b) Semi-global alignment example



(c) Local alignment example

Fig. 1 Alignment of S_1 and S_2 sequences shown in Fig. 2. **a** Global alignment example. **b** Semi-global alignment example. **c** Local alignment example

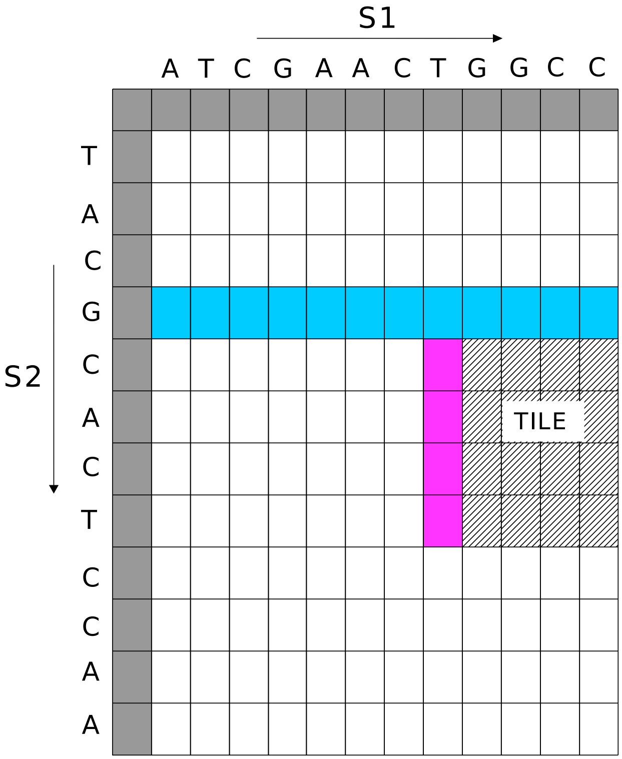


Fig. 2 Identical *H*, *E* and *F* matrix

To allow the overlapping of GPU and CPU execution, all the GPU kernel launches are asynchronous i.e. control is immediately returned to the CPU after the kernel launch. In this way, the launching thread can perform other tasks instead of waiting for the kernel to complete. Using *CUDA streams*, it is possible to launch one or more kernels on GPU before the results of a previously launched kernel has been copied back to the CPU. CUDA streams also allow to asynchronously perform the copying operations. Hence, one can just launch all the operations and perform other tasks on the CPU. Subsequently, the `cudaStreamQuery()` API function can be used to test whether all the operations in a given stream have completed or not.

Previous research works

GPU acceleration of sequence alignment has been the topic of many research papers like [11–13]. Apart from sequence alignment , GPUs are also used for accelerating many other bioinformatics algorithms, such as, described in [14, 15]. Moreover, various biomedical image analysis applications are accelerated with GPUs. Kalaiselvi et al. [16] surveys the GPU acceleration of medical image analysis algorithms. In [17, 18], GPUs are used to accelerate the processing of MRI images for brain tumour detection and segmentation. Most of the previous work on accelerating sequence alignment, was focused on developing search engines for databases of protein sequences. The alignment of DNA and RNA sequences during the processing of high-throughput NGS data poses a different set of challenges than database searching as described below.

1 The sequences to be aligned in NGS processing are generated specifically for each experiment. In contrast, in database searching, the database of sequences is known in advance and may be preprocessed for higher performance.
2 In database search programs, one or few query sequences are aligned against all the sequences in the database (may be regarded as target sequences), whereas the processing of NGS data requires

pairwise one-to-one, one-to-many or all-to-all pairwise sequence alignment. Due to this, a common performance improvement technique in database search programs, like using *query profile*, is not feasible in NGS data alignment.

3 In programs containing GPU accelerated sequence alignment, the alignment step is tightly coupled with the rest of the program. The GPU alignment kernel is specifically tailored to meet the requirements of the program. Therefore, reusing the kernel to accelerate the sequence alignment in other programs is not easy.

Due to these differences, GPU accelerated database search cannot be used to accelerate the alignment step in NGS data processing programs. gpu-pairAlign [19] and GSWABE [20] present only all-to-all pairwise local alignment of sequences. All-to-all alignment is easier to accelerate on GPU. Since, only one query sequence is being aligned to all other sequences, the query sequence may reside in the GPU cache, substantially reducing global memory accesses. On the other hand, in one-to-one alignment each query sequence is different limiting the effectiveness of caching these sequences. In many NGS data processing applications, one-to-one pairwise alignment is required (e.g., DNA read mapping). In DNA read mapping, local alignment takes a substantial percentage of the total run time. For example, in the BWA-MEM DNA read aligner the local alignment takes about 30% of the total execution time with query lengths of 250bp (or base pairs), while calculating only the score, start-position and end-position.

None of the previously published research efforts have developed any GPU accelerated sequence alignment library that can be easily integrated in other programs that require to perform pairwise alignments. NVBIO [21] is the only public library that contains GPU accelerated functions for the analysis of DNA sequences. Although this library contains a GPU accelerated function for sequence alignments, its performance is limited. Therefore, in this paper we present a GPU accelerated library for pairwise alignment of DNA and RNA sequences, GASAL2 (GPU Accelerated Sequence Alignment Library v2), as an extension of our previously developed GASAL library described in [22]. The library contains functions that enable fast alignment of sequences and can be easily integrated in other programs developed for NGS data analysis. Functions for all three types of alignment algorithms (i.e., local, global and semi-global) are available in GASAL2. One-to-one as well as all-to-all and one-to-many pairwise alignments can be performed using affine-gap penalties. The contributions of the paper are as follows:

- A GPU accelerated DNA/RNA sequence alignment library that can perform global, semi-global (all types) as well as local alignment between pair of sequences. The library can compute the alignment score and the actual alignment between two sequences by performing traceback. The actual alignment is generated in CIGAR format and contains the exact position of matches, mismatches, insertion and deletion in the alignment. Optionally it can compute the alignment score with only the end, and if required, the start position of the alignment.
- The library uses CUDA streams so that the alignment functions can be called asynchronously and the host CPU can perform other tasks instead of waiting for the alignment to complete on the GPU.
- GASAL2 is the fastest sequence alignment library for high-throughput Illumina DNA sequence reads in comparison to highly optimized CPU-based libraries, and it is also much faster than NVBIO, NVIDIA's own GPU library for sequence analysis.
- GASAL2 can be easily integrated in bioinformatics applications, such as accelerating the seed-extension stage of BWA-MEM read mapper.

Implementation

In this paper, we describe GASAL2, a GPU accelerated library for pairwise sequence alignment. The sequences are first transferred to the GPU memory, where they are *packed* into unsigned 32-bit integers. If needed, any number of sequences can then be reverse-complemented. Finally, the alignment is performed and the results are fetched back from the GPU memory to the CPU memory. This section gives an overview of the implementation choices of GASAL2 and describes various stages in the data processing pipeline performed on the GPU.

Stage-1: data packing

The user passes the two batches of sequences to be pairwise aligned. A batch is a concatenation of the sequences. Each base is represented in a byte (8-bits). DNA and RNA sequences are made up of only 5 nucleotide bases, A, C, G, T/U (T in case of DNA and U in RNA) and N (unknown base), 3 bits are enough to represent each symbol of a sequence. However, we represent each base in 4 bits for faster packing. Due to the compute bound nature of the GASAL2 alignment kernel, using 3-bits does not result in any significant speedup over the 4-bit representation, but instead complicates the data packing process. Registers in the GPU are 32-bits wide. Therefore, a batch of sequences is packed in an array of 32-bit unsigned integers in which each base is represented by 4 bits. NVBIO also packs the sequences on CPU using 4 bits per base. As the total number of bases in a batch is quite large, packing the data on the CPU is very slow. Figure 3 shows the percentage of data packing in the total execution time for one-to-one pairwise alignment of the input dataset.

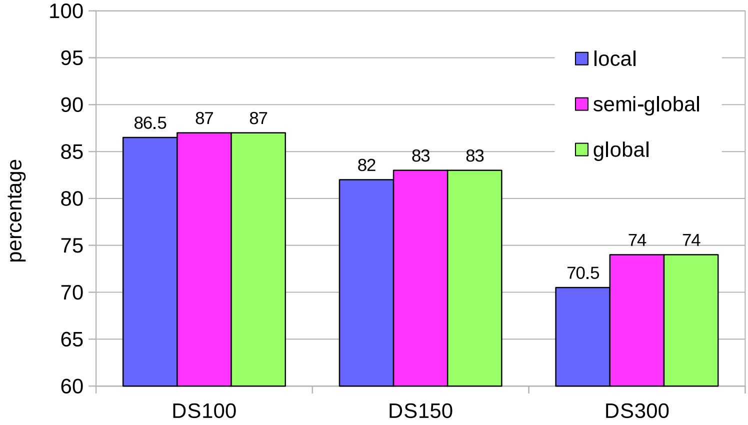


Fig. 3 NVBIO data packing time as percentage of total execution time

The input dataset and GPU platform are described in "Input dataset and execution platforms" section on page 6. Figure 3 shows that NVBIO data packing takes around 80% of the total time. Hence, in NVBIO preparing the sequences for the alignment on GPU takes much more time then actually aligning the sequences. Based on this observation, we accelerate the data packing process on GPU and unpacked batches of sequences are copied to the GPU global memory for this purpose. Figure 4 shows how the GPU data packing kernel works on GPU. Each GPU thread loads eight bases at a time from global memory. Each base is converted from 8-bit to 4-bit representation by masking the upper 4 bits, and then packed into an unsigned 32-bit integer which is written back to global memory. Figure 5 shows the achieved speedup of our novel approach of packing the sequences on GPU as compared to sequence packing performed by NVBIO on CPU. GASAL2 is at least 580x faster than NVBIO. Since, only few milliseconds are required to pack the sequences in GASAL2, the data packing time is completely eliminated. After the data packing is complete, packed sequences reside on the GPU memory and all subsequent operations are completely done on the GPU, only the final results of the alignment need to be copied from GPU to CPU.

Stage-2 (optional): reverse-complementing kernel

GASAL2 is able to reverse and/or complement any number of sequences from any batch. Any sequence can be flagged to be reversed, complemented, or reverse-complemented. The reverse-complementing process is performed on the GPU on already packed sequences to take advantage of the high parallelism of the task.

Stage-3: alignment

The sequence alignment kernel is launched to perform pairwise alignment of the sequences using affine-gap scoring scheme. GASAL2 employs inter-sequence parallelization and each GPU thread is assigned a pair of sequences to be aligned. All pairs of sequences are independent of the others, so there is no data dependency and all the alignments run in parallel. An alignment algorithm using affine-gap penalties compute cells in three dynamic programming (DP) matrices. These matrices are usually named as H, E and F. The matrices are shown in Fig. 2. Each cell needs the results of 3 other cells: the one on top, the one on the left, and the one on the top-left diagonal. Since the sequences are packed into 32-bits words of 8 bases each, the alignment fetches a word of both sequences from memory and computes an 8x8 tile of the matrix. Hence, 64 cells of the DP matrices are computed with a single memory fetch reducing the number of memory requests. All the tiles are computed from left to right, then top to bottom. To jump from one tile to the next one on the right, we need to store 8 intermediate values (which are the values of the cell of the left for the next tile). To jump from one row of tiles to the next row, we need to store a full row of intermediate values (which are the values of the cell of the top for the next row of tiles). Hence, instead of storing the whole matrix, we only store an 8-element column and a full row, which reduces the memory requirement from $O(n^2)$ to $O(n)$. Since, the stored column has only 8 elements it can easily reside in the GPU register file. For ease of representation, Fig. 2 shows a 4 x 4 tile, and the intermediate values that are stored are shown shaded. Our library can also compute the start-position of the alignment without computing the

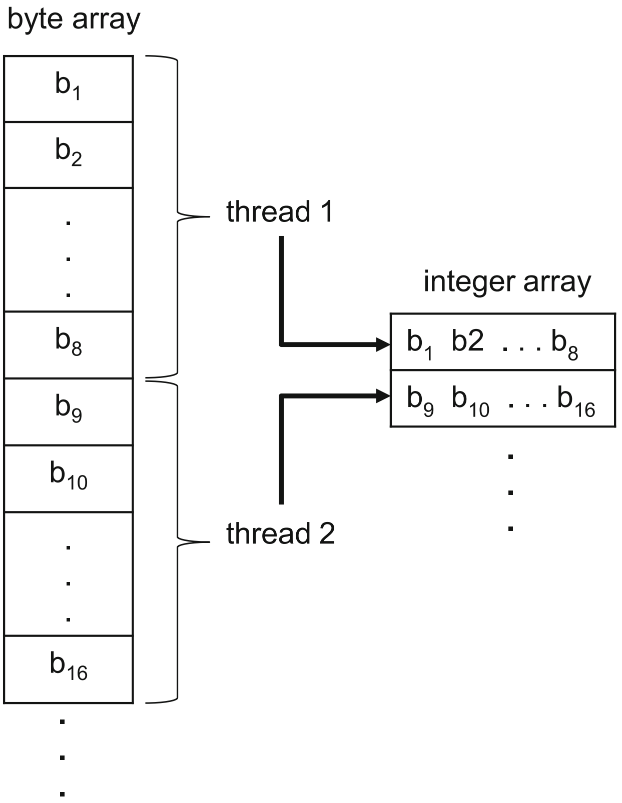


Fig. 4 Packing the sequences on GPU. $b_1, b_2, \ldots,$ are the bases

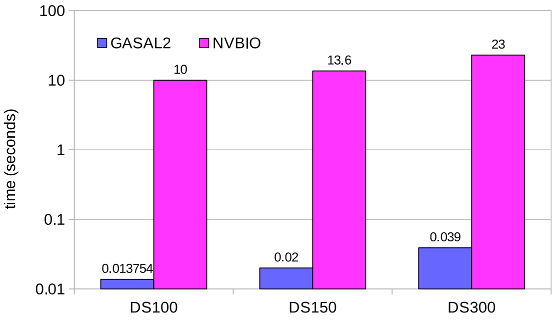


Fig. 5 Data packing time, GASAL2 vs NVBIO

traceback. To do so, we restart the computation, but now from the end-position in the backward direction, and exit where the score becomes equal to the previously found score. The coordinates of the cells at the exit point give the start-position of the alignment.

For computing the traceback a *direction* matrix is stored in the global memory of the GPU while computing the alignment. The direction matrix is similar to the one shown in Fig. 2 with $|S_1| \times |S_2|$ cells. Each cell is represented by 4-bits in the memory. The lower 2 bits are used to encode whether the current cell is match, mismatch, insertion or deletion. The upper two bits are for the next cell on the alignment path. If the next cell is a gap then the upper bits of the current cell represent whether it is a gap-open or gap-extension, one bit each for insertion and deletion. The direction matrix is stored in the memory using `uint4` CUDA vector data type. `uint4` has 4 aligned 32-bit unsigned integers. A single store/load instruction is required to access `uint4` data from the memory. A single `uint4` data element can store 32 direction matrix cells, and hence half the cells in a tile. Moreover, the direction matrices of all the pairs aligned on the GPU are stored in an interleaved fashion for coalesced memory access. The actual alignment is generated using the direction matrix by starting from the end cell and tracing back to the start of the alignment to compute the exact location of matches, mismatches, deletions and insertions.

The output of this stage depends on the users choice. There are three possible outputs: 1) only score and end-position of the alignment. 2) score, end-position and start-position of the alignment without performing traceback. 3) score, end-position, start-position and actual alignment in CIGAR format.

Kernel specialization through templates

GASAL2 supports various kinds of parameters for kernel launches, to tailor the results to the user's need. For example, the traceback will only be calculated if the user requests it. In addition, GASAL2 can adapt to any kind of semi-global alignment where the initialization or the search for a maximum can vary, depending on the user requesting the beginning and/or the end of any sequence.

Dealing with this kind of issue is not trivial in the case of GPU programming, as creating a simple branch through an *if* statement slows down the whole kernel dramatically (for a single *if* in the innermost loop of an alignment kernel, this can cause an approximate slowdown of 40%). Duplicating the kernels is not a viable solution for code maintenance: for example, for the semi-global kernel, there are $2^4 = 16$ types; and adding the possibility of asking for the start-position doubles this number.

The solution that we adopted allows to generate all the possible kernels at compilation time, so that they are all ready to run at full speed without branches. CUDA implementation of C++ templates (according to C++11 specifications) allows to generate all template-kernels at compile time. The programming model that we adopted allows to create a new kernel specialization by writing *if* statements that are resolved at compilation time, to prune the useless branches.

GPU launch parameters choice

GPU threads are organized in *blocks*, and blocks are grouped into *kernel grid*. A *block* is run on a SM that has several hardware resources such as cores, register file, cache, etc. Two parameters characterize the kernel launch:

- the block size, which is the number of threads in a block.
- the grid size, which is the total number of blocks.

Block size affects the *SM occupancy*. The SM occupancy is the ratio of number of active warps and the maximum number of warps allowed on a SM. Increasing the occupancy helps in memory-bound applications. Large occupancy makes sure that they are always enough number of warps that are ready to be scheduled to the streaming processors so that all cores (SP's) in the SM are fully utilized. GASAL2 alignment kernel is not memory-bound. It can compute a 8x8 tile of cells in only 2-3 memory requests. Thus, increasing the occupancy does not help much. However, GASAL2 alignment kernels use a block size of 128 for reasonable occupancy value. GASAL2 uses the inter-sequence parallelization and each GPU thread performs only one alignment. Hence, the grid size is always the ratio of number of alignments to be performed and the block size (128).

GASAL2 asynchronous execution

GASAL2 allows the user to overlap GPU and CPU execution. This is known as *asynchronous* or *non-blocking* alignment function call as opposed to *synchronous* or *blocking* call used in GASAL [22]. In a blocking alignment function call, the calling thread is blocked until the alignment on the GPU is complete. GASAL2 uses CUDA streams to enable asynchronous execution. In asynchronous calls, the calling thread is not blocked and immediately returns after launching various tasks on the GPU. In GASAL2 these tasks are CPU-GPU memory transfers, and the GPU kernels for data packing, reverse-complementing (optional), and pairwise-alignment. The application can perform other tasks on the CPU rather than waiting for the GPU tasks to complete. This helps to eliminate idle CPU cycles in case of a blocking call. Hence, the time spent in the alignment function is merely a small overhead to call the CUDA API asynchronous memory copy functions and launch the kernels.

GASAL2 versus GASAL and NVBIO

The advantages of GASAL2 over GASAL and NVBIO are listed below:

1. GASAL2 can generate the actual alignment between a pair of sequences by computing traceback. The traceback contains the exact position of matches, mismatches, insertion and deletion in the alignment. This facility is not provided in GASAL.
2. GASAL2 is much faster than NVBIO.
3. Asynchronous execution. This is a unique facility that is not available in NVBIO or GASAL.
4. In NVBIO and GASAL, an ambiguous base (N) is treated as a ordinary base having the same match and mismatch scores as A, C, G or T. But, in most sequence analysis programs, the match/mismatch score of "N" is different. For example, in BWA-MEM the score of aligning "N" against any other base (A, C, G, T or N) is always -1. Extending NVBIO to adopt this new scoring scheme to handle "N" increases the execution time of GPU kernels by 30% for global and semi-global alignment, and by 38% for local alignment. In GASAL2 the score of aligning "N" against any other base is configurable. Due to this, the execution time of global, semi-global and local kernels is higher than that of GASAL by 17%, 15% and 6%, respectively.
5. In GASAL, the GPU memory allocations are performed just before the batch of sequences are copied from CPU to GPU. The allocated memory is freed after the alignment is complete and the results are copied from GPU to CPU. If the input batch is not very large, the time spent in memory allocation and de-allocations may become significant and, thus reduces the performance. In GASAL2, we have a separate API function for memory allocation and de-allocation which is called only once at the beginning and end of the program, respectively. At the beginning of the program, user calls the memory allocation function by passing an estimated input batch size. Separate data structures are maintained to keep track of the allocated memory. If the actual input batch is larger, GASAL2 automatically handles the situation by seamlessly allocating more memory. The allocated memory is freed up at the end of the application.
6. GASAL2 supports all types of semi-global alignments. NVBIO and GASAL supports only one type of semi-global alignment in which the gaps at the beginning and end of the query sequence are ignored.
7. GASAL2 can also compute the second-best local alignment score. GASAL only computes the best score.
8. GASAL2 has a reverse-complementing GPU kernel. In NVBIO and GASAL, the user has to manually reverse-complement the sequence before writing it to the input batch.

Results

Input dataset and execution platforms

To evaluate the performance of GASAL2 we performed *one-to-one pairwise* alignments between two set of sequences. We considered the case of DNA read mapping. Read mappers have to perform billions of one-to-one pairwise alignments between short segments of DNA and substrings of the reference genome. In this paper, we also perform one-to-one pairwise alignments between two set of sequences for evaluation purposes. Affine-gap scoring scheme is used in which the match score, mis-match penalty, gap open penalty and gap extension penalty is 6, 4, 11 and 1, respectively. In the rest of the paper,we will refer to the substrings of the reference genome as *target* sequences. The length of the read sequence is fixed, while the length of the target sequence may vary. Table 1 shows the different datasets used in this paper. The read set consists of reads simulated with Wgsim [23] using UCSC hg19 as the reference genome. To generate the target set, these reads and the hg19 reference genome are used as the input for BWA-MEM. During the seed-extension phase of BWA-MEM, the mapper aligns the reads with the substrings of the reference genome. These substrings are stored and used as the target set. Three typical read lengths generated by Illumina high-throughput DNA sequencing machines are used: DS100, DS150 and DS300 representing 100, 150 and 300bp, respectively. Table 1 shows the number of sequences in the read and target set and the corresponding maximum and average length of the sequences in each set. Minimum target sequence length in each case is approximately equal to the length of the read.

The CPU-based libraries are executed on a high end machine consisting of two 2.4 GHz Intel Xeon E5-2680 v4 (Broadwell) processors and 192 gigabytes of RAM. Each processor has 14 two-way hyper-threaded cores. Hence, there are 28 physical and 56 logical cores in total. We measured the execution time of the CPU-based libraries with 28 and 56 threads and reported the smallest execution time of the two. GASAL2 and NVBIO are executed

Table 1 Characteristics of the input dataset

Dataset	Read Set			Target Set		
	avg. len.	max. len.	No. of seq.	avg. len.	max. len.	No. of seq.
DS100	100	100	10e6	162	177	10e6
DS150	150	150	10e6	260	277	10e6
DS300	300	300	10e6	538	571	10e6

on a NVIDIA Geforce GTX 1080 Ti GPU. Only one CPU thread is used in case of GASAL2 and NVBIO. GASAL2 is compiled with CUDA version 10.0.

Libraries compared with GASAL2

We compared GASAL2 against the fastest CPU and GPU based libraries available, which are:

- *SeqAn* contains the vectorized implementation of all types of alignments using SSE4, AVX2 and AVX512 SIMD instructions [24]. For SeqAn we used the test-suite provided by the developers of the library [25]. AVX2 implementation of SeqAn is used in the experiments with 16 bits per score. Since the test data set is based on Illumina reads, we have used `align_bench_par` and `align_bench_par_trace` which follows the *chunked* execution policy giving the fastest execution for short DNA reads. The chunked policy is also used to generate the results in [24] for Illumina reads. `align_bench_par` calculates the alignment score and does not report the start and end positions of the alignment. We have not used the banded version of `align_bench_par` as it does not guarantee correct results. `align_bench_par_trace` is used for computing alignment with traceback. In this paper, we are performing one-to-one alignment for the experiments. The timings reported in the SeqAn paper [24] are not for the one-to-one alignment. The paper used a so-called "olc" alignment mode which is similar to the different one-to-many alignments. The library is compiled with GCC 7.3.1.
- *ksw* module in klib [26] contains a fast SSE based implementation local alignment algorithm. It can also compute the start-position, but does not compute the traceback for local alignment. It has a function for computing the traceback for global alignment, but it is not vectorized, and hence very slow. ksw is faster than SSW [27]. We developed our own test program for ksw (commit:cc7e69f) which uses OpenMP to distribute the alignment tasks among the CPU threads. The test program is compiled with GCC 4.8.5 using O3 optimization flag.
- *Parasail* [28] contains the SIMD implementation of the local, global and semi-global alignment with and without traceback. Ten types of semi-global alignments are supported. We developed our own test program for Parasail (version-2.4) which uses OpenMP to distribute the alignment tasks among the CPU threads. The test program is compiled with GCC 4.8.5 using O3 optimization flag. Parasail allows the user to choose between SSE and AVX2 SIMD implementations. It also consists of different vectorization approaches namely *scan*, *striped*, *diagonal* and *blocked*. We have used the *scan* approach implemented with AVX2 instructions as it is the fastest for our dataset. Parasail does not compute the start-position directly without computing traceback. Therefore, the original sequences are aligned to obtain score and end-position, then both sequences are reversed to calculate the start-position without traceback.
- *NVBIO* contains the GPU implementations of local global and semi-global alignment with and without traceback. Only one type of semi-global alignment is supported shown in Fig. 1. We used `sw-benchmark` program in the NVBIO repository. The original program performs one-to-all alignments. We modified `sw-benchmark` to perform one-to-one alignments. Moreover, in the original program reading the sequences from the files and packing the sequences is done in a single API function call. To exclude the I/O time from the measurements, we first loaded the sequences in an array of strings and then pack the sequences using NVBIO API functions. NVBIO does not contain any function that directly computes the start-position of the alignment without computing the traceback. To compute the start-position without traceback, we make two copies of each sequence, one in original form and other reversed. The alignment of original sequences is used to compute the score and end-position, while the reverse sequence are aligned to compute the start-position. Moreover, as described before, NVBIO considers "N" as an ordinary base and extending the library to correctly handle the ambiguous base makes it more than 30% slower. In our comparison we have used the original NVBIO implementation. NVBIO is compiled with CUDA version 8 as it cannot be compiled with latest CUDA version.

There are also very fast CPU-based libraries that compute the edit distance or sequence alignment with linear-gap penalty e.g. EDlib [29], BitPAl [30] and [31]. EDlib computes the Levenshtein distance between two sequences. Edit distance is the minimum number of substitution, insertions and deletion required to transform one sequence to the other. BitPAl and BGSA [31] can perform global and semi-global alignments with linear-gap penalty. Many bioinformatics applications require sequence alignment with affine-gap penalty which allows to have different penalties for gap opening and gap extension. Moreover EDlib, BitPAl and BGSA cannot compute local alignment.

GASAL2 alignment kernel performance

Table 2 shows a comparison of the alignment kernel execution times of NVBIO and GASAL2. The times listed in the table represent the total time spent in the GPU

Table 2 Alignment kernel times (in seconds) for NVBIO and GASAL2

	DS100		DS150		DS300	
GPU kernel	NVBIO	GASAL2	NVBIO	GASAL2	NVBIO	GASAL2
Local (only score)	1	1	2.2	2.2	8.4	9.6
Local with start	2	1.9	4.4	3.3	16.8	13.6
Local with traceback	6	1.58	14	3.6	57.8	15.5
Semi-global (only score)	0.9	1	2	2.2	8	9.3
Semi-global with start	1.8	1.8	4	3.9	16	16
Semi-global with traceback	6	1.43	14	3.4	62	15
Global (only score)	0.9	1	2	2.3	8	9.5
Global with traceback	6	1.4	14	3.5	63	15

alignment kernel while performing all the one-to-one pairwise alignment between the sequences in the read and target set. These times do not include data packing and data copying time. Three different types of kernels are timed. The “only score” kernels only compute the score and end position. The “with start” kernels compute the score as well as start and end position without computing the traceback. There is no need to compute the start position for global alignment. The “with traceback” computes the actual alignment along with the score, start-position and end-position. The table shows that the alignment kernel execution times of NVBIO and GASAL2 are almost the same with and without computing the start-position. For finding the start-position GASAL2 kernel first finds the score and end-position. It then again aligns the two sequences in the backward direction beginning form the cell corresponding to the end-position. This backward alignment is halted as soon as its score reaches the previously calculated maximum score. This approach helps to reduce the number of DP cells need to be computed for finding the start-position. With traceback computation GASAL2 GPU kernels are around 4x faster than NVBIO. On the other hand, NVBIO is more space efficient and uses an approach similar to Myers-Miller algorithm [32] to compute the traceback.

Total execution time

In this section, we compare the performance of GASAL2 and other libraries in terms of the total execution time. The *total execution time* is the total time required to perform all the one-to-one pairwise alignment between the sequences in the read and target set. Figure 6 shows the flow chart of the test program used to measure the total execution time of the GASAL2. While filling the parameters we specify the type of alignment algorithm and one of the three following types of computations: 1) only score and end-position. 2) score, start and end-position without traceback. 3)score, end-position start-position and actual alignment in CIGAR format. Two batches of 500K sequences each are filled in each iteration. Hence, there are 20 iterations in total for the dataset of 10 million pair of sequences. GASAL2 initializes 5 CUDA streams and each stream performs one-to-one alignment of 100K pair of sequences. The total execution time of GASAL2 is the time starting from selecting an available stream till the time all the streams are completed i.e. allowing all the operations, from copying batches to copying results, to finish. Since the data transfer time is much smaller than the GPU alignment kernel time (at most 30% of kernel time) and GASAL2 uses CUDA streams, the data transfer is almost entirely overlapped with GPU execution. For the experiments, we are not reverse-complementing the sequences.

Local alignment

Figure 7 shows the total execution times computing only the score and end-position of the alignment. In this case GASAL2, NVBIO, ksw and Parasail are reporting the score as well as the end-position of the alignment. SeqAn only reports the alignment score. The execution times for SeqAn, ksw and Parasail shown in Fig. 7 are obtained with 56 CPU threads. For DS100, the figure shows that GASAL2 is 5.35x, 4.3x, 10x and 2x faster than ksw, Parasail, NVBIO and SeqAn, respectively. With DS150 the speedup of GASAL2 over ksw, Parasail, NVBIO and SeqAn is 4.75x, 3.6x, 7x and 2.4x, respectively. GASAL2 is 3.4x, 2.3x, 3.4x and 2.4x faster than ksw, Parasail, NVBIO and SeqAn, respectively for DS300. These results indicate that the speedup achieved by GASAL2 over ksw and Parasail decreases with longer reads. This is due to the fact that the ksw and Parasail use the striped heuristic that limits the computational complexity for longer reads, as compared to the GPU implementation. The results also show that the speedup achieved by GASAL2 compared to NVBIO decreases with longer reads. The reason for this decreasing speedup over NVBIO with increasing read lengths is the reduction of the data packing percentage (Fig. 3) as the alignment time continues to increase. GASAL2 speeds up the data packing while its alignment kernel performance remains similar to that of NVBIO.

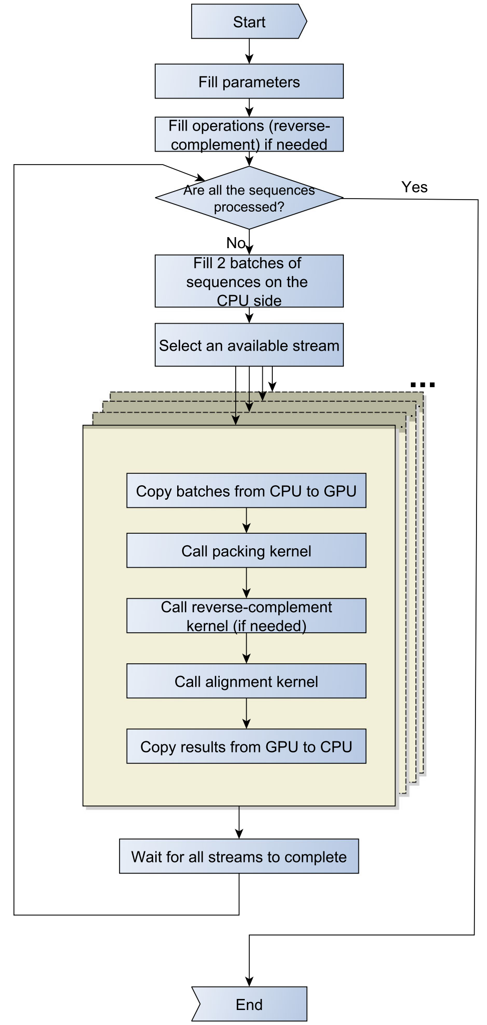

Start
Fill parameters
Fill operations (reverse-complement) if needed
Are all the sequences processed?
Yes
No
Fill 2 batches of sequences on the CPU side
Select an available stream
...
Copy batches from CPU to GPU
Call packing kernel
Call reverse-complement kernel (if needed)
Call alignment kernel
Copy results from GPU to CPU
Wait for all streams to complete
End

Fig. 6 The flow chart of the test program used to measure the total execution time of GASAL2

The speedup of GASAL2 over SeqAn remains constant around 2x with increasing read lengths. This is because both of them employ inter-sequence parallelization and use the standard DP algorithm having the complexity of $|S1| \times |S2|$ (Fig. 2). Hence, the execution time increases quadratically with read length for both GASAL2 and SeqAn.

Figure 8 shows the total execution time computing the start-position of the alignment without traceback. Since SeqAn neither reports the end-position nor the start-position, it is omitted in this comparison. The execution time values shown for ksw and Parasail are obtained with 56 CPU threads. The figure shows that GASAL2 is 6x, 5.3x and 4x faster than ksw; 4.8x, 3.7x and 2.4x faster than Prasail; 13x, 8.7x and 4.4x faster than NVBIO for DS100, DS150 and DS300 respectively. The reason for decreasing speedup of GASAL2 over CPU-based libraries is the same as described for local alignment without computing the start-position. The speedup over NVBIO is more in this case as compared to alignment without start-position computation. With start-position computation the packing time of NVBIO nearly doubles but the packing time of GASAL2 remains the same. Another interesting point to note is that the GASAL2 total execution time with start-position computation is smaller than the total alignment kernel time shown in Table 2. This happens because the alignment kernels of 5 batches are launched in parallel and their execution may overlap on GPU.

Figure 9 shows the total execution of the local alignment with traceback. The traceback computation gives the actual alignment between the pair of sequences along with the score, end-position and start-position. SeqAn and Parasail timings are obtained with 56 CPU threads. GASAL2 is 8.5x, 7.25x and 5x faster than NVBIO for DS100, DS150 and DS300, respectively. With increasing read lengths the data packing percentage in NVBIO decreases but the kernel speedup of GASAL2 over NVBIO remains constant (4x). The speedup of GASAL2 over SeqAn and Parasail is around 8x and 20X for all datasets.

Semi-global and global alignment

There are many types of semi-global alignments. All types of semi-global alignments are possible with GASAL2. SeqAn supports all types of semi-global alignments. Prasail support 10 types. NVBIO supports only one type. In the paper we are showing the results for semi-global alignment supported by all the libraries i.e. gaps at end and beginning of the read sequence are not penalized. The relative performance of GASAL2, Parasail and SeqAn for the remaining types is similar. Figure 10 shows the total execution time of semi-global alignment computing only the score and end-position. Like local alignment, SeqAn only reports the alignment score. Whereas, GASAL2, Prasail and NVBIO compute the alignment score as well as the end-position of the alignment. The execution times for SeqAn and Parasail are obtained with 56 CPU threads. GASAL2 is 4x, 10x and 1.7x faster than Parasail, NVBIO and SeqAn, respectively for DS100. For DS150 the speedup of GASAL2 over Parasail, NVBIO and SeqAn is 3.4x, 6.8x and 1.9x, respectively. In case of DS300 GASAL2 is 2.2x, 3.75x and 2x faster than Parasail, NVBIO and SeqAn, respectively. The reasons for decreasing speedup over Parasail and NVBIO with

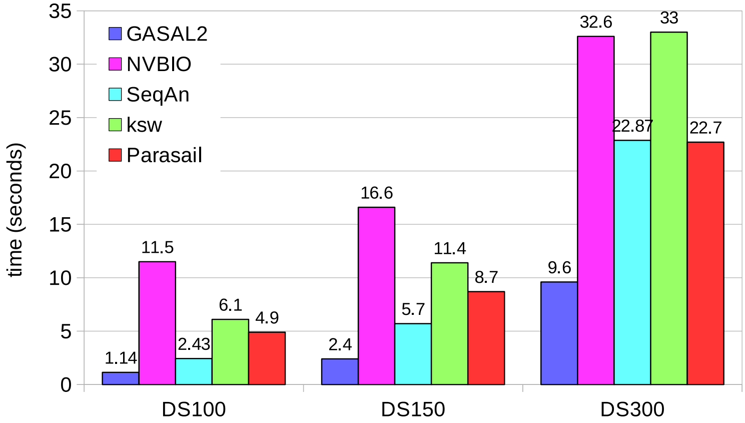


Fig. 7 Total execution times for local alignment computing only the score and end-position. The execution time of CPU-based libraries is obtained with 56 threads

increasing read lengths are the same as described for local alignment.

Figure 11 shows the total execution time of the semi-global alignment computing start-position without traceback. SeqAn does not compute the start-position, which is hence omitted in the comparison. The results for Parasail are obtained with 56 CPU threads. The figure shows that GASAL2 is 4.7x, 3.7x and 2.6x faster than Parasail and 13x, 8.4x and 4.4x faster than NVBIO for DS100, DS150 and DS300, respectively.

Figure 12 shows the total execution of the semi-global alignment with traceback. The speedups of GASAL2 over NVBIO and Parasail (56 CPU threads) are similar to local alignment. For SeqAn the fastest execution time for DS100 is obtained with 56 threads, whereas for DS150 and DS300 28 threads are faster than 56 threads. GASAL2 is 3x, 3.5x and 13.5x faster than SeqAn for DS100, DS150 and DS300 respectively.

Figure 13 and 14 shows the total execution time required for global alignment without and with traceback, respectively. The thread settings and the speedups achieved by GASAL2 are similar to that of semi-global alignment. With traceback computation GASAL2 becomes even more faster than other CPU libraries. For

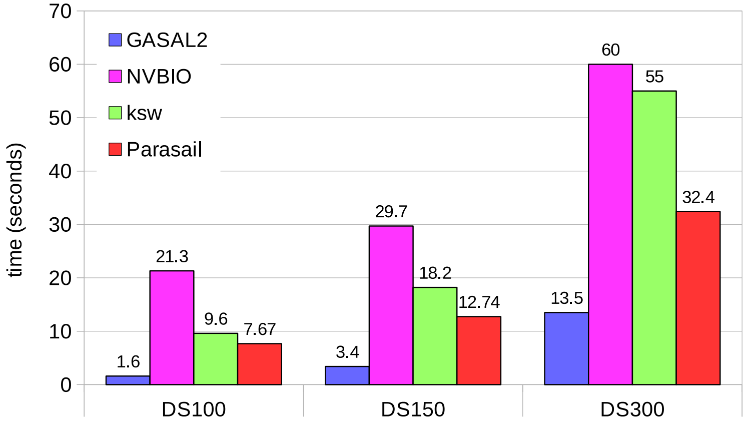


Fig. 8 Total execution times for local alignment computing start-position without traceback. The execution time of CPU-based libraries is obtained with 56 threads

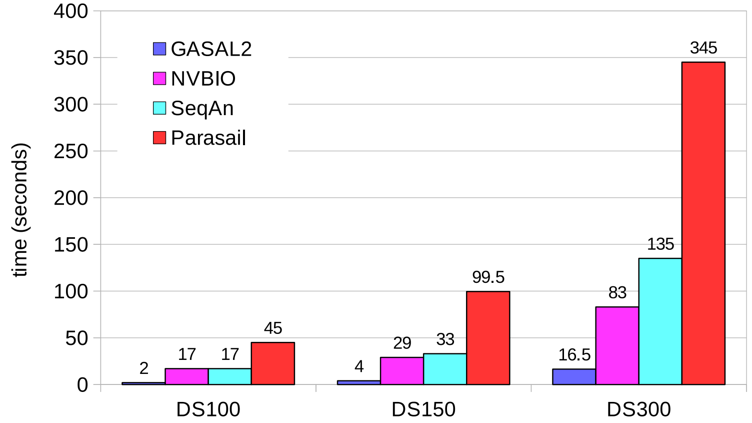


Fig. 9 Total execution times for local alignment with traceback computation. The execution time of CPU-based libraries is obtained with 56 threads

semi-global and global alignments with traceback the speedup of GASAL2 over SeqAn increases with increasing read lengths.

Discussion

GASAL2 is a GPU accelerated sequence alignment library. It can perform global alignment, local alignment and all types of semi-global alignment with and without traceback. It returns the alignment score, end-position and optionally the start-position of the alignment. It can also compute the second best local alignment score. Results show that GASAL2 is faster than NVBIO and state-of-the-art CPU-based SIMD libraries, making it a good choice for sequence alignment in high-throughput NGS data processing libraries. In the following, we show how to use the library to accelerate the BWA-MEM application.

Case Study:

BWA-MEM is a well known *seed-and-extend* DNA read mapper. In the seeding step, it finds substrtings of the read that match exactly somewhere in the reference genome. In the extension step, BWA-MEM tries to align the whole read around that match. The algorithm used in the extension step is similar to local alignment, where the start-position is also calculated. We accelerated BWA-MEM using GASAL2. Two paired-end read datasets of length

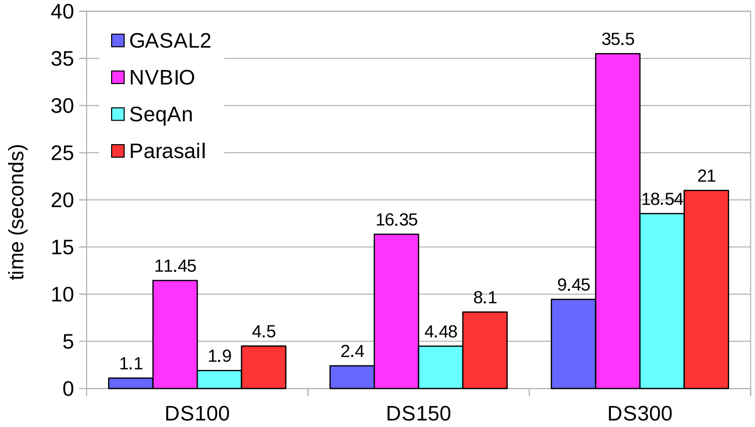


Fig. 10 Total execution times for semi-global alignment computing only the score and end-position. The execution time of CPU-based libraries is obtained with 56 threads

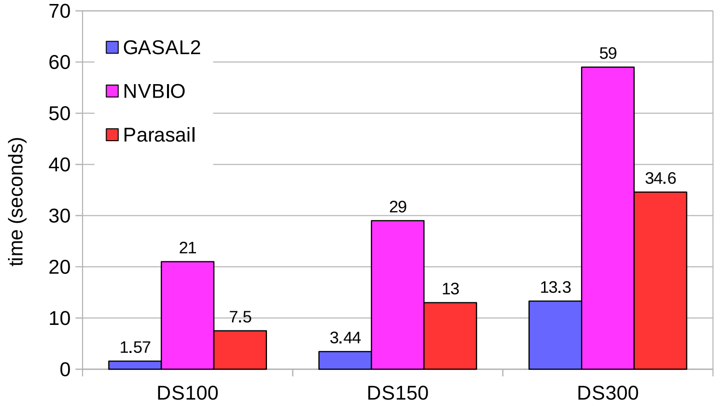


Fig. 11 Total execution times for semi-global alignment computing start-position without traceback. The execution time of CPU-based libraries is obtained with 56 threads

150 bp (SRR949537) and 250 bp (SRR835433) are used. The experiments are run on an NVIDIA Tesla K40c GPU. The GPU host machine has two 2.4GHz Intel Xeon E5-2620 v3 processors and 32 gigabytes of RAM. Each processor has six cores with 2-way hyper-threading. The BWA-MEM version used in this case study is 0.7.13. We also accelerated BWA-MEM using GASAL and compared it with the results obtained with GASAL2. The original GASAL published in [22] has two shortcomings described in "GASAL2 versus GASAL and NVBIO" section: a) GASAL treats base 'N' as an ordinary base. This causes BWA-MEM to abort due to an error. We updated GASAL so that it treats base 'N' in the same manner as GASAL2, b) GASAL allocates and de-allocates the GPU memory just before and after the memory transfers between CPU and GPU, respectively. This causes the whole BWA-MEM application to slow down substantially due to repetitive GPU memory allocations and de-allocations. We updated GASAL so that the memory allocation and de-allocation are performed same as in GASAL2 i.e. only once, at the beginning and end of the application. The accelerated BWA-MEM is executed in the same manner as the original BWA-MEM (same command line arguments). The only difference between the accelerated BWA-MEM and the original version is that the seed-extension is performed on the GPU instead of CPU.

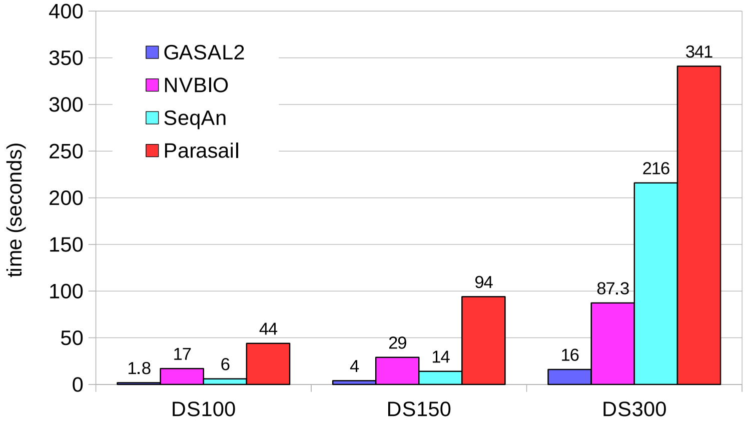


Fig. 12 Total execution times for semi-global alignment with traceback computation. The execution time of CPU-based libraries is obtained with 56 threads except of SeqAn. For SeqAn the DS100 results are with 56 threads, whereas the DS150 and DS300 results are with 28 threads

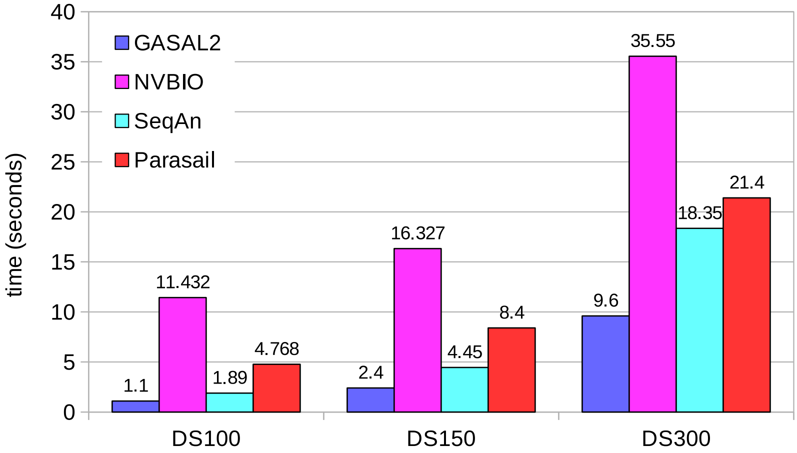


Fig. 13 Total execution times for global alignment without traceback. The execution time of CPU-based libraries is obtained with 56 threads

Execution timeline

Figure 15 shows the execution timeline of BWA-MEM before and after acceleration. Figure 15a shows the execution in the original BWA-MEM. Figure 15b shows the BWA-MEM execution with the extension step accelerated using GASAL. Note that the seeding and extension steps are performed for a batch of reads to mitigate the CPU-GPU memory transfer overhead and to fully utilize GPU resources. Furthermore, the thread running on the CPU remains idle while the extension is performed on the GPU. Figure 15c shows how the GASAL2 alignment function can be used for overlapping CPU and GPU execution. A batch of reads is further broken down into *sub-batches*, numbered 1, 2 and 3. CPU execution is overlapped with the seed extension on GPU. This is achieved via the GASAL2 asynchrnous alignment function call facility. Empty time slots on the CPU timeline are also present in (c), but these are much smaller than (b). These empty slots in (c) will not be present if extension on GPU is faster than post-extension processing or vice-versa. We test both approaches i.e. (b) and (c), to accelerate the extension step of BWA-MEM. In practice, due to load balancing (explained below) we used a batch size that varies between 5000 to 800 reads. The number of sub-batches are either 5 or 4.

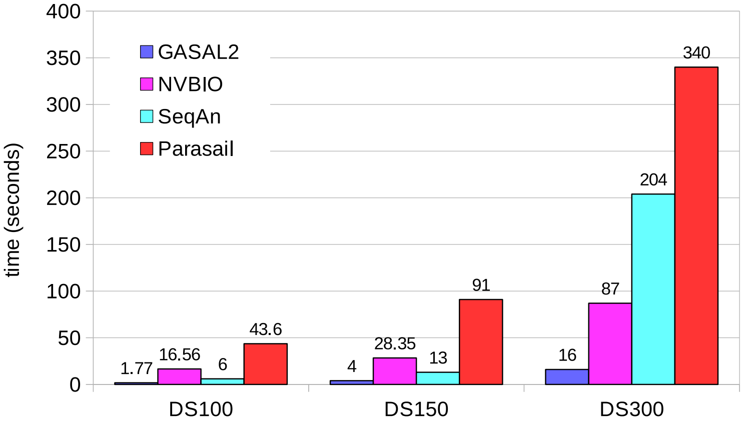


Fig. 14 Total execution times for global alignment with traceback computation. The execution time of CPU-based libraries is obtained with 56 threads except for SeqAn. For SeqAn the DS100 results are with 56 threads, whereas the DS150 and DS300 results are with 28 threads

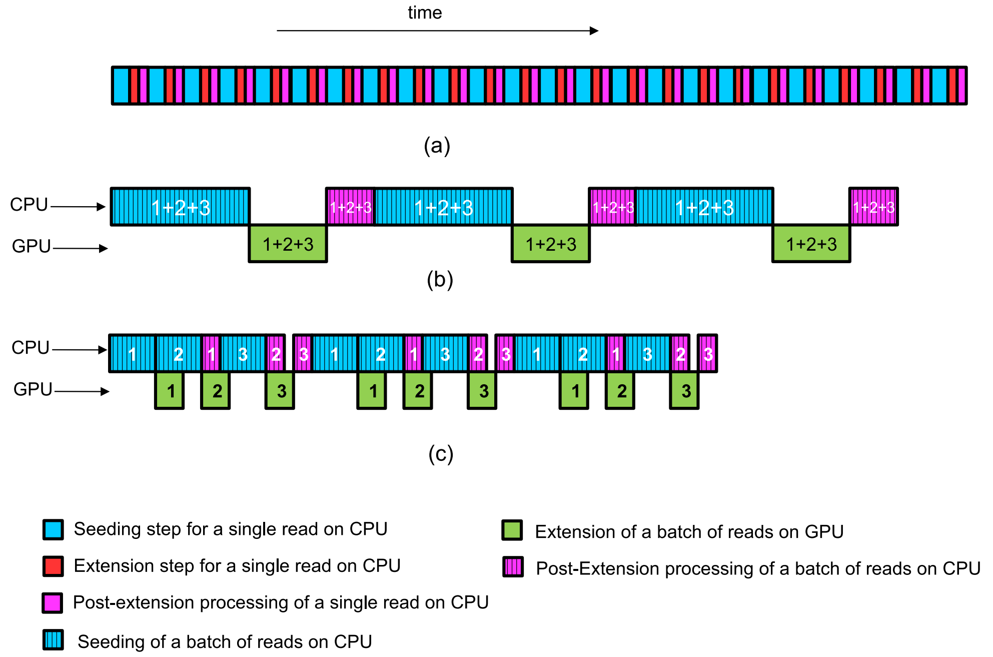

time
(a)
CPU
GPU
1+2+3
(b)
CPU
GPU
1
2
3
(c)
Seeding step for a single read on CPU
Extension step for a single read on CPU
Post-extension processing of a single read on CPU
Seeding of a batch of reads on CPU
Extension of a batch of reads on GPU
Post-Extension processing of a batch of reads on CPU

Fig. 15 Execution timeline of original and accelerated BWA-MEM

Load balancing

In the original BWA-MEM, each thread is assigned a number of reads to process and one read is processed by a thread at a time. If a thread has finished processing all of its allocated reads, it will process the remaining reads of unfinished threads. Due to this, all of the threads remain busy until the whole data is processed resulting in maximum CPU utilization. On the other hand, in case of GPU acceleration reads are processed in the form of batches. Therefore, some threads may finish earlier than others and remain idle while waiting for all of the threads to finish. The idle time of these threads causes underutilization of the CPU. Decreasing the batch size helps to increase the CPU utilization, but at the same time may reduce the alignment function speedup due to increased data transfer overhead and poor GPU utilization. To circumvent this problem, we used dynamic batch sizes in our implementation. At the start, the batch size for each CPU thread is 5000 reads, but can be reduced to as low as 800 reads, depending upon the number of free threads which have finished processing there allocated reads. Doing so help to reduce the time wasted by a CPU thread in waiting for other threads to finish. We measured the *wasted time* as the difference between the finishing times of slowest and the fastest thread. By applying our dynamic batch size approach the wasted time is reduced by 3x for 150bp reads and 2x for 250 bp reads with 12 CPU threads.

Performance with 150bp reads

For 150bp reads, Fig. 16 shows the comparison of time spent in the seed extension for the original BWA-MEM executed on the host CPU and the GPU accelerated BWA-MEM in which the seed extension is performed using GASAL2 alignment functions. The extension performed using GASAL2 (GASAL2-extend) is the sum of time to asynchronously call the GASAL2 alignment function and the time required in getting back the results using `gasal_is_aln_async_done()` function, in addition to the time of the empty slots before the post-processing of the last sub-batch. GASAL2-extend is more than 42x faster than the CPU time represented by original BWA-MEM extension function(orig-extend) for one thread, and over 20x faster for 12 CPU threads. Hence, the GASAL2 asynchronous alignment function allows to completely eliminate the seed extension time. The GASAL alignment function (GASAL-extend) is 3-4x slower than GASAL2-extend and is hence around 7-10x fassimilarter than orig-extend.

Figure 17 shows the total execution times of the original BWA-MEM and GASAL2 for 150bp data. The *ideal-total*

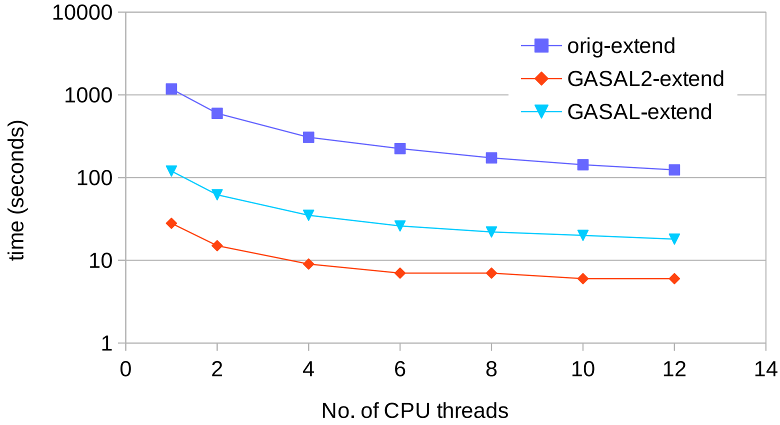


Fig. 16 Time spent in the extension step of BWA-MEM for 150bp reads

is total execution time for the case in which the time spent in the extension step is zero, and thus, represents the maximum achievable speedup. For 1 to 4 CPU thread, the GPU speedup is almost identical to the ideal one. For higher CPU threads, the speedup is slightly smaller than ideal. For 12 threads, the GASAL2 speedup and ideal speedup are 1.3 and 1.36, respectively. Since the time consumed by the seed extension function in BWA-MEM is 25-27%, the total execution time of GASAL is only slightly higher than GASAL2. For 12 threads, the GASAL speedup is 1.26. The main cause of the difference between ideal and actual speedup for higher number of CPU threads is imperfect load balancing between the CPU threads.

Performance with 250 bp reads

Same analysis is repeated for 250 bp reads. Figure 18 shows the seed extension time of original BWA-MEM and GASAL2 alignment functions. GASAL2-extend is 32x to 14x faster than orig-extend for 1 to 12 CPU threads, respectively. The reduction in speed-up as compared to 150bp reads is due to reduction in GPU alignment kernel speed for longer reads, which widens the empty slots in the CPU timeline of Fig. 15c. GASAL-extend is 7x to 3x faster than CPU extension for 1 to 12 CPU threads, respectively. This means that GASAL-extend is 4-5x slower than GASAL2-extend. Hence, for longer reads the speedup of GASAL2 over GASAL increases.

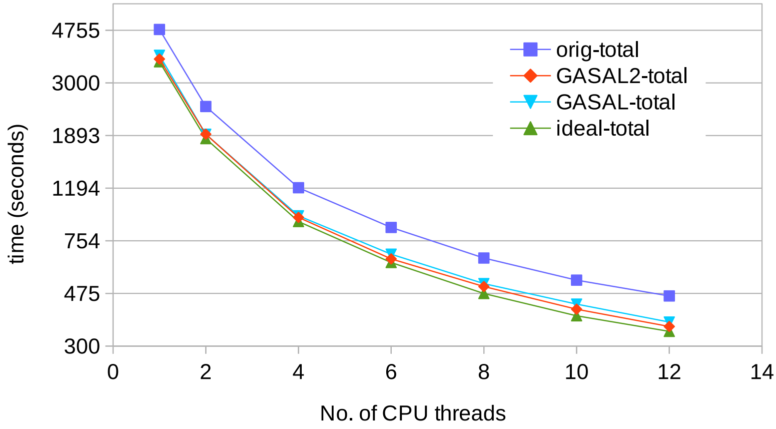


Fig. 17 Total execution time of BWA-MEM for 150 bp reads

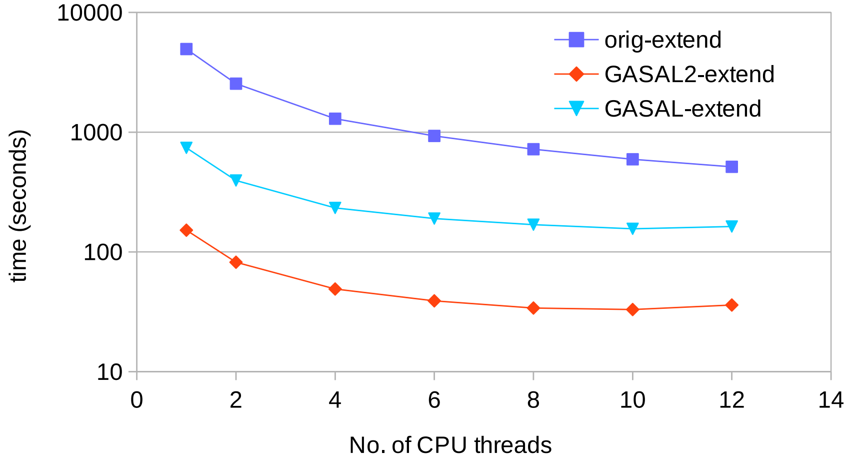


Fig. 18 Time spent in the extension step of BWA-MEM for 250bp reads

Figure 19 shows the total execution time for 250 bp reads. For up to 2 CPU threads, GASAL2-total, GASAL-total and ideal-total all are the same. Above 2 CPU threads, GASAL2-total becomes faster than GASAL-total. For 12 CPU threads, the ideal speedup is 1.49 whereas the speedup with GASAL2 and GASAL is 1.35 and 1.2, respectively. The gap between the ideal speedup and speedup achieved with GASAL2 is larger for 250 bp reads as compared to 150 bp reads. This happened due to imperfect load balancing between threads as well as decreased speedup of the seed extension step for 250bp reads.

In summary GASAL2 gives seed-extension speedup in excess of 10x even when 12 CPU threads share a single NVIDIA Tesla K40c GPU.

Conclusions

In this paper, we presented GASAL2, a high performance and GPU accelerated library, for pairwise sequence alignment of DNA and RNA sequences. The GASAL2 library provides accelerated kernels for local, global as well as semi-global alignment, allowing the computation of the alignment with and without traceback. It can also compute the start position without traceback. In addition, one-to-one as well as all-to-all and one-to-many pairwise alignments can be performed. GASAL2 uses the novel approach of also performing the sequence packing on GPU, which is over 750x faster than the NVBIO approach. GASAL2 alignment functions are asynchronous/non-blocking which allow fully overlapping CPU and GPU execution. GASAL2 can compute all types of semi-global

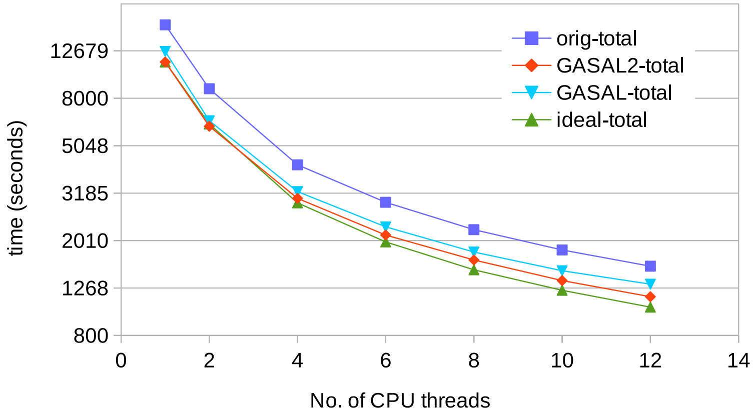


Fig. 19 Total execution time of BWA-MEM for 250 bp reads

alignments. These represent unique capabilities not available in any earlier GPU sequence alignment library. The paper compared GASAL2's performance with the fastest CPU-optimized SIMD implementations such as SeqAn, ksw, Parasail and NVBIO (NVIDIA's own GPU library for sequence analysis of high-throughput sequencing data). Experimental results performed on the Geforce GTX 1080 Ti GPU show that GASAL2 is up to 5.35x faster than 56 Intel Xeon threads and up to 10x faster than NVBIO with a read length of 100bp, computing only the score and end-position. For 150bp reads, the speedup of GASAL2 over CPU implementations (56 Intel Xeon threads) and NVBIO is up to 4.75x and up to 7x, respectively. With 300bp reads, GASAL2 is up to 3.4x faster than CPU (56 Intel Xeon threads) and NVBIO. The speedup of GASAL2 over CPU implementations (56 Intel Xeon threads) computing start-position without traceback is up to 6x, 5.3x and 4x for 100, 150 and 300bp reads, respectively. With start-position computation, the speedup of GASAL2 over NVBIO is up to 13x, 8.7x and 4.4x for 100, 150 and 300bp reads, respectively. With traceback computation GASAL2 becomes even faster. GASAL2 traceback alignment is 13x and 20x faster than SeqAn and Parasail for read lengths of up to 300 bases. The GPU traceback alignment kernel of GASAL2 is 4x faster than NVBIO's kernel, giving an overall speedup of 9x, 7x and 5x for 100, 150 and 300bp reads, respectively. GASAL2 is used to accelerate the seed extension function of BWA-MEM DNA read mapper. It is more than 20x faster than the CPU seed extension functions with 12 CPU threads. This allows us to achieve nearly ideal speedup for 150 bp reads. The library provides easy to use APIs to allow integration into various bioinformatics tools. GASAL2 is publicly available and can be downloaded from: https://github.com/nahmedraja/GASAL2.

Availability and requirements

Project name: GASAL2- GPU Accelerated Sequence Alignment Library.

Project home page: https://github.com/nahmedraja/GASAL2

Operating system(s): Linux

Programming language: C++, CUDA

Other requirements: CUDA toolkit version 8 or higher.

License: Apache 2.0

Any restrictions to use by non-academics: Not applicable

Abbreviations

AVX2: Advanced vector extensions version-2; CPU: Central processing unit; CUDA:Compute unified device architecture; GPU: Graphical processing unit; NGS: Next generation sequencing; SIMD: Single instruction multiple data; SM: Streaming multiprocessor; SP: Streaming processor; SSE: Streaming SIMD extensions

Authors' contributions

NA carried out the study, developed the library, performed the experiments and wrote the manuscript. JL contributed in the library development and writing the manuscript. SR and HM provide useful suggestions and revised the manuscript. KB headed the research and revised the manuscript. ZA proposed the idea, supervised the research and helped in writing the manuscript. All authors read and approved the final manuscript.

Funding

Not applicable.

Availability of data and materials

Not applicable.

Ethics approval and consent to participate

Not applicable.

Consent for publication

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Author details

[1]Delft University of Technology, Delft, Netherlands and University of Engineering and Technology, Lahore, Pakistan. [2]Delft University of Technology, Netherlands, Delft, Netherlands. [3]Maastricht UMC+, Netherlands, Maastricht, Netherlands.

Received: 2 April 2019 Accepted: 6 September 2019

Published online: 25 October 2019

References

1. Li H. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. arXiv. 2013.
2. Langmead B, S S. Fast gapped-read alignment with Bowtie 2. Nat Methods. 2012;9:357–59.
3. Huang X, Yang S-P. Generating a Genome Assembly with PCAP. 2002.
4. de la Bastide M, McCombie WR. Assembling Genomic DNA Sequences with PHRAP. 2002.
5. Salmela L, Schröder J. Correcting errors in short reads by multiple alignments. Bioinformatics. 2011;27(11):1455–61.
6. Kao W-C, Chan AH, Song YS. ECHO: a reference-free short-read error correction algorithm. Genome Res. 2011;21(7):1181–92.
7. Poplin R, et al. Scaling accurate genetic variant discovery to tens of thousands of samples. bioRxiv. 2017.
8. Needleman SB, Wunsch CD. A general method applicable to the search for similarities in the amino acid sequence of two proteins. J Mole Biol. 1970;48(3):443–53.
9. Smith TF, Waterman MS. Identification of common molecular subsequences. J Mole Biol. 1981;147(1):195–7.
10. Gotoh O. An improved algorithm for matching biological sequences. J Mole Biol. 1982;162(3):705–8.
11. Liu Y, Huang W, Johnson J, Vaidya S. In: Alexandrov VN, van Albada GD, Sloot PMA, Dongarra J, editors. GPU Accelerated Smith-Waterman. Berlin, Heidelberg: Springer; 2006, pp. 188–95.
12. Liu Y, Wirawan A, Schmidt B. CUDASW++ 3.0: Accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. BMC Bioinformatics. 2013;14(1):117.
13. Hasan L, Kentie M, Al-Ars Z. DOPA: GPU-based protein alignment using database and memory access optimizations. BMC Res Notes. 2011;4(1): 261.
14. Ren S, Bertels K, Al-Ars Z. Efficient Acceleration of the Pair-HMMs Forward Algorithm for GATK HaplotypeCaller on Graphics Processing Units. Evol Bioinforma. 2018;14:1176934318760543.
15. Ren S, Ahmed N, Bertels K, Al-Ars Z. An Efficient GPU-Based de Bruijn Graph Construction Algorithm for Micro-Assembly. In: 2018 IEEE 18th International Conference on Bioinformatics and Bioengineering (BIBE); 2018. p. 67–72.

16. Kalaiselvi T, Sriramakrishnan P, Somasundaram K. Survey of using gpu cuda programming model in medical image analysis. Informa Med Unlocked. 2017;9:133–44.
17. Sriramakrishnan P, Kalaiselvi T, Rajeswaran R. Modified local ternary patterns technique for brain tumour segmentation and volume estimation from mri multi-sequence scans with gpu cuda machine. Biocyber Biomed Eng. 2019;39(2):470–87.
18. Bhosale P, Staring M, Al-Ars Z, Berendsen FF. GPU-based stochastic-gradient optimization for non-rigid medical image registration in time-critical applications. In: SPIE Medical Imaging 2018; 2018.
19. Blazewicz J, Frohmberg W, Kierzynka M, Pesch E, Wojciechowski P. Protein alignment algorithms with an efficient backtracking routine on multiple GPUs. BMC Bioinformatics. 2011;12(1):181.
20. Liu Y, Schmidt B. GSWABE: faster GPU-accelerated sequence alignment with optimal alignment retrieval for short DNA sequences. Concurr Comput: Pract Exp. 2015;27(4):958–72.
21. Pantaleoni J, Subtil N. NVBIO. 2015. https://nvlabs.github.io/nvbio. Accessed 1 October, 2017.
22. Ahmed N, Mushtaq H, Bertels K, Al-Ars Z. GPU accelerated API for alignment of genomics sequencing data. In: 2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM); 2017. p. 510–5.
23. Li H. wgsim: Reads simulator. https://github.com/lh3/wgsim. Accessed 1 October, 2017.
24. Ehrhardt M, Rahn R, Reinert K, Budach S, Costanza P, Hancox J. Generic accelerated sequence alignment in SeqAn using vectorization and multi-threading. Bioinformatics. 2018;34(20):3437–45.
25. R R. DP Bench - A benchmark tool for SeqAn's alignment engine.
26. Chaos A. Klib: a Generic Library in C. https://github.com/attractivechaos/klib. Accessed 2 January, 2019.
27. Zhao M, et al. SSW library: An SIMD smith-waterman c/c++ library for use in genomic applications. PLoS ONE. 2013;8.
28. Daily J. Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. BMC Bioinformatics. 2016;17(1):1–11.
29. Šošić M, Šikić M. Edlib: a C/C++ library for fast, exact sequence alignment using edit distance. Bioinformatics. 2017;33(9):1394–5.
30. Benson G, Loving J, Hernandez Y. BitPAl: a bit-parallel, general integer-scoring sequence alignment algorithm. Bioinformatics. 2014;30(22):3166–73.
31. Lan H, Zhang J, Chan Y, Liu W, Shang Y, Schmidt B. BGSA: a bit-parallel global sequence alignment toolkit for multi-core and many-core architectures. 2018.
32. Myers EW, Miller W. Optimal alignments in linear space. Bioinformatics. 1988;4(1):11–7.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Ready to submit your research? Choose BMC and benefit from:

- fast, convenient online submission
- thorough peer review by experienced researchers in your field
- rapid publication on acceptance
- support for research data, including large and complex data types
- gold Open Access which fosters wider collaboration and increased citations
- maximum visibility for your research: over 100M website views per year

At BMC, research is always in progress.

Learn more biomedcentral.com/submissions

BMC

SOFTWARE

GPU Acceleration of Darwin Read Overlapper for de Novo Assembly of Long DNA Reads

Nauman Ahmed[1,2*], Tong Dong Qiu[1], Koen Bertels[1] and Zaid Al-Ars[1]

Abstract

Background: In Overlap-Layout-Consensus (OLC) based de novo assembly, all reads must be compared with every other read to find overlaps. This makes the process rather slow and limits the practicality of using de novo assembly methods at a large scale in the field. Darwin is a fast and accurate read overlapper that can be used for de novo assembly of state-of-the-art third generation long DNA reads. Darwin is designed to be hardware-friendly and can be accelerated on specialized computer system hardware to achieve higher performance.

Results: This work accelerates Darwin on GPUs. Using real Pacbio data, our GPU implementation on Tesla K40 has shown a speedup of 109x vs 8 CPU threads of an Intel Xeon machine and 24x vs 64 threads of IBM Power8 machine. The GPU implementation supports both linear and affine gap, scoring model. The results show that the GPU implementation can achieve the same high speedup for different scoring schemes.

Conclusions: The GPU implementation proposed in this work shows significant improvement in performance compared to the CPU version, thereby making it accessible for utilization as a practical read overlapper in a DNA assembly pipeline. Furthermore, our GPU acceleration can also be used for performing fast Smith-Waterman alignment between long DNA reads. GPU hardware has become commonly available in the field today, making the proposed acceleration accessible to a larger public.

Keywords: Genomics; Read overlapper; de Novo Assembly; Long DNA reads; GPU acceleration

Introduction

DNA sequencing techniques used today produce short pieces of data (called reads) that represent parts of the sampled DNA, possibly containing some errors. The length and error rate of these reads depends on the sequencing technique used. DNA assembly tries to combine the reads into larger, more accurate DNA segments. For these DNA reads, graph-based assemblers are used for the assembly process, which comes in two flavors: Overlap-Layout-Consensus (OLC) and de Bruijn Graph (dBG).

The OLC assemblers [1] first find overlaps and build an overlap graph. Each node represents a read, and each edge represents an overlap between two reads. During the layout phase, the graph is analyzed to find paths, corresponding to segments of the original genome. The perfect graph contains one path that visits each node exactly once. This problem can be described as finding a Hamiltonian Path. A Hamiltonian Path includes all vertices of a graph exactly once. Examples of assemblers that use the OLC approach are Dazzler [2] and SGA [3].

In dBG based assemblers [4], each read is divided into K-*mers*. A K-*mer* is a substring of a read having a length K. Each K-*mer* represents a directed edge between two vertices, where the source vertex represents the first $K-1$ bases of the K-*mer*, and the destination vertex the last $K-1$ bases of the K-*mer*. When a particular $K-1$ vertex does not exist, it is created, otherwise, the existing one is reused. The weights of the edges indicate how many times a particular K-*mer* is encountered. The next step is to find an Eulerian Path, which is a path that includes all edges of a graph exactly once. Examples of dBG assemblers are Velvet [5], ABySS [6] and SOAPdenovo2 [7].

So called Next Generation Sequencing (NGS) techniques produce reads with lengths anywhere from 50 to 500 base pairs. They can be produced at high throughput but at the expense of a smaller read length. However, DNA can contain repeat regions, where a certain piece of DNA is repeated many times back-to-back, or

---

[*] Correspondence: n.ahmed@tudelft.nl
[1] Delft University of Technology, Delft, Netherlands
[2] University of Engineering and Technology Lahore, Lahore, Pakistan
Full list of author information is available at the end of the article

a repeat could appear in many different places in the genome. Since these repeats can be longer than the produced short reads, this means the reads cannot be used to resolve these repeat regions. Third generation sequencing produces much longer reads, of up to 60K base pairs. Due to their length, these reads are more likely to contain a whole repeat region, which makes them suitable for accurately reconstructing the repeat regions. A major drawback of longer DNA reads is their higher error rate, ranging from 15-30%, depending on the exact sequencing technology. dBG based assembly is the more preferred approach for NGS reads, which are much shorter and have much lower error rates. However, de Bruijn Graphs are quite susceptible to sequencing errors, since one substituted base pair causes K incorrect K-$mers$. Pair this with an often-used values of K above 50 and third generation sequencing error rate of about 15%, it is clear that the graph will contain a lot of incorrect edges. Therefore, OLC based assemblers are more suitable for state-of-the-art third generation long DNA reads produced by Pacbio and Oxford Nanopore sequencers.

Darwin [8] is a read overlapper for the assembly of long DNA reads. Darwin is designed to be highly accurate, achieving a sensitivity of 99.89% and a precision of 88.30% for simulated Pacbio reads. This is higher than other commonly used read overlappers such as Daligner [2]. The ASIC (Application-Specific Integrated Circuit) implementation of Darwin is shown to be hundreds of times faster than other software based overlappers. However, ASIC implementation requires bulk volume production to be economically feasible. Moreover, DNA analysis using high-throughput DNA sequencing is an evolving field, and any major improvement in the algorithm will require a new ASIC implementation which costs both time and money.

Heterogeneous systems with GPU accelerators have become easily accessible due to their widespread use. They have shown convincing speedups in many high performance computing applications. In this paper, we present a GPU accelerated version of Darwin. We identified the computational bottleneck in the Darwin software and replaced it with the GPU accelerated version. The accelerated implementation proposed in this paper is orders of magnitude faster than its software counterpart. The contributions of the paper are as follows:

- The paper shows the GPU implementation of the Darwin read overlapper used in the de novo assembly of long DNA reads.
- The paper shows that the GPU acceleration of Darwin is orders of magnitude faster than the multithreaded software version on both IBM Power8 and Intel Xeon machines using a real Pacbio dataset.
- The results in the paper show that the GPU implementation of Darwin can also be applied for accelerating Smith-Waterman alignment of long DNA reads.

Background

Smith-Waterman (SW) [9] algorithm finds local alignment between a pair of sequences. Smith-Waterman is exact, producing the optimal local alignment. It can be implemented using dynamic programming which computes a 2D matrix S. Let V and W be the two sequences to be aligned. Let $V_0, V1, \ldots, V_{|V|-1}$ and $W_0, W1, \ldots, W_{|W|-1}$ be the bases of V and W, respectively. $|V|$ and $|W|$ are the lengths of V and W. $S(i,-1) = S(-1,j) = 0$ for $i = 0, 1, 2 \ldots, |V| - 1$ and $j = 0, 1, 2 \ldots, |W| - 1$. The cells in the matrix are computed using the following recurrence relation:

$$S(i,j) = max \begin{cases} S(i-1,j) + gap \\ S(i,j-1) + gap \\ S(i-1,j-1) + subt(V_i, W_j) \\ 0 \end{cases} \tag{1}$$

$$m_{i,j} = \begin{cases} (i,j) & S(i,j) > m \\ m_{i,j} & S(i,j) \leq m \end{cases} \tag{2}$$

$$m = max \begin{cases} m \\ S(i,j) \end{cases} \tag{3}$$

$$D(i,j) = \begin{cases} 0 & S(i,j) = 0 \\ \uparrow & S(i,j) = S(i-1,j) \\ \leftarrow & S(i,j) = S(i,j-1) \\ \nwarrow & S(i,j) = S(i-1,j-1) + subt(V_i, W_j) \end{cases} \tag{4}$$

Here, S and D are the score and traceback matrices, respectively. $match$, $mismatch$ and gap are numeric parameters. $subt(V_i, W_j)$ is equal to $match$ if $V_i = W_j$, and is equal to $mismatch$ otherwise. gap is the penalty for inserting a gap. m is the alignment score, which is initialized to zero, and $m_{i,j}$ is the corresponding position on V and W. The traceback matrix is required to compute the actual alignment. Traceback starts from the highest scoring cell and follows the arrows in D until a zero or a boundary of the matrix is encountered. Equations 1 and 3 indicate that for computing the alignment score m there is no need to store the whole S matrix as all cells of the S matrix are computed using only the values of three other cells $S(i-1,j)$, $S(i,j-1)$ and $S(i-1,j-1)$. Hence, to compute the alignment score, storing only the values in the previous row and column are sufficient to compute m. The above equations are for calculating the alignment with a linear-gap scoring model. However, Darwin and our GPU implementation also support the more commonly used affine gap penalty model in which there are separate penalties for opening a gap (*gapo*) and extending a gap (*gape*).

A straightforward way of finding all overlaps is performing an alignment algorithm, such as Smith-Waterman, on every pair of reads. The number of alignments is quadratic with the number of reads, and the runtime of one alignment is quadratic with the lengths of the involved reads, making this method not feasible. Many heuristic algorithms have been developed to perform this alignment, for different lengths and error rates. Seed-and-extend is one heuristic, which dramatically reduces the amount of computation needed [10]. A seed is a K-*mer* made up of K consecutive bases of a read. Instead of performing Smith-Waterman on each read pair, only read pairs that have one or more common K-*mers* are aligned. A common K-*mer* between two or more reads is known as a “seed hit”. Darwin also uses the seed-and-extend approach, which reduces the amount of computation needed, without compromising the output by much. Other algorithms, like BLAST [11], also use the seed-and-extend approach, but give sub-optimal alignments. Results in [8] show that Darwin provides optimal Smith-Waterman alignments between long DNA sequences with error rates up to 40%.

Darwin

Darwin is read overlapping algorithm for de novo assembly of third-generation long DNA reads. It is based on the seed-and-extend. It consists of a filter called D-SOFT (Diagonal-band Seed Overlapping based Filtration Technique), which finds seed hits, and GACT (Genome Alignment using Constant memory Traceback), which extends the seed hit by performing sequence alignment between the sequences on the left and right of the seed hit. Figure 1 shows the seed-and-extend technique employed in Darwin to find the overlap between *Read A* and *Read B*. To compute the overlap, a seed hit is extended on both sides by aligning *R_left* with *Q_left* and *R_right* with *Q_right*. This speedups the computation by avoiding the computation of a large number of dynamic programming matrix cells (grey cells in Figure 1).The dynamic programming matrix computed to align *R_left* with *Q_left* is known as *left extension matrix*. Similarly, the dynamic programming matrix computed to align *R_right* with *Q_right* is known as *right extension matrix*.

D-SOFT

D-SOFT is the seeding stage of Darwin, also known as the *filtering* stage. Darwin uses minimizers [12] as seeds which are K-*mers* extracted from all the reads to be overlapped. The position of a seed is stored in a minimizer table which records the location of the seed in a read along with the identifier of the read. The window size w is the most important parameter for building a minimizer table and must be smaller than the seed length (K). To obtain seed hits, ‘N’ K-*mers* of a read are used as seeds that are located in other reads using the minimizer table. Two reads are considered for alignment in the extension phase if they have at least h unique bases in common. The pair of reads passing this filter are aligned using a modified Smith-Waterman algorithm described below.

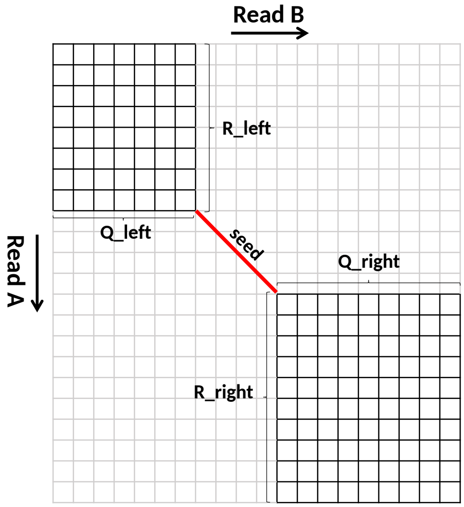


Figure 1 The seed-and-extend method used in Darwin to find the overlap between *Read A* and *Read B*. *R_left* and *R_right* are the substrings of Read A. *Q_left* and *Q_right* are the substrings of Read B.

GACT

The seed-and-extend approach to find overlap between two reads is much faster than performing the complete Smith-Waterman alignment algorithm. However, the reduced dynamic programming matrices could still be quite large. State-of-the-art third generation sequencers produce reads having lengths in megabases and the dynamic programming matrix in the seed extension may have around 1 Tera cells to compute. For example, if the seed hit lies at the beginning of the two reads, i.e. near the top left corner in Figure 1, the right extension matrix is nearly as large as the full matrix.

Listing 1 GACT algorithm

```
tb_left = []
(i_curr, j_curr) = (i_seed, j_seed)
t = 1
while (i_curr > 0 and j_curr > 0)
{
   (i_start, j_start) = (Max(0, i_curr - T), Max(0, j_curr - T))
   (R_tile, Q_tile) = (R[i_start : i_curr], Q[i_start : i_curr])
   (i_off, j_off, i_max, j_max, tb) = Align(R_tile, Q_tile, t, T-O)
   tb_left.Prepend(tb)
   if (t == 1){
      (i_curr, j_curr) = (i_max, j_max)
      t = 0
   }
   if (i_off == 0 and j_off == 0) {
       break;
   }
   else {
       (i_curr, j_curr) = (i_curr - i_off, j_curr - j_off)
   }
}
return (i_max, j_max, tb_left)
```

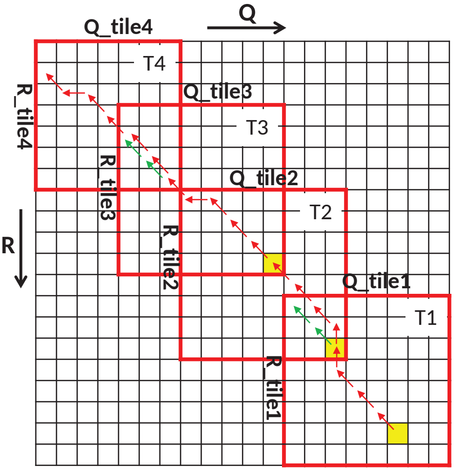


Figure 2 An example of the GACT algorithm.

Numerous efforts to accelerate Smith-Waterman have been made, both by using hardware like [13] and software [14]. But the memory required to store the traceback matrix D is still an issue. One can apply the Hirschberg's algorithm described in [15] to reduce the RAM storage but at the cost of increase in computation time. Therefore, Darwin proposed the GACT algorithm for seed extension. It has two advantages: 1) All the cells of the right and left extension matrix are not computed reducing the computation time. 2) The traceback matrix is very small. GACT performs normal Smith-Waterman on a submatrix of the extension matrix, known as *tiles* of size TxT. After computing a tile it computes the next tile, which overlaps the previous tile with at least O cells on both axes. For reasonable values for T and O, GACT has shown to produce the same result as normal Smith-Waterman [8]. Figure 2 shows an example of computing the extension matrix with the GACT algorithm. In the example $T = 8$ and $O = 3$. The tiles are computed in the order $T1, T2, T3$ and $T4$. The example in Figure 2 can be used to explain both the computation of left and right extension matrices. The only difference is $R = \overline{R\_right}$ and $Q = \overline{Q\_right}$ in case of right extension, where $\overline{R\_right}$ and $\overline{Q\_right}$ is the reverse of R_right and Q_right sequences, respectively. Listing 1, shows the algorithm for left extension. Positions i_curr and j_curr are produced by D-SOFT. The start and end position of the current tile are stored in (i_start, j_start) and (i_curr, j_curr), respectively. The traceback path of the whole left extension is kept in *tb_left*. The function $Align()$ uses Smith-Waterman to compute traceback matrix D between subsequences *R_tile* and *Q_tile*. Once the traceback matrix is filled, traceback is performed starting from the bottom-right cell, except for the first tile, where traceback starts from the highest-scoring cell. The starting cells of the traceback are coloured yellow in Figure 2. $Align()$ returns the number of bases in R and Q aligned by this tile (*i_off*,*j_off*), the traceback arrows/pointers (tb) and the position of the highest-scoring cell (*i_max*,*j_max*). $Align()$ also limits *i_off* and *j_off* to at most $T - O$ bases, to ensure the next tile overlaps by at least O bases on both R and Q. The green arrows shows the path taken by the traceback in a tile if there is no limit and the traceback is allowed to complete. The left extension finishes when it hits the end of R or Q, or when traceback cannot add any bases to the existing alignment. The memory needed for the traceback is $\mathcal{O}(T^2)$, which is constant since T is chosen upfront. The whole alignment of the extension is contained in *tb_left* and is equivalent to the path traced by the red arrows in Figure 2. The alignment score of the extension can also be computed with the help of *tb_left* The right extension operates on the reverse of R and Q.

The performance of GACT is linear ($O(max\{|ReadA|, |ReadB|\} \cdot T)$) where $|ReadA|$ and $|ReadB|$ are the lengths of *Read A* and *Read B*, respectively. It is more suited for long reads than banded alignment [16] because banded alignment uses a static band around the main diagonal. GACT allows for flexible bands since the position of the new tile depends on the traceback path, this is useful for long reads that have high indel rates.

GPU processing

A GPU is a Graphics Processing Unit, which is a processor that is mainly used to perform video processing. GPUs contain many cores that allow them to perform parallelizable tasks very quickly. A GPGPU, or General Purpose GPU, can be programmed to perform tasks that are different from video processing. GPUs cannot operate on their own, they must be guided by a CPU. The functions that run on a GPU are called *kernels* and are usually launched by a CPU.

CUDA is a parallel programming platform that allows people to use Nvidia GPUs for their applications. Developers can write kernels, launched from a CPU function. The GPU is referred to as *device* and the CPU as *host*. Kernels can be launched from the CPU with a certain number of *thread blocks* with each block containing many GPU threads. The number of blocks and the number of threads in a block are the kernel launch parameters. Each thread executes the kernel code, although they usually operate on different data.

On a hardware level, an NVIDIA GPU is divided into Streaming Multiprocessors (SM). Each SM contains several cores, or Streaming Processors (SP), these are the basic building blocks and perform the actual calculations. Each block is assigned to at most one SM. This block's threads are then executed as warps, with 32 threads per warp. Each SM has multiple warp schedulers, so multiple warps can run in parallel on an SM. All threads in a warp must execute the same instruction, if a thread is the only to take a branch, the other threads must wait until the branch is completed, this is called *thread divergence*.

GPUs have several different memory types and levels. It has its own DRAM known as the *global memory* and a cache shared by all SM's. Accesses to the global memory are also executed in parallel, this means that all threads try to read/write to the memory in parallel. If the addresses are next to each other, only one memory transaction is needed, since a transaction processes a whole memory line. This is known as coalescing. Non-coalesced memory accesses cause multiple memory transactions.

A general workflow using a GPU is:

1 Data is copied from the main memory to the GPU global memory.
2 The CPU launches the GPU kernel.
3 The GPU executes the kernel.
4 Results are copied from the GPU to the CPU memory.

Previous research

Multiple efforts to accelerate the DNA alignment algorithm on GPU have been made. MUMmerGPU [17] is one of the first GPU accelerated algorithms, it stores a suffix tree of the reference sequence on the GPU, and aligns it with queries. Its newest GPU implementation shows a 13x speedup over the CPU implementation. CUDAlign [18] accelerates the exact Smith-Waterman algorithm and allows an affine gap. The input sequence length is only restricted by the available global memory. It uses linear space and boasts a 702x and 19.5x speedup compared to 1 core and 64 cores, respectively. CUSHAW2-GPU [19] is an accelerated short read aligner. Other work has been done on accelerating BWA-MEM [20] and Protein database search [21].

Implementation

Profiling

We measured the runtime of various elements of the Darwin algorithm on the CPU. Two notable parts are D-SOFT (which consists of building the minimizer table, and finding the seeds) and aligning using GACT.

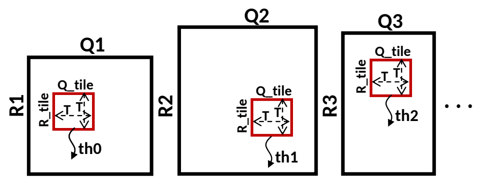


Figure 3 GPU thread assignment for parallelization of the tile computation in GACT. *th* is a GPU thread.

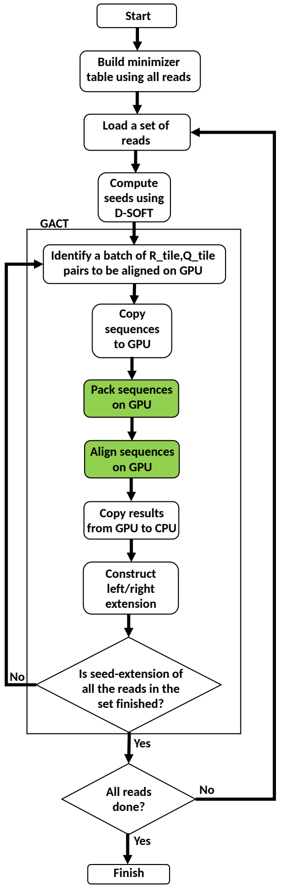

Start
Build minimizer table using all reads
Load a set of reads
Compute seeds using D-SOFT
GACT
Identify a batch of R_tile,Q_tile pairs to be aligned on GPU
Copy sequences to GPU
Pack sequences on GPU
Align sequences on GPU
Copy results from GPU to CPU
Construct left/right extension
Is seed-extension of all the reads in the set finished?
No
Yes
All reads done?
No
Yes
Finish

Figure 4 The flow graph of GPU accelerated Darwin. The green boxes are GPU kernels

Of those two, the *Align*() function in GACT (Listing 1) takes the most time, namely 99.9% for Pacbio reads. Therefore, we accelerated the *Align*() function on GPU. We selected a tile size T of 320 as it gives optimal Smith-Waterman alignment scores [8]. With this setting, 63% of the tiles are exactly 320x320. The remaining tiles are smaller as they occur near the edges of R-Q matrix (Figure 2). If some GPU threads have a smaller tile size, it will cause some divergence, because they will have to wait until the threads with larger tile sizes are finished.

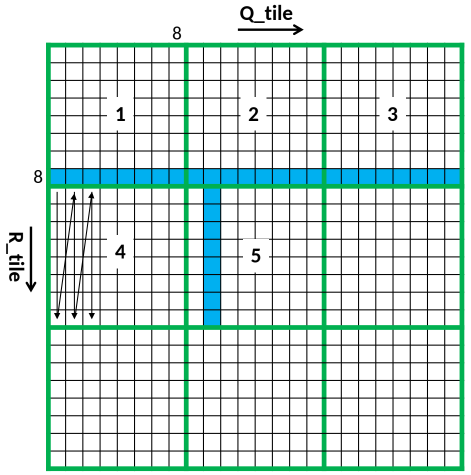


Figure 5 Layout of the tile computed on the GPU. Each green box is a 8x8 submatrix.

Acceleration

It is possible to run the whole GACT kernel on the GPU, for both left and right extension. However, since it is not known how long the resulting alignment will be, and all GPU threads have to wait until all threads are done, this will cause lots of idle time. Instead, it is chosen to only have a single tile of size TxT aligned per GPU-thread per GPU-invocation as shown in Figure 3. The *Align*() function for many different R,Q pairs are executed in parallel on the GPU. Figure 4 shows the flow graph of GPU accelerated Darwin. It has two GPU kernels shown by green processes in the flow graph. All other tasks are performed on the CPU. The CPU builds the minimizer table using all the reads for which the overlaps have to be computed. The accelerated algorithm processes a set of reads to exploit the massive parallelism of the GPU. The CPU first computes the seed hits for all the reads in the set using the D-SOFT algorithm. With the help of the seed

Algorithm 1: The GPU kernel for the *Align*() function of GACT (Listing 1) to compute a tile of size TxT

Input: Two sequences R_{tile}_pack and Q_{tile}_pack of length $T/8$ packed in 32 bit words where each word contains 4 bases (4 bits per base); A backtrack matrix D of size $(T \cdot T) \cdot n_{threads}$ allocated in the GPU memory, where $n_{threads}$ is the number of GPU threads; tile number t; maximum number of bases to align max_{off}; number of GPU threads $n_{threads}$, GPU thread ID tid, score for match $match$, mismatch penalty $mismatch$ and gap penalty gap are known by default

Output: Number of aligned bases in R_{tile} and Q_{tile} (i_{off}, j_{off}); position of the maximum alignment score (i_{max}, j_{max}) and the traceback arrows/pointers tb

```
Function AlignOnGPU($R_{tile}\_pack$ , $Q_{tile}\_pack$, $D$, $t$, $max_{off}$) begin
    Initialize $H$ array of length $T$ containing zeros
    $max\_sc = 0$
    for $i = 0$ to $T/8 - 1$ do
        Initialize $h$ and $p$ arrays of length of $9$ containing zeros
        $r = R_{tile}\_pack[i]$ // $r$ is a 32 bit word
        $r_{idx} = i\cdot$
        $q_{idx} = 0$
        for $j = 0$ to $T/8$ do
            $q = Q_{tile}\_pack[i]$ // $q$ is a 32 bit word
            for $k = 1$ to $8$ do
                $qbase = (q >> (32 - (k \cdot 4)))\&15$  /* C++ like shift operation followed by AND to extract a base from a 32 bit word */
                $h[0] = H[q_{idx}]$
                for $m = 1$ to $8$ do
                    $rbase = (r >> (32 - (m \cdot 4)))\&15$  /* C++ like shift operation followed by AND to extract a base from a 32 bit word */
                    if $qbase == rbase$ then
                        $tmp = p[m] + match$
                    else
                        $tmp = p[m] - mismatch$
                    $h[m] = \max\{tmp, p[m] - gap, h[m-1] - gap, 0\}$
                    if $h[m] == 0$ then
                        $D[(((r_{idx} + (m-1)) \cdot T) + q_{idx}) \cdot n_{threads}) + tid] = 0$
                    else if $h[m] == tmp$ then
                        $D[(((r_{idx} + (m-1)) \cdot T) + q_{idx}) \cdot n_{threads}) + tid] = \nwarrow$
                    else if $h[m] == p[m] - gap$ then
                        $D[(((r_{idx} + (m-1)) \cdot T) + q_{idx}) \cdot n_{threads}) + tid] = \leftarrow$
                    else if $h[m] == h[m-1] - gap$ then
                        $D[(((r_{idx} + (m-1)) \cdot T) + q_{idx}) \cdot n_{threads}) + tid] = \uparrow$
                    if $h[m] > max\_sc$ then
                        $max\_sc = h[m]$
                        $i_{max}, j_{max} = \{r_{idx} + (m-1), q_{idx}\}$
                    $p[m] = h[m-1]$
                $H[q_{idx}] = h[m]$
                $q_{idx} = q_{idx} + 1$
    // the traceback starts below
    if $t$==$1$ then
        $(start_i, start_j) = (i_{max}, j_{max})$
    else
        $(start_i, start_j) = (T-1, T-1)$
    while $D[((start_i \cdot T) + start_j) \cdot n_{threads}) + tid]! = 0$ and $i_{off} \leq max_{off}$ and $j_{off} \leq max_{off}$ and $start_i \geq 0$ and $start_j \geq 0$ do
        $tb$.prepend($D[((start_i \cdot T) + start_j) \cdot n_{threads}) + tid]$)
        if $D[((start_i \cdot T) + start_j) \cdot n_{threads}) + tid] == \nwarrow$ then
            $start_i = start_i - 1$
            $start_j = start_j - 1$
            $i_{off} = i_{off} + 1$
            $j_{off} = j_{off} + 1$
        else if $D[((start_i \cdot T) + start_j) \cdot n_{threads}) + tid] == \uparrow$ then
            $start_i = start_i - 1$
            $i_{off} = i_{off} + 1$
        else if $D[((start_i \cdot T) + start_j) \cdot n_{threads}) + tid] == \leftarrow$ then
            $start_j = start_j - 1$
            $j_{off} = j_{off} + 1$
    return $\{(i_{off}, j_{off}), (i_{max}, j_{max}), tb\}$
```

hit location the sequences for the left and right extension matrices are determined. i.e. (*R_left*,*Q_left*) and (*R_right*,*Q_right*). One tile (*R_tile*, *Q_tile* pair) from each extension matrix is assigned to a GPU thread for alignment. All the tile alignments are computed in parallel on the GPU. There are enough seed hits, and hence sufficient extension matrices in the set of reads to fully utilize all the GPU resources. In the post-processing step, full alignment of the extension using *tb_left* (and *tb_right* for right extension) is constructed on the CPU. As described in the "Profiling" section all the tasks other than computing the alignment between *R_tile* and *Q_tile* takes a negligible amount of time on the CPU.

To reduce the GPU memory accesses, the alignment is preceded by a packing step as indicated in the flow graph of Figure 4. The bases of both sequences are packed in a 4-bit format, where 8 bases are packed into a 32-bit integer. This packing is performed on the GPU and it is hundreds of times faster than packing bases on CPU [22]. To align *R_tile* with *Q_tile*, we extended the local alignment kernel of GASAL (GPU Accelerated Sequence Alignment Library) library [22]. The tile is subdivided into submatrices of size 8x8. Since there are 8 bases in one integer, only two global memory accesses are required to compute a single submatrix. The layout of a tile computed on the GPU is shown in Figure 5. Each green box contains an 8x8 submatrix. The submatrices are computed in the order shown by their number. The arrows in the submatrix number 4 show the order of computation of the dynamic programming cells in a submatrix. The figure shows that the *Q_tile* sequence is read multiple times. Hence, packing the sequence with 4 bits per base helps to keep it in the cache for faster access. It is clear from Equations 1-4 that to compute a column of the submatrix only the cells in the left column and the row above it are required. The required column and row are colored blue in Figure 5. The column has only 8 elements, and hence can be stored in GPU registers. Therefore, the total amount of memory required is $\mathcal{O}(T+T^2)$, where $\mathcal{O}(T)$ is required for computing maximum alignment score m and $\mathcal{O}(T^2)$ for storing the traceback matrix D. Algorithm 1 shows the GPU implementation of the $Align()$ function. The pseudocode above Line 35 is for computing the position of maximum alignment score $(i_{max}, jmax)$ and the traceback matrix D. Observe that the all the writes in D are coalesced to optimize the memory bandwidth and reduce the number of memory transactions. The pesudocode below Line 35 is for computing the traceback path tb. The GPU accelerated Darwin supports both linear as well as affine gap penalties. Algorithm 1 shows the alignment using the linear gap penalties. The algorithm with affine gap penalties has a similar layout and omitted here for brevity.

Sequence alignment of long DNA sequence is a performance bottleneck in genome analysis algorithms. The results of Darwin alignment are same as for normal Smith-Waterman for reasonable values of T and O. Hence, the Darwin algorithm can also be applied for Smith-Waterman alignment between two long DNA sequences and our GPU implementation of Darwin can be used to accelerate Smith-Waterman alignment (with traceback) for long DNA sequences.

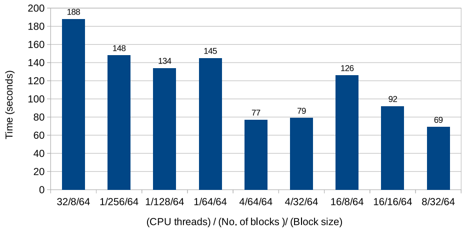


Figure 6 Total execution time of GPU accelerated Darwin for different values of number of CPU threads, numbers of blocks and block size on an IBM machine.

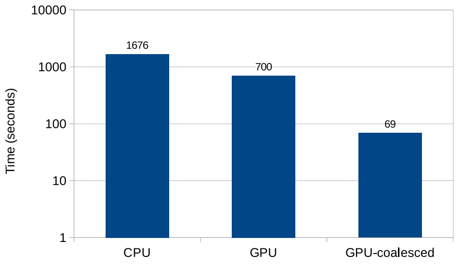


Figure 7 Total execution time of the Darwin's CPU implementation and GPU accelerated Darwin, on the IBM machine. The CPU implementation is running with 64 CPU threads.

Results

We compared our GPU acceleration with the hand-optimized CPU version of Darwin [23] (commit: 16bdb81). Tests are performed on both IBM as well as Intel machines. The IBM machine (S824L) has 2 sockets with each socket containing a 10-core POWER8 @ 3.42 GHz processor. Each core has 8-way Simultaneous Multithreading. Hence, there are 160 logical cores in total. The machine has 256 GB of RAM and a Tesla K40m. The CUDA version is 7.5, and the operating

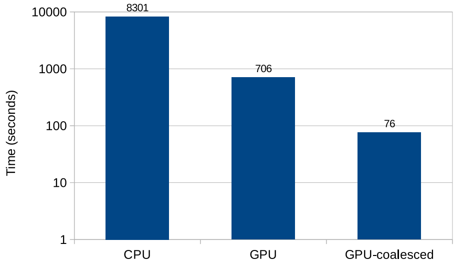


Figure 8 Total execution time of Darwin's CPU implementation and GPU accelerated Darwin, on the Intel machine. The CPU implementation is running with 8 threads.

Table 1 Runtimes and speedup for different scoring schemes on the IBM machine

	(2,-1,-2,-2)	(1,-3,-1,-1)	(5,-4,-10,-1)
CPU	31m15	21m28s	31m27s
GPU-coalesced	76.0	59.3	78.0
speedup	24.7	21.7	24.2

system is Ubuntu 3.19.0-28-generic. The GCC version is 4.9.2.

The Intel machine has 2 sockets with each socket containing a 6-core Xeon E5-2620 @ 2.4 GHz processor. Each core has 2-way Hyperthreading. Hence, there are 24 logical cores in total. The machine has 32 GB of RAM and a Tesla K40c. The CUDA version is 9.2, and the operating system is CentOS 7.5. The GCC version is 4.8.5. The K40c and K40m have the same performance, the only difference lies in their cooling method.

We use Pacbio 54x Human sequencing data [24]. The data has a size of 172 gigabytes containing 21,856,161 reads. Since the runtime of the experiments has a quadratic relationship with the number of reads, we use first 50 megabytes (8566 reads) as the input dataset to finish the experiments in a reasonable time. Even with 50 megabytes input dataset, the CPU implementation takes more than 2 hours to run on 8 threads of the Intel machine (Figure 8). The input dataset contains reads up to 33 kilo bases long with an average read length of 6 kilo bases. Darwin computes the overlaps between the reads in the input dataset. The settings for GPU and CPU implementation are as follows: $match, mismatch, gapo, gape = (1, -1, -1, -1)$, $N = 800$, $T = 320$, $O = 120$, $K = 14$, $h = 21$, $w = 1$.

Since our GPU implementation accelerates only the $Align()$ function on the GPU, everything else is executed on the CPU with multiple threads. Each CPU thread launches a batch of R_tile and Q_tile sequences to be aligned on the GPU. Since all these CPU threads share a single GPU, it is necessary to investigate how the choice of numbers of CPU threads, number of GPU blocks and the number of threads in a block affect the performance. Figure 6 shows the total execution size for various settings of these factors. The figure shows that the fastest execution time is obtained with 8 CPU threads running with the GPU launch parameters of 32 blocks and 64 threads per block. We performed a similar analysis for the Intel machine and found that 8/32/64 is the best setting in the case of the Intel machine as well. Therefore, we used the 8/32/64 ((Number of CPU threads) / (number of blocks) / (number of threads per block)) setting for running the GPU implementation in the remainder of the experimental results.

Figure 7 shows the total execution time of the CPU implementation of Darwin and compares it with the total execution time of the GPU accelerated Darwin, for the IBM machine. Note that the y-axis of the figure represents a logarithmic scale due to the high speedup achieved by the GPU implementation. The CPU implementation is running with 64 threads, which gives nearly the fastest execution time. Two GPU times are reported: “GPU” and “GPU-coalesced”. “GPU” is the time without coalescing the accesses to the traceback matrix D. The figure shows that the GPU acceleration without coalescing is 2.4x faster than the CPU implementation. Coalescing further accelerates the GPU implementation by 10x to achieve an overall speedup of 24x.

Figure 8 show the comparison of total execution times on the Intel machine, again with a logarithmic scale on the y-axis. The CPU implementation is running with 8 threads, which gives nearly the fastest execution time. The non-coalesced GPU implementation achieves a speedup of 11.8x over the CPU. With coalesced memory accesses the speedup becomes 109x. Figure 7 and 8 indicate that coalescing helps to improve the speedup by around 10x. This happens due to efficient utilization of large GPU global memory bandwidth

The above results were obtained with the linear-gap penalty model which is also the default setting in Darwin. However, Darwin and hence our GPU acceleration also support the affine gap model. Table 1 shows the total execution time of the CPU and GPU implementation of Darwin for various values of *match*, *mismatch*, *gapo* and *gape* on the IBM machine. The CPU implementation is running with 64 threads. The table shows that the speedup nearly remains constant (20x-25x) regardless of the scoring scheme. Hence our GPU acceleration is equally effective for both linear and affine gap penalty scoring models.

Conclusions

Read overlapping is an important step in OLC based de novo assemblers. Darwin is a fast and accurate read overlapper for assembly of long DNA reads. It is based on the seed-and-extend paradigm. It has two stages: 1) D-SOFT, to compute the seeds and 2) GACT, to extend the seed hits on both sides to compute the overlap between two reads. The ASIC implementation of Darwin is shown to be hundreds of times faster than software based read overlappers. GPUs are cost-effective and easily accessible processing units that are used to accelerate many high performance applications. In this paper, we have shown a GPU implementation of Darwin which accelerates the Smith-Waterman alignment with traceback computation used in the GACT stage. We pack the sequences on the GPU and compute the Smith-Waterman alignment matrix by dividing the matrix into 8x8 submatrices. This helps to reduce the GPU memory accesses. To further reduce the memory transactions, writing to the traceback matrix is coalesced. We tested our implementation against the hand-optimized CPU implementation of Darwin. The results show that using the real Pacbio dataset, our GPU implementation is 24x faster than 64 IBM Power8 threads and 109x faster than 8 Intel Xeon threads, regardless of the scoring scheme (linear or affine gap). The GPU implementation can also be used to accelerate generic Smith-Waterman alignment of long DNA sequences.

Availability and requirements
Project name: darwin-gpu
Project home page: `https://github.com/Tongdongq/darwin-gpu`
Operating system(s): Linux
Programming language: C++, CUDA
Other requirements: CUDA toolkit version 8 or higher.
License: Apache 2.0
Any restrictions to use by non-academics: Not applicable

List of abbreviations

ASIC: Application Specific Integrated Circuit, **CPU**: Central Processing Unit, **CUDA**:Compute Unified Device Architecture, **DNA**: Deoxyribonucleic Acid, **D-SOFT**: Diagonal-band Seed Overlapping based Filtration Technique, **GACT**: Genome Alignment using Constant memory Traceback, **GASAL**: GPU Accelerated Sequence Alignment Library, **GPU**: Graphical Processing Unit, **NGS**: Next Generation Sequencing, **SM**: Streaming Multiprocessor, **SP**: Streaming Processor.

Author's contributions

NA provided the initial GPU implementation, wrote the manuscript and performed some experiments. TDQ carried out the study, extended the GPU implementation to be integrated in Darwin, performed the experiments and wrote some parts of the manuscript. KB headed the research and revised the manuscript. ZA proposed the idea, supervised the research and helped in writing the manuscript.

Acknowledgements

We are thankful to Yatish Turakhia from Stanford University for his help during the research.

Author details

[1]Delft University of Technology, Delft, Netherlands. [2]University of Engineering and Technology Lahore, Lahore, Pakistan.

References

1. Kececioglu, J.D., Myers, E.W.: Combinatorial algorithms for dna sequence assembly. Algorithmica **13**(7) (1995)
2. Myers, G., Tischler, G., Cunial, F., Pippel, M.: DAZZLER: Dresden Azzembler for Long Read DNA Projects. `https://https://dazzlerblog.wordpress.com`. Accessed 2 July, 2019
3. Simpson, J.T., Durbin, R.: Efficient de novo assembly of large genomes using compressed data structures. Genome Research **22**(3), 549–556 (2012)
4. Pevzner, P.A., Tang, H., Waterman, M.S.: An eulerian path approach to dna fragment assembly. Proceedings of the National Academy of Sciences of the United States of America **98**(17), 9748–9753 (2001)
5. Zerbino, D., Birney, E.: Velvet: algorithms for de novo short read assembly using de bruijn graphs. Genome research, 074492 (2008)
6. Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E.: Abyss: a parallel assembler for short read sequence data. Genome research, 089532 (2009)
7. Luo, R., Liu, B., Xie, Y., Li, Z.: Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler. Gigascience **1**(18), 1–6 (2012)
8. Yatish Turakhia, G.B., Dally, W.J.: Darwin: A Genomics Co-processor Provides Up to 15,000X Acceleration on Long Read Assembly. In: Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '18, pp. 199–213 (2018)
9. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. Journal of Molecular Biology **147**(1), 195–197 (1981)
10. Ahmed, N., Bertels, K., Al-Ars, Z.: A comparison of seed-and-extend techniques in modern dna read alignment algorithms. In: 2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), pp. 1421–1428 (2016)
11. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of Molecular Biology **215**(3), 403–410 (1990)
12. Roberts, M., Hayes, W., Hunt, B.R., Mount, S.M.: Reducing storage requirements for biological sequence comparison. Bioinformatics **20**(18), 3363–3369 (2004)
13. Rucci, E., Garcia, C., Botella, G., De Giusti, A., Naiouf, M., Prieto-Matias, M.: SWIFOLD: Smith-Waterman implementation on FPGA with OpenCL for long DNA sequences. BMC Systems Biology **12**(5), 96 (2018)
14. Farrar, M.: Striped smith–waterman speeds database searches six times over other SIMD implementations. Bioinformatics **23**(2), 156–161 (2007)
15. Hirschberg, D.S.: A Linear Space Algorithm for Computing Maximal Common Subsequences. Commun. ACM **18**(6), 341–343 (1975)
16. Chao, K.M., Pearson, W.R., Miller, W.: Aligning two sequences within a specified diagonal band. Computer applications in the biosciences : CABIOS **8**(5), 481–487 (1992)
17. Trapnell, C., Schatz, M.C.: Optimizing data intensive gpgpu computations for dna sequence alignment. Parallel Computing **35**(8), 429–440 (2009)
18. 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. https://ieeexplore.ieee.org/document/6012857/
19. Liu, Y., Schmidt, B.: CUSHAW2-GPU: Empowering Faster Gapped Short-Read Alignment Using GPU Computing. Design Test, IEEE **31**(1), 31–39 (2014)
20. Houtgast, E.J., Sima, V.M., Bertels, K.L.M., Al-Ars, Z.: An efficient gpu-accelerated implementation of genomic short read mapping with bwa-mem. In: Proc. International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, Hong Kong, China (2016)
21. Hasan, L., Kentie, M.A., Al-Ars, Z.: Dopa: Gpu-based protein alignment using database and memory access optimizations. BMC

Research Notes **4**, 1–11 (2011)

22. Ahmed, N., Mushtaq, H., Bertels, K.L.M., Al-Ars, Z.: Gpu accelerated api for alignment of genomics sequencing data. In: Proc. IEEE International Conference on Bioinformatics and Biomedicine, Kansas City, USA (2017)
23. Turakhia, Y.: Darwin: A co-processor for long read alignment. `https://github.com/yatisht/darwin`. Accessed Nov. 5 2018
24. Data Release: 54x Long-Read Coverage for PacBio-only De Novo Human Genome Assembly. `https://www.pacb.com/blog/data-release-54x-long-read-coverage-for/` (2014)

4

Comparison of Seed-and-Extend Algorithms

In this chapter, we discuss the comparison of commonly used seeding and extension algorithms in the context of seed-and-extend based DNA read mappers. The chapter first gives a brief overview of various seeding and extension techniques. We also present GASE, our generic seed-and-extend read aligner/mapper. Next, the results of the comparison are discussed. Based on the results, we propose suitable combinations of seeding and extension algorithms for various read lengths in DNA read mappers.

This chapter consists of the following articles:

- © 2016 IEEE. Reprinted, with permission, from "N. Ahmed, K. Bertels, and Z. Al-Ars, *A Comparison of Seed-and-Extend Techniques in Modern DNA Read Alignment Algorithms*, in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)* (2016) pp. 1421–1428".

A Comparison of *Seed-and-Extend* Techniques in Modern DNA Read Alignment Algorithms

Nauman Ahmed
Computer Engineering Lab
Delft University of Technology
2628CD Delft,The Netherlands
n.ahmed@tudelft.nl

Koen Bertels
Computer Engineering Lab
Delft University of Technology
2628CD Delft,The Netherlands
k.l.m.bertels@tudelft.nl

Zaid Al-Ars
Computer Engineering Lab
Delft University of Technology
2628CD Delft,The Netherlands
z.al-ars@tudelft.nl

***Abstract*—DNA read alignment is a major step in genome analysis. However, as DNA reads continue to become longer, new approaches need to be developed to effectively use these longer reads in the alignment process. Modern aligners commonly use a two-step approach for read alignment: 1. seeding, 2. extension. In this paper, we have investigated various seeding and extension techniques used in modern DNA read alignment algorithms to find the best seeding and extension combinations. We developed an open source generic DNA read aligner that can be used to compare the alignment accuracy and total execution time of different combinations of seeding and extension algorithms. For extension, our results show that local alignment is the best extension approach, achieving up to 3.6x more accuracy than other extension techniques, for longer reads. For seeding, if BLAST-like seed extension is used, the best seeding approach is identifying all SMEMs in the DNA read (e.g., approach used by BWA-MEM). This combination is up to 6x more accurate than other seeding techniques, for longer reads. With local alignment, we observed that the seeding technique does not impact the alignment accuracy. Furthermore, we showed that an optimized implementation of local alignment using vector instructions, enabling 4.5x speedup, makes it the fastest of all extension techniques. Overall, we show that using local alignment with non-overlapping maximal exact matching seeds is the best seeding-extension combination due to its high accuracy and higher potential for optimization/acceleration for future DNA reads.**

I. INTRODUCTION

High throughput DNA sequencing techniques have caused an enormous decrease in the cost of whole genome sequencing [1]. This decrease has ushered a new era of genome analysis for a large number of applications like genetic disease diagnosis, personalized medicine, agriculture and livestock trait selection. To extract meaningful information from the sequenced genome, it has to pass through various DNA sequence analysis stages. *DNA read alignment* or *DNA read mapping* is the core stage in this analysis. The DNA sequencing machines output the sequenced genome in the the form of millions of short DNA sequences without giving any information about their actual location in the genome. These short DNA sequences are known as *DNA reads* or simply as *reads*. In read alignment, the task is to find the actual location of these DNA reads within a *reference genome* of the species to which the sequenced genome belongs.

TABLE I. Seeding and extension techniques used in modern DNA read aligners

Aligner	Seeding				Extension		
	all-SMEM	nov-SMEM	fix-len (0)[1]	fix-len (1)[2]	global	local	BLAST-ext
BWA-MEM	✓	✓					✓
Bowtie2			✓	✓	✓	✓	
Novoalign			✓			✓	
Cushaw2	✓					✓	

[1] 0 mismatch
[2] at most 1 mismatch

To align a DNA read of length *m* to a genome of length *n*, a dynamic programming algorithm (e.g., Smith-Waterman, Needleman-Wunsch or alike) will require *O(nm)* computation steps. For a human reference genome, $n \approx 3$ billion characters (or bases) long, and therefore, a straight forward application of a dynamic programming algorithm is impractically slow. Moreover, in a typical DNA sequencing experiment, there are hundreds of millions of DNA reads that need to be aligned against the genome. Most modern DNA read aligners tackle this problem by using the *seed-and-extend* approach. The observation behind this approach is that two highly matching sequences contain short substrings that are exactly (or nearly exactly) matching. This approach, pioneered by BLAST [2], aligns a DNA read in two steps: 1. *seeding* and 2. *extension*. Figure 1 shows the seeding and extension phases in a DNA read aligner. During seeding, the aligner first finds substrings of a DNA read that are exactly matching (or nearly exactly) in the genome at one or more than one places. These substrings are known as *seeds*. During extension, the read is aligned to the region around the location of the seed. Such aligners are called seed-and-extend aligners. Many modern DNA read aligners like Novoalign [3], BWA-MEM [4], Bowtie2 [5], and Cushaw2 [6] are seed-and-extend aligners. Table I shows the different seeding and extension strategies used by these aligners (see Section III). In this work, we compare 4 seeding and 3 extension algorithms found in contemporary DNA read aligners.

This paper has the following contributions:

- We developed an open source, generic DNA read aligner that can be used with different seeding and extension techniques [7].
- We compared different combinations of seeding and extension techniques for short as well as long read lengths in terms of accuracy and speed to find the best combinations.

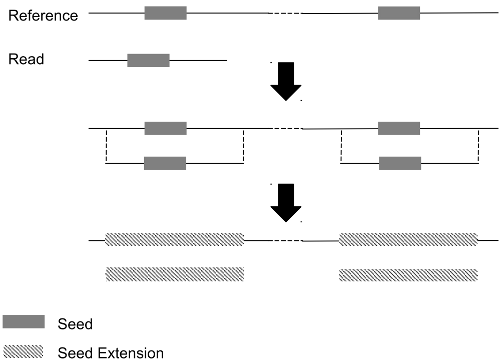


Fig. 1. Seeding followed by extension of short DNA reads against a reference genome

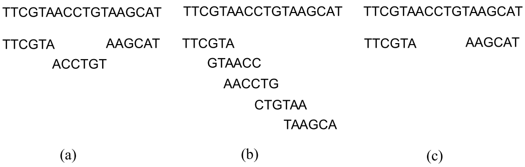
- We optimized the code of the local alignment extension technique to achieve the shortest runtime and one of the highest accuracy combination for longer reads

The rest of the paper is organized as follows: Section II describes the motivation for this paper. Section III and IV introduce different seeding and extension techniques used in the comparison, respectively. Section V presents the details of the DNA read aligner that we have implemented for the comparison. Results of the comparison are shown in Section VI. Finally, we conclude the paper in Section VII.

II. MOTIVATION

Aligners need to be fast and accurate, and have to rely on various heuristics to find a good balance between speed and accuracy. Seeding followed by extension is a heuristic used by many modern DNA read aligners. With the growing importance of genome analysis, many fast and accurate seed-and-extend DNA read aligners have been proposed in recent times, each having its own seed-and-extend method. The accuracy and execution time of a DNA read aligner heavily depends upon the type of seeding and extension technique used.

There are many comparisons of DNA read aligners in the literature. A more recent one is given in [8]. There is also a web based tool for comparing the accuracy of different read aligners [9]. These comparisons evaluate the complete DNA read aligner without focusing on the individual stages of a DNA read aligner. Li and Homer [10] describe different read alignment techniques used by read aligners. They also give an overview of different seeding techniques without discussing their effect on execution time and accuracy on DNA read alignment. A comparison of different kinds of fixed length seeds is given in [11]. The effect of these fixed length seeds on the mapping accuracy and execution time of the DNA read alignment is not discussed. Maximal exact matching seeds are not part of the discussion of any previous comparison. In addition, no previous research has discussed the effect of different extension algorithms on the mapping accuracy and time of the DNA read alignment. Hence, all earlier comparisons in the literature lack the measurement of the contribution of the algorithms used in the seeding and extension phase of the aligner on the accuracy and total execution time of the DNA read alignment. In this paper we will perform such analysis.



(a) (b) (c)

Fig. 2. Simple seeding example (a) seed length = seed interval (b) seed length > seed interval (c) seed length < seed interval

The decreasing cost of DNA sequencing will make computer-based analysis more viable for different application domains. Due to this reason one can foresee the emergence of more DNA read alignment algorithms in the future. This paper will serve as a guideline for developers of DNA read aligners in selecting an appropriate algorithm in the seeding and extension phases of the read alignment process.

III. SEEDING TECHNIQUES

A seed is a substring of the DNA read that is exactly (or nearly exactly) matching in the genome at one or more than one places. Modern DNA read aligners use two kind of seeds: (i) fixed length seeds, and (ii) maximal exact matching seeds. As listed in Table I, Novoalign and Bowtie2 use fixed length seeds while BWA-MEM and Cushaw2 use maximal exact matching seeds.

A. Types of Seeds

1) Fixed length seeds: In this seeding scheme all the seeds have the same fixed length. They are simply overlapping or non-overlapping substrings of the read, all having the same length. Two parameters control the number of seeds generated from a DNA read. The *seed length* and *seed interval*. The seed interval is the number of the DNA read symbols between starting point of two consecutive seeds. Figure 2 shows seeds of DNA read for different relationships between seed length and seed interval. Decreasing the seed length and/or seed interval increases the number of the seeds which increases the sensitivity but at the same time increases the number of candidate seeds to be extended in the extension phase of the read alignment resulting in an increase in the computation time. It is also possible to allow mismatches in a seed. Such seeds are known as *spaced seeds*. Novoalign uses fixed length exact matching seeds. Bowtie2 allows the user to choose between fixed length exact matching seeds and fixed length seeds with at most 1 mismatch.

2) Maximal exact matches: A *maximal exact match* (MEM) is the longest exact match that cannot be further enlarged in either direction. Let P and T be the DNA read and reference string, respectively. Let $P[i,j]$ and $T[i,j]$ be defined as substring of P and T, respectively, starting from the ith

symbol and ending at jth symbol. Then a MEM of the read can be defined as a tuple $(P[q,r], T[m,n])$ such that

$$\begin{aligned} P[p] = T[t] \quad & \forall p \quad q \leq p \leq r \\ & \forall t \quad m \leq t \leq n \end{aligned}$$

and

$$\begin{aligned} P[q-1] &\neq T[m-1] \\ P[r+1] &\neq T[n+1] \end{aligned}$$

A more refined form of the MEM is proposed in [12] and is called as *super maximal exact match* (SMEM). A MEM which is not contained in any other MEM of the read is known as SMEM. Let there be k MEMs of a DNA read: MEM$_1$= $(P[q_1,r_1], T[m_1,n_1])$, MEM$_2$= $(P[q_2,r_2], T[m_2,n_2])$ $\ldots$ MEM$_k$= $(P[q_k,r_k], T[m_k,n_k])$. Then MEM_i for $i = 1 \ldots k$ is an SMEM if and only if:

$$\begin{aligned} (q_i < q_j \text{ or } r_i > r_j) \text{ and } (m_i < m_j \text{ or } n_i > n_j) \\ \forall j \quad j = 1, 2, \ldots i-1, i+1 \ldots k-1, k \end{aligned}$$

In this work, a seeding technique which finds all the overlapping and non-overlapping SMEMs in a read will be called as *all-SMEM*, whereas the scheme in which only the non-overlapping SMEMs are generated will be called as *nov-SMEM*. As an example, consider the following genome:

$CCAATGTCTCATGGTGTCTCAGCTCTCAGAATTCAGATC$

and a DNA read:

$$CAATGTCTCAGATAA$$

The all-SMEM seeds of the this read are $CAATGTCTCA$, $TGTCTCAG$, $TCAGAT$ and AA. The nov-SMEM seeds are $CAATGTCTCA$, GAT and AA. For the same seed setting (i.e., minimum required seed length) all-SMEM is more sensitive than nov-SMEM but nov-SMEM is faster.

B. Seed Computation

Seed computation refers to finding the seed sequence and computing its starting position(s) in the reference genome. As described above seeds are substrings of the read that are exactly (or nearly exactly) matching at one or more than one places in the reference genome. Computation of a seed requires a pre-built index of the reference genome. Different kinds of genome indexes can be built. Here we will only focus on those which are found in modern DNA read aligners. Contemporary DNA read aligners compute seeds by either using hash table index (e.g., as in Novoalign) or using FM-index [13] (e.g., as in BWA-MEM, Bowtie2 and Cushaw2).

1) FM-index: FM-index [13] is a memory efficient index of the reference genome. It is a representation of the suffix/prefix trie of the reference genome. Other representations also exist like suffix array [14] and enhanced suffix array [15], but FM-index has the smallest memory footprint. The FM-index consists of three arrays: (1) The count array C, (2) the BWT array B, and (3) the suffix array SA. The count array has four entries, one for each of the four DNA base symbols (i.e., A, C, T and G). An entry for symbol e stores the number of symbols in the reference DNA that are lexicographically smaller than e. The BWT array is the Burrows-Wheeler transform of the reference DNA. The suffix array holds the starting positions of the suffixes of the reference DNA. Computing a seed using FM-index is a two step process:

Algorithm 1: Backward Search using FM-index

```
Input: String $W$ and length of reference genome $|T|$. $B$ and $C$ array are
       assumed to be known
Output: Set of suffix array intervals $[I_l, I_u]$ of W and the match length
Function BackwardSearch($W, |T|$) begin
    Initialize $[I_l, I_u]$ as $[0, |T|-1]$
    $i \leftarrow |W| - 1$
    // match_len is used to compute nov-SMEM
    // $match\_len \leftarrow 1$
    while $I_l \leq I_u$ and $i > -1$ do
        $I_l \leftarrow C[W[i]] + Occ(W[i], I_l - 1)$
        $I_u \leftarrow C[W[i]] + Occ(W[i], I_u) - 1$
        $i \leftarrow i - 1$
        $match\_len \leftarrow match\_len + 1$
    // Uncomment the following line to find nov-SMEM
    // return $([I_l, I_u], match\_len)$
    if $i = -1$ and $I_l \leq I_u$ then
        return $([I_l, I_u])$
    else
        // return empty interval
        return $\emptyset$
Function Occ($a, j$) begin
    $x \leftarrow 0$
    $y \leftarrow 0$
    while $x \leq j$ do
        if $B[x] = a$ then
            $y \leftarrow y + 1$
        $x \leftarrow x + 1$
    return $y$
```

Algorithm 2: Computing the starting position for a given suffix array index

```
Input: Suffix array index $k$. Suffix array sampling rate $r$; $B$ , $C$ and $SSA$ array
       are assumed to be known
Output: Starting position corresponding to $k$
Function CalcStart($k$) begin
    $i \leftarrow 0$
    while $k \mod r \neq 0$ do
        $k \leftarrow C[B[k]] + Occ(B[k], k-1)$
    return $SSA[k]$
```

Step 1—Computing suffix array interval: Given a seed W of length $|W|$ and the FM-index of the reference DNA T, the suffix array interval of W can be computed using Algorithm 1. The proof of the algorithm is given in [13].

The algorithm returns the suffix array interval of W written as $[I_l, I_u]$ where:

$$\begin{aligned} I_l(W) &= \min\{i : W \text{ is the prefix of } SA(i)\} \\ I_u(W) &= \max\{i : W \text{ is the prefix of } SA(i)\} \end{aligned}$$

From I_l to I_u are the suffix array indexes of all those suffixes of T in which W is the prefix. The algorithm returns an empty interval if W is not present in the reference DNA.

Algorithm 1 is known as backward search as it starts from the last symbol of W and then builds the string in the backward direction. Each iteration of the `while` loop enlarges the string by one symbol and may be called as a *search step*. Each search step returns the suffix array interval of the enlarged string. If $I_l \leq I_u$, the enlarged string exits in T otherwise not. Algorithm 1 is used to find the suffix array interval of a fixed length seed. To find the suffix array interval of all the fixed length seeds in a DNA read, the `BACKWARDSEARCH` function is called with W set equal to the seed sequence. To find the suffix array interval of nov-SMEM, uncomment line 12. To find the first nov-SMEM, call the `BACKWARDSEARCH` with $W = P$, where P is the DNA read sequence. The function will return $match\_len$ along with the suffix array interval. If $match\_len = |P|$, we are done, otherwise call the function again to find the second nov-SMEM with $W = P_1$ where $P_1 = P[match\_len, |P|]$. Similarly, if $match\_len = |P_1|$, the algorithm completes successfully, otherwise the function is called again to find the third nov-SMEM with $W = P_2$ where $P_2 = P_1[match\_len, |P|]$, and so on. The algorithm to compute the suffix array interval of all-SMEMs is given in [12].

Step 2—Computing start position: Once the suffix array intervals of the seeds are computed, the suffix array can be used to find the starting position of a seed (if present). Usually, to reduce memory a sampled suffix array SSA is used. A SSA with a sampling rate of r is the set $\{SA(k) : k \text{ is divisible by } r\}$. Algorithm 2 computes the starting position of a seed with SSA for a given suffix array index value. Each suffix array index value from I_l to I_u corresponds to one occurrence of the seed. Hence, the `CALCSTART` function is called for $k = I_l, \ldots I_u$.

The advantage of using FM-index is its memory efficiency. The complete FM-index for the the human reference genome occupies only 1.5 Gbytes of memory. The time required to compute a seed of length n is $O(n + mr)$ where $m = I_u - I_l + 1$ i.e. the number of occurrences of the seed. In practice, computing seeds with FM-index consume a lot of time due to pseudo-random accesses to the large B array which is nearly 1 GB. Such a large array cannot reside in the cache. As shown in Algorithm 1 during every search step the algorithm accesses the B array. Similarly B array is also accessed in every iteration of the loop in Algorithm 2. In [16] the memory access patterns of B array are studied. The study shows that these accesses are quite random resulting in a large number of data cache and data TLB misses due to poor temporal and spatial locality of the accesses. These large D-cache and D-TLB misses cause the algorithm to almost always be waiting for the memory, substantially slowing down the algorithm.

2) Hash table index: Fast computation of fixed length seeds can be performed using a hash table. Hash table stores the starting position of n-mers of the reference genome, where n is the length of the seed. To find the starting position of a fixed length seed just index the hash table with the seed sequence. Hence, they require $O(1)$ time to compute a seed. Hash tables are sensitive to the value of n and the sampling frequency s of the genome. Sampling frequency is the distance (in no. of bases) between the starting position of two consecutive n-mers of the genome. For $n > 15$ the hash table size becomes excessively large. Novoalign has a hash table index that occupies 17 GB of RAM with $n = 15$ and $s = 3$ for the human genome. Hence, a hash table index, although fast, is memory demanding. The index has to be rebuilt if the seed length is changed. Hash tables cannot be directly used to compute maximal exact matches. To find maximal exact matching seeds with hash table index, first find the start position of a substring of the read with a hash table and then enlarge it on both sides by a direct comparison between the read and the reference genome. A similar approach has been adopted by the HPG DNA read aligner [17].

IV. EXTENSION TECHNIQUES

Flanking bases of the reference genome around the seed are fetched to perform the extension step. Three types of seed extension techniques are used in modern DNA read aligners: (1) Global alignment (Needleman-Wunsch algorithm), (2) Local alignment (Smith-Waterman algorithm), and (3) BLAST-like seed extension. All these three techniques are implemented using dynamic programming with affine gap penalties. Cushaw2 performs local alignment, Bowtie2 allows the user to choose between local and global alignment, while BWA-MEM performs BLAST-like seed extension.

A. Global and local alignment

In global and local alignment, the bases around the seed in the reference genome are fetched to form a target sequence that contains the surrounding bases as well as the seed. In global alignment the goal is to find the highest scoring alignment of the full read against the target sequence. In practice the target sequence is longer than the read. Therefore, a semi-global alignment is performed in which gaps on both ends of the read are ignored. In local alignment the goal is only to achieve the highest scoring alignment and thus the resulting alignment may not contain the full read sequence. Global and local alignment algorithm are explained in detail in textbooks. Readers may refer to [18] for more in depth coverage of local and global alignment algorithms.

B. BLAST-like seed extension

BLAST-like seed extension is a fast extension technique that is performed in two steps by calling Algorithm 3 twice, which shows the BLAST-like seed extension algorithm. First step: *seq1 = read bases on the left side of the seed, seq2 = reference bases on the left side of the seed, start_score = seed score*. Second step: *seq1 = read bases on the right side of the seed, seq2 = reference bases on the right side of the seed, start_score = seed score + alignment score of first step*. The BLAST-like seed extension algorithm is similar to local alignment with the following differences:

1. Non-zero start score.
2. A standard local alignment algorithm computes all the

local alignments between two sequences. BLAST-like seed extension is faster as it only computes one local alignment that must contain the seed as a substring. The pseudo-code framed in the first box (i.e., lines 25-26) in Algorithm 3 ensures that this requirement is met. Further speedup is achieved due to the pseudo-code framed in the second box (i.e., lines 31-44) which prunes the Dynamic Programming (DP) matrix entries that cannot result in a final alignment containing the seed. Hence, BLAST-like seed extension does not compute all the the entries of the DP matrix making it faster than local and global alignment techniques.
3. The starting positions of the alignment are always known, so no traceback is required, thereby reducing run time.

C. Optimized seed extension

The dynamic programming based extension stage is compute bound. Therefore, to reduce the total DNA read alignment execution time, extension schemes can be optimized. Striped Smith-Waterman (SSW) is a SIMD optimization of local alignment [19]. The implementation of SSW is also available in the form of a C/C++ library [20]. SIMD optimized DP allows concurrent computation of many DP matrix cells. Another optimization is banded DP, which limits the number of DP matrix cell to be calculated to a narrow band along the main diagonal [21]. Banded DP works well in situations where the two sequences to be aligned are *homologous* as the case of DNA read alignment.

V. GASE GENERIC ALIGNER

For the comparison of different seeding and extension techniques, we built GASE (Generic Aligner for *Seed-and-Extend*) that can be used with different seeding and extension techniques. The idea is to use GASE to measure the corresponding alignment accuracy and total execution time for different combinations of seeding and extension. GASE is a minimalistic aligners that mainly depends on the seeding and extension technique being used to identify the read alignment, with little added heuristics coded in the aligner. This results in an aligner with an alignment accuracy that is mainly determined by the seeding and extension technique being used. The different components of our read aligner are outlined below.

1) Index: The FM-index used in our aligner is same as the one generated in BWA-MEM. BWA-MEM builds the FM-Index of $T \oplus \overline{T}$, where T is the reference genome string , $\overline{T}$ is Watson-Crick reverse complement of T and $\oplus$ is the string concatenation operator. The advantages of such kind of index are: 1) Apart from backward search, shown in Algorithm 1, where the string is enlarged from right to left in the reverse direction, a *forward search* is also possible in which the string can also be enlarged from left to right in forward direction 2) A read P is only aligned against $T \oplus \overline{T}$, rather than aligning P and its Watson-Crick reverse complement $\overline{P}$ against T separately. This roughly doubles the speed of the aligner at the cost of memory.

Algorithm 3: BLAST-like seed extension

```
Input: The two sequences to be aligned $seq1$, $seq2$ and the start score
       start_score. Penalty of gap open $gapo$ and gap extension $gape$ are
       assumed to be known values
Output: Maximum score $score$ and its position of achievement on read
        sequence $read\_end$ and on the reference $ref\_end$
1  Function BlastSeedExtension($seq1, seq2, start\_score$) begin
2      Initialize $H$, $E$ and $F$ arrays of size of $(|seq1| + 1) * (|seq2| + 1)$
         containing zeros
3      $H[0][0] \leftarrow start\_score$
4      $H[0][1] \leftarrow max\{start\_score - gapo - gape, 0\}$
5      for $j \leftarrow 2$ to $|seq1|$ do
6          $H[0][j] \leftarrow max\{H[0][j-1] - gape, 0\}$
7      $H[1][0] \leftarrow max\{start\_score - gapo - gape, 0\}$
8      for $j \leftarrow 2$ to $|seq2|$ do
9          $H[j][0] \leftarrow max\{H[j-1][0] - gape, 0\}$
10     $max\_score \leftarrow start\_score$
11     $read\_end \leftarrow \emptyset$
12     $ref\_end \leftarrow \emptyset$
13     $beg \leftarrow 1$
14     $end \leftarrow |seq1|$
15     for $i \leftarrow 1$ to $|seq2|$ do
16         $row\_max \leftarrow 0$
17         $max\_j \leftarrow \emptyset$
18         for $j \leftarrow beg$ to $end$ do
19             $E[i][j] \leftarrow$
                 $max\{H[i-1][j] - gapo - gape, , E[i-1][j] - gape, 0\}$
20             $F[i][j] \leftarrow$
                 $max\{H[i][j-1] - gapo - gape, , F[i][j-1] - gape, 0\}$
21             $H[i][j] \leftarrow max\{H[i-1][j-1] + S(seq1[j-$
                 $1], seq2[j-1]), E[i][j], F[i][j], 0\}$
22             if $H[i][j] > row\_max$ then
23                 $row\_max = H[i][j]$
24                 $max\_j = j$
           ┌──────────────────────────────────────────────
25         │ if $row\_max = 0$ then
26         │     return $\{max\_score, read\_end, ref\_end\}$
           └──────────────────────────────────────────────
27         if $row\_max > max\_score$ then
28             $max\_score = row\_max$
29             $read\_end = max\_j$
30             $ref\_end = i$
           ┌──────────────────────────────────────────────
31         │ $j \leftarrow beg$
32         │ while $j < end$ do
33         │     if $H[i-1][j-1] = 0$ and $H[i-1][j] = 0$ and
           │       $E[i-1][j] = 0$ then
34         │         $j \leftarrow j + 1$
35         │     else
36         │         break
37         │ $beg \leftarrow j$
38         │ $j \leftarrow end$
39         │ while $j >= beg$ do
40         │     if $H[i-1][j-1] = 0$ and $H[i-1][j] = 0$ and
           │       $E[i-1][j] = 0$ then
41         │         $j \leftarrow j - 1$
42         │     else
43         │         break
44         │ $end \leftarrow j$
           └──────────────────────────────────────────────
45     return $\{max\_score, read\_end, ref\_end\}$
```

2) Seeding: Seeds are computed depending upon the seeding technique under test. We have compared four seeding methodologies used in contemporary read aligners. 1) Fixed length seeds without mismatch: we varied the seed length from 15 to 50 in steps of 5 2) Fixed length seeds with at most one mismatch allowed: we varied the seed length from 15 to 90 in steps of 5 3) all-SMEMs: we varied the minimum required seed length from 15 to 50 in steps of 5 4) nov-SMEMs: we varied the minimum required seed length from 15 to 50 in steps of 5. For all the above seeds, the seed interval is varied as 1, 5, 10, 15, . . . , *seed length*. If a seed is located at more than 500 positions in the reference genome, it is not extended.

3) Chaining: The seeds which lie nearby on the reference genome are chained together. The chains are sorted in descending order on the basis of the weight of the chain. The weight of a chain is the number of reference genome bases covered by the seeds in a chain. A chain is filtered out if it is overlapping with the next higher weight chain by more than 50%.

4) Extension: The seeds in a chain are sorted on the basis of their length. The longest seeds is extended first using one of the three extension techniques being studied. The next seed in the sorted list of seeds is then extended if it is not already covered in the extension of the previous seed and so on. The process is repeated for all the chains. Three different extension techniques have been tested: global alignment, local alignment and BLAST-like seed extension. The output is written in SAM (Sequence Alignment/Map) format [22].

VI. EXPERIMENTAL RESULTS

We measured the error in alignment as:

$$\text{error} = \frac{\text{No. of incorrectly mapped reads}}{\text{No. of mapped reads}}$$

We tested all the 12 possible combinations of seeding and extension techniques. A read aligned within ±20 base pairs of the true position is considered correct. For each combination of seeding and extension technique, the seed length and seed interval (if applicable) is varied over a range discussed in Section V-2. Only those seed settings have been considered in the comparison in which the number of mapped reads $\geq$ 99.5% of the total number of reads.

A. Input data set

5 Mega bases of the chromosome 21 of human genome (UCSC hg19) are used as a reference. 1 million single ended reads were generated using Wgsim read simulator [23]. Ten reads are aligned in parallel by running ten threads on Intel Xeon E5-2670 2.5 GHz processor. The reads have a mutation rate of 0.4% where 25% of these mutations are indels. 70% of the indels are extended (length greater than 1). This mutation rate in the simulated reads represents the upper limit in human genome variation [24]. Similarly this percentage of indels and their extension rate in the simulated reads correspond to observed values in the human genome [25]. The reads have 2% sequencing errors as well.

B. Selecting seeding parameters

Table II shows the results of measuring the mapping error and the total execution time of DNA read aligner for different seeding and extension techniques with varying read lengths. The read length is specified in base pairs (bp). As described in Section V-2 a number of parameters of the seed are varied over a wide range. Each seed setting results in a different mapping error and execution time. The values in Table II represent a tradeoff between error and time. For each setting we measured the corresponding error and time. If the difference in error between the *most accurate* seed setting and another seed setting is at most 9 incorrectly mapped reads and achieving at least 30% faster execution time than the most accurate seed setting, then the other seed setting is selected.

C. Comparison of seeding techniques

Table II shows that fixed length seeds with no mismatch are not a good choice in any case. They result in more error in the mapping and larger total execution time with all kinds of extension techniques as compared to SMEM seeds and fixed length seed with at most 1 mismatch for all read lengths. In some cases of fixed length seeds with no mismatch, we have not tested some smaller seed lengths due to orders of magnitude higher total execution time as compared to other seeding techniques, and hence useless to consider. With BLAST-like seed extension, all-SMEM is the best approach due to its higher accuracy and lower execution time. For example with 600 bp read length, it is 1.34, 6 and 2 times more accurate than nov-SMEM, fixed length (0 mismatch) and fixed length (at most 1 mismatch) seeds, respectively. Similarly for 600 bp read length, the execution time is 1.37 and 2 times less than fixed length (0 mismatch) and fixed length (at most 1 mismatch) seeds, respectively, and comparable with nov-SMEM. For local alignment all-SMEM, nov-SMEM and fixed length seeds (at most 1 mismatch) have comparable mapping error. For global alignment all-SMEM is the most accurate. With local alignment all-SMEM is faster than nov-SMEM and fixed length seeds (at most 1 mismatch). For global alignment nov-SMEM is the fastest.

D. Comparison of extension techniques

Table II shows that the local alignment is the most accurate for all kinds of seeding techniques. The local alignment becomes more accurate as compared to the global and BLAST-like seed extension techniques with increasing read lengths. For 600 bp read length it is 3.6 times and 1.6 times more accurate than BLAST-like seed extension and global alignment, respectively. The results also show that BLAST-like seed extension should only be used with all-SMEM as its accuracy drops significantly as compared to local and global alignment with other three seeding techniques (i.e. nov-SMEM, fixed length with no mismatch and fixed length with at most 1 mismatch). Global alignment is less accurate than local alignment but more accurate than BLAST-like seed extension for all kinds of seeding techniques. With regard to speed of the unoptimized techniques, BLAST-like seed extension is the fastest and it becomes faster with increasing read lengths as compared to other two extension techniques. With all-SMEM seeding it is 2.2 to 3.4 times faster than unoptimized local and global alignment for 600 bp read length. Although global alignment is more accurate than BLAST-like seed

TABLE II. Mapping error and total execution time of our DNA read aligner GASE with different combinations of seeding and extension techniques. The values before the slash (/) in the time column represent the execution time with unoptimized extension stage, whereas the values after slash are obtained with optimized extension stage.

	Read len.	all-SMEM		nov-SMEM		fix-len. (0-mismatch)		fix-len. (1-mismatch)	
		error	time (sec.)	error	time (sec.)	error	time (sec.)	error	time (sec.)
global	150	1.24e-3	93	1.3e-3	74	1.39e-3	1704	1.21e-3	209
	250	3.35e-4	173	3.81e-4	151	4.66e-4	2949	3.73e-4	185
	400	8.3e-5	370	9.4e-5	427	2.15e-4	11637	1.2e-4	710
	600	4e-5	890	4.9e-5	742	2.7e-4	43021	1.07e-4	1194
local	150	1.22e-3	59/26	1.3e-3	74/24	1.262e-3	1138	1.21e-3	183/113
	250	3.28e-4	180/61	3.68e-4	159/58	3.68e-4	554	3.45e-4	284/99
	400	6.8e-5	354/126	7e-5	507/128	6.9e-5	5864	5.6e-5	419/200
	600	2.5e-5	805/260	3e-5	1241/272	5.5e-5	3866	3.2e-5	824/392
BLAST-like seed extension	150	1.25e-3	28/27	1.37e-3	24	1.80e-3	120	1.34e-3	42
	250	4.16e-4	64/59	4.8e-4	60	1.05e-3	65	4.77e-4	262
	400	1.36e-4	148/134	1.97e-4	136	6.91e-4	193	2.57e-4	216
	600	9.4e-5	305/260	1.26e-4	276	5.05e-4	364	1.88e-4	419

extension, it is not suitable for aligning *split reads* (also known as chimeric reads). Split reads are generated due to large structural variations in the genome. The speed of optimized Blast-like and local alignment techniques is comparable.

E. Selecting the best seeding-extension combination

From Table II we can conclude that:

- Blast-like seed extension is the fastest and should at least be combined with all-SMEM
- Local alignment is the most accurate and its accuracy is nearly the same with all-SMEM, nov-SMEM and fixed length (at most 1 mismatch) seeds

To come up with the best seeding-extension combination we must first decide the best seeding technique to be used with local alignment. To do this we will compare the execution times of optimized DNA read aligners. The speed optimization performed here does not affect the mapping error. These optimization speed up the extension stage of the DNA read aligner with techniques described in Section IV-C. We only focus on optimizing the extension rather than the seeding due to the memory bound nature of seed computation using FM-index, which makes it hard to optimize/accelerate. Hash table index is a fast seeding mechanism (as compared to the FM-index) for finding fixed length seeds, but it cannot be used in our case as the values reported for fixed length seeds with at most 1 mismatch in Table II are mostly for very long seeds (30 or above). Shorter fixed length seeds have much higher error as compared to the values given in Table II. Building hash table for seeds longer than 20 bp is impractical. We can select the best seeding technique to be used with local alignment by measuring the execution time of the DNA read aligner with optimized local alignment. The local alignment is SIMD optimized using SSW. It is implemented with Intel SSE2 instruction set which has 128-bit SIMD registers. A signed two-byte integer is used to store the score value. This allows the concurrent computation of 8 DP matrix cells.

Figure 3 compares the execution of local alignment with three seeding techniques (all-SMEM, nov-SMEM and fixed length seeds with at most 1 mismatch) before and after applying the SIMD optimization. The input data set is the same as the one used in Table II. The mapping error for each seeding technique remains nearly the same as reported in Table II and hence, is comparable to all techniques. Figure 3 shows that after SIMD optimization nov-SMEM-local becomes slightly faster than fixed-length(1)-local and has nearly same execution time as all-SMEM-local. The figure shows the significant reduction in execution time of local alignment with nov-SMEM. Its execution time scales down by 4.57x for 600 bp read length. For the range of DNA read lengths shown, the reduction in execution time of nov-SMEM is 2.74x up to 4.57x, for all-SMEM it is 2.32x up to 3.1x and for fixed length seeds with at most 1 mismatch it is 1.62x up to 2.86x. Therefore, the combination of nov-SMEM with local alignment achieves the highest reduction in execution time with minimal reduction in accuracy. Although, nov-SMEM generate less seeds as compared to all-SMEM and fixed length seeds, however for longer reads the amount of seeds generated by nov-SMEM is sufficient for accurate mapping of the read. The high reduction in the execution time of nov-SMEM-local for longer reads shows that with future DNA reads nov-SMEM-local has high potential for acceleration/optimization without sacrificing accuracy.

Now that we have seen that nov-SMEM is a better seeding approach for local alignment, we will now compare the execution time of SIMD optimized nov-SMEM-local and banded all-SMEM-BLAST-ext DNA read aligners (which is the fastest combination using the Blast-like extension algorithm) to come up with the best seeding-extension combination. Our banded implementation of BLAST-like seed extension is the same as done in BWA-MEM. Figure 4 compares the execution time of the SIMD optimized nov-SMEM-local and banded all-SMEM-BLAST-ext. The input data set is the same as the one used in Table II. The optimization has not increased the mapping error of both schemes and therefore, nov-SMEM-local remains more accurate than all-SMEM-BLAST-ext for all read lengths (with 600 bp read length nov-SMEM-local

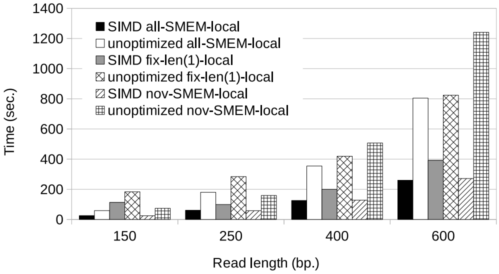


Fig. 3. Comparison of execution time of local alignment with all-SMEM, nov-SMEM and fixed length seeds (at most 1 mismatch) before and after SIMD optimization

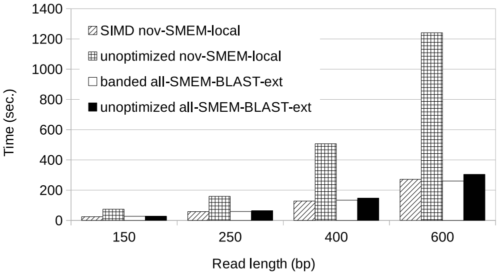


Fig. 4. Comparison of execution time of BLAST-like seed extension with all-SMEM and nov-SMEM-local before and after optimization

is 3.1x more accurate than all-SMEM-BLAST-ext). Figure 4 shows that the banded all-SMEM-BLAST-ext seed extension was not able to gain much speed as compared to the unoptimized all-SMEM-BLAST-ext of Table II, whereas SIMD nov-SMEM-local shows good speed up has nearly same execution time as banded all-SMEM-BLAST-ext. Therefore, we can conclude that SIMD optimized nov-SMEM-local seeding-extension combination have performed best in this comparison due to its high accuracy, and more potential for optimization/acceleration for future DNA reads.

VII. CONCLUSION

In this paper we compared different seeding and extension techniques used in modern DNA read aligners. The compared seeding techniques were maximal exact matching seeds and fixed length seeds. Three seed extension techniques were compared: global alignment, local alignment and BLAST-like seed extension. For the purpose of the comparison we built an open source generic seed-and-extend DNA read aligner called GASE and then measured the accuracy and execution time for all the possible seeding and extension techniques. Our results showed that fixed length seeds (0-mismatch) are not a good seeding choice with any type of extension algorithm, while all-SMEM is the best seeding approach with BLAST-like seed extension. Local alignment is more accurate than BLAST-like seed extension, especially for longer reads. all-SMEM, nov-SMEM and fixed length seeds (at most 1 mismatch) have comparable accuracies with local alignment. Overall, SIMD optimized nov-SMEM-local seeding-extension combination has performed best in this comparison due to its high accuracy and more potential for optimization/acceleration for future DNA reads.

REFERENCES

[1] Wetterstrand KA., “DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP),” Available at: www.genome.gov/sequencingcosts, Accessed [30th September, 2015].

[2] S. F. Altschul *et al.*, “Basic local alignment search tool,” *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403 – 410, 1990.

[3] “NovoAlign,” http://www.novocraft.com/products/novoalign/.

[4] H. Li, “Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM,” *arXiv [q-bio.GN]*, May 2013. [Online]. Available: http://arxiv.org/abs/1303.3997

[5] B. Langmead and S. S., “Fast gapped-read alignment with bowtie 2,” *Nature Methods*, vol. 9, pp. 357–359, 2012.

[6] Y. Liu and B. Schmidt, “Long read alignment based on maximal exact match seeds,” *Bioinformatics*, vol. 28, no. 18, pp. i318–i324, 2012.

[7] “GASE generic aligner,” https://github.com/nahmedraja/GASE.

[8] J. Shang *et al.*, “Evaluation and comparison of multiple aligners for next-generation sequencing data analysis,” *BioMed Research International*, 2004.

[9] “Genome Comparison and Analytic Testing,” http://www.bioplanet.com/gcat.

[10] H. Li and N. Homer, “A survey of sequence alignment algorithms for next-generation sequencing,” *Briefings in Bioinformatics*, vol. 11, no. 5, pp. 473–483, 2010.

[11] I. Mandoiu and A. Zelikovsky, *Bioinformatics Algorithms: Techniques and Applications*. John Wiley and Sons, 2008, ch. 6, pp. 117–142.

[12] H. Li, “Exploring single-sample SNP and indel calling with whole-genome de novo assembly,” *Bioinformatics*, vol. 28, no. 14, pp. 1838–1844, Jul 2012.

[13] P. Ferragina and G. Manzini, “Opportunistic data structures with applications,” in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, ser. FOCS ’00, 2000, pp. 390–398.

[14] U. Manber and G. Myers, “Suffix arrays: A new method for on-line string searches,” *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[15] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, “Replacing suffix trees with enhanced suffix arrays,” *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53 – 86, 2004.

[16] J. Zhang *et al.*, “Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures,” in *CCGrid*, Delft, Netherlands, May 2013.

[17] J. Trraga *et al.*, “Acceleration of short and long dna read mapping without loss of accuracy using suffix array,” *Bioinformatics*, 2014.

[18] W. K. Sung, *Algorithms in Bioinformatics: A Practical Introduction*. CRC Press, 2009, ch. 2, pp. 29–56.

[19] M. Farrar, “Striped smithwaterman speeds database searches six times over other SIMD implementations,” *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[20] M. Zhao *et al.*, “SSW library: An SIMD smith-waterman c/c++ library for use in genomic applications,” *PLoS ONE*, vol. 8, 12 2013.

[21] K. Chao, W. R. Pearson, and W. Miller, “Aligning two sequences within a specified diagonal band,” *Computer applications in the biosciences : CABIOS*, vol. 8, no. 5, pp. 481–487, 1992.

[22] “SAM format specification,” https://samtools.github.io/hts-specs/SAMv1.pdf.

[23] “Wgsim,” https://github.com/lh3/wgsim.

[24] S. Tishkoff and K. K. Kidd, “Implications of biogeography of human populations for ’race’ and medicine,” *Nature Genetics*, vol. 36, no. 11s, pp. S21 – S27, 2004.

[25] R. Mills *et al.*, “An initial map of insertion and deletion (indel) variation in the human genome,” *Genome Research*, vol. 16, no. 5, pp. 1182–90, 2006.

5

Predictive Genome Analysis

As described in Section 1.4.4 of Chapter 1, delay in the sequencing process of the DNA represents a major bottleneck to reduce the latency of DNA analysis. In this chapter, we present our approach to overcome the delay in GATK variant calling analysis using short Illumina reads. We describe the methods to reduce the sequencing delay by starting the computational analysis before the sequencing is completely finished by predicting the unknown bases and the corresponding base quality scores. Using a real exome sequencing dataset, we measured the accuracy by calculating the area under the precision-recall curve (APR) and found that the analysis output of our approach with 50 unknown bases is more than 90% accurate compared to the full dataset. This helps to reduce the sequencing time by up to a day.

This chapter consists of the following articles:

- © 2017 IEEE. Reprinted, with permission, from “N. Ahmed, K. Bertels, and Z. Al-Ars, *Predictive Genome Analysis using Partial DNA Sequencing Data*, in *2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE)* (2017) pp. 119–124”.

Predictive Genome Analysis Using Partial DNA Sequencing Data

Nauman Ahmed, Koen Bertels and Zaid Al-Ars
Computer Engineering Lab, Delft University of Technology, Delft, The Netherlands
{n.ahmed, k.l.m.bertels, z.al-ars}@tudelft.nl

***Abstract*—Much research has been dedicated to reducing the computational time associated with the analysis of genome data, which resulted in shifting the bottleneck from the time needed for the computational analysis part to the actual time needed for sequencing of DNA information. DNA sequencing is a time consuming process, and all existing DNA analysis methods have to wait for the DNA sequencing to completely finish before starting the analysis. In this paper, we propose a new DNA analysis approach where we start the genome analysis before the DNA sequencing is completely finished. The genome analysis is started when the DNA reads are still in the process of being sequenced. We use algorithms to predict the unknown bases and their corresponding base quality scores of the incomplete read. Results show that our method of predicting the unknown bases and quality scores achieves more than 90% similarity with the full dataset for 50 unknown bases (slashing more than a day of sequencing time). We also show that our base quality value prediction scheme is highly accurate, only reducing the similarity of the detected variants by 0.45%. However, there is still room to introduce more accurate prediction schemes for the unknown bases to increase the effectiveness of the analysis by up to 5.8%.**

***Index Terms*—DNA Sequencing delay; Prediction; GATK;**

I. INTRODUCTION

The decreasing cost of DNA sequencing [1] has enabled scientists to perform genome analysis easily and with increasing resolution for applications ranging from research to clinical diagnostics.

In *Variant Calling* the sequenced DNA sample is compared against a reference genome to find the genetic variations in the sample as opposed to the reference. In this paper, we will use variant calling as case study for predictive genome analysis. Genome Analysis Toolkit (GATK) [2] is a widely-used variant calling pipeline. The stages in the GATK pipeline for detecting SNPs and INDELs are described in [3]

Both *DNA sequencing* as well as *DNA analysis* consume a lot of time before variants are available for further investigation (e.g., diagnostics). High-throughput Illumina DNA sequencing machines (such as the HiSeq 2500) require up to a week to fully sequence the DNA. Similarly, the processing of the large amounts of data by the genome analysis pipeline results in a huge computation time as well.

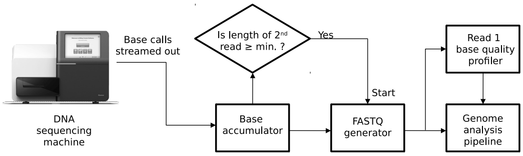


Fig. 1: The proposed scheme to hide DNA sequencing delay

A lot of effort has been made in the past to accelerate the computation time of individual stages of the pipeline as well as accelerating the whole pipeline using cluster based computing. BWA-MEM is accelerated in [4]. A multithreaded version of Picard tools is presented in [5]. An FPGA acceleration of the PairHMM calculation is given in [6]. A cluster based Spark implementation of the whole GATK pipeline is presented in [7]. As a result of these efforts in reducing the computation time, the process of DNA sequencing is becoming the limiting-factor in the total time required for genome analysis. The process of DNA sequencing takes days to complete [8], while implementations of the genome analysis pipeline on computer clusters can process hundreds of gigabytes of DNA sequencing data in less than two hours [9].

In this paper, we overcome the problem of long DNA sequencing time by partially hiding its delay. This is achieved by starting the genome analysis while the sequencing of the DNA read data is still in progress. In this way, our scheme does not wait for the DNA sequencing process to completely finish before starting the analysis. As the genome analysis is started while the DNA read is still being sequenced, we do not know the values of the last bases of the read and their corresponding base quality scores. Therefore, we introduced an additional stage in the genome analysis pipeline that predicts the value of the unknown bases and their corresponding base quality scores. A patent based on the work in this paper is also filed in Europe [10].

The outline of the rest of the paper is as follows. Our approach to hide the DNA sequencing delay is presented in Section II. The method of predicting unknown bases, their corresponding base quality scores and some additional SAM fields is described in Section III. Experimental results of our proposed techniques are discussed in Section IV. We finally conclude the paper in Section V.

II. APPROACH

In this paper, we propose a scheme in which the large DNA sequencing time is partially hidden by starting the genome analysis before the DNA sequencing is completely finished. Current high throughput DNA sequencing machines (e.g., Illumina), sequence both ends of a DNA fragment to generate paired-end read data. Paired-end read data allows

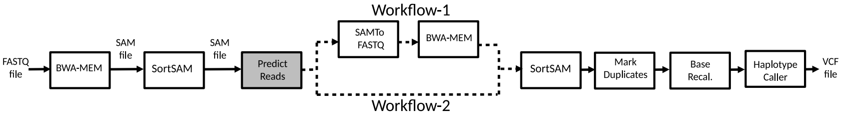


Fig. 2: (a) Workflow 1 and 2. Workflow 1 has two additional stages: `SAMToFASTQ` conversion and remapping with `BWA-MEM`.

more accurate genome analysis as compared to single-end reads. The DNA sequencing technology used by Illumina is known as *Sequencing by Synthesis* (SBS). In SBS, the first and the second read in the paired-end data is generated by sequencing the forward and reverse strand, respectively. The two reads in the pair are sequenced one after the other. Hence, the first read in the paired-end is completely sequenced followed by the second read. Moreover, one base is sequenced at a time. Sequencing a base, produces two values: 1) The actual value of the base (i.e., A, T, C, G or N (ambiguous base)) and 2) The *base quality score* which is the probability of error in the sequencing process.

In high-end Illumina DNA sequencing machines (e.g., HiSeq 2500), generating around 1 Terabyte of data, sequencing a base takes around 30 minutes [8]. Therefore, hiding the sequencing of even a few bases results in large saving in time. In this work, we have reduced the DNA sequencing latency by starting the genome analysis while the DNA sequencing of the second read in a paired-end read is still in progress. In this way, we can save a large amount of time, and the genome analysis can be completed much earlier as compared to the case in which the genome analysis is started after the DNA sequencing is completely finished. Figure 1 shows the proposed scheme. The sequenced bases are streamed out of the DNA sequencing machine while the sequencing is still in progress. The bases are stored in a base accumulator. After enough bases of the second read have been accumulated, the FASTQ file is generated. At the same time a base quality profile of the first read is also generated. The FASTQ file and the base quality profile of the first read are used to perform the genome analysis and complete the unknown part of the reads

III. METHODS

In the rest of the paper we will use the following terminology:

1) The paired-end read dataset which would have been generated if the DNA sequencing is allowed to finish is called *original read dataset*.
2) The second read in the paired-end read dataset which would have been generated if the DNA sequencing is allowed to finish is called *original second read*.
3) The paired-end read dataset in which the second read has unknown bases due to incomplete DNA sequencing is called *incomplete read dataset*.
4) The first read in the incomplete read dataset is exactly the same as that in the original read dataset and is simply called *first read*.
5) The second read in the paired-end dataset with unknown bases due to incomplete DNA sequencing is called *incomplete second read*.
6) The number of unknown bases in the incomplete second read is called $n\_unknown$.
7) The paired-end read dataset in which the unknown bases and quality scores of the second read are completed by our read prediction schemes is called *completed read dataset*.
8) The second read in the paired-end read dataset with unknown bases and quality scores that have been completed by our read prediction schemes are called *completed second read*.

A. Input read dataset

In the experiments in this paper, we use the whole exome sequencing of NA12878 dataset. This dataset has 150x coverage with paired-end reads and a read length of 100 base pairs (bp) [11]. The 150x dataset is used to generate subsets of datasets with 50x and 100x coverage. Throughout the paper we will use these three read datasets as the original read datasets. Last tens of bases of the second reads of these datasets are clipped to form the incomplete read datasets.

B. Workflows

As described above we have reduced the DNA sequencing delay by starting the genome analysis while the second read in the paired-end DNA read data is still being sequenced. Therefore, the values of the last bases of the second read and their corresponding base quality scores are unknown to us. We have designed a *prediction stage* `PredictReads` that predicts the values of the unknown bases of the second read and their corresponding base quality scores. We have tested the efficacy of our prediction stage using two different workflows. Figure 2 shows Workflow-1 and Workflow-2, respectively, of our prediction scheme.

Workflow-1 (WF-1) starts with a FASTQ file in which the last bases of the second read and their corresponding base quality score values are unknown. It first maps the incomplete read dataset using `BWA-MEM`. The mapped reads are sorted (w.r.t. mapping position) using Picard's `SortSAM`. The unknown bases of the second read and their corresponding base quality score are predicted using our `PredictReads` stage. The predicted bases and their corresponding base quality scores are appended at the end of the second read. This SAM file is then converted into a FASTQ file using Picard's `SAMToFASTQ` utility. The output of the `SAMToFASTQ` is a

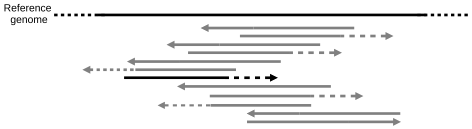


Fig. 3: Set of overlapping reads mapped to a reference genome. Lines with arrowheads are the reads

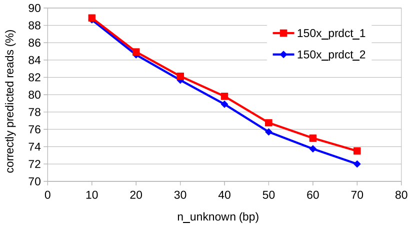


Fig. 4: Prediction accuracy of unknown bases

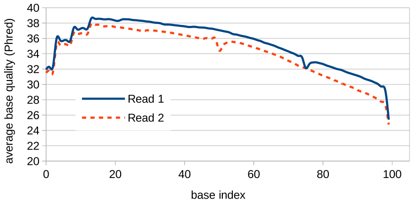


Fig. 5: Average base quality profile of WES 150x dataset.

FASTQ which is a completed read dataset. This FASTQ file is used as an input for a run of the whole GATK pipeline of.

Workflow-2 (WF-2) starts with the same first three stages (from `BWA-MEM` to `PredictReads`) of WF-1. WF-2 then sorts the SAM output file of the read prediction stage and continues to execute the remaining stages in the GATK pipeline after `SortSAM`. The read prediction stage of WF-2 is slightly different from the read prediction stage of WF-1. Apart from predicting the unknown bases of the second read and their corresponding base quality scores, the read prediction stage of WF-2 has to also predict/correct some additional fields of the input SAM file, as described in Section III-C3.

C. Read prediction

The core of the the proposed scheme is the read prediction stage. Read prediction has to perform the following: 1) Predict the values of the unknown bases of the second read, 2) Predict the base quality scores of the unknown bases of the second read, and 3) Predict some additional fields of the input SAM file in case of WF-2.

1) Predicting unknown bases: We predict the unknown bases by detecting the overlap between the incomplete second read and the reads that are mapped close by. Figure 3 shows a set of overlapping reads mapped to a reference genome. The black-colored read is an example of an incomplete second read, while the gray-colored reads are the overlapping reads. The reads containing dotted lines are the incomplete second reads, where the dots show the unknown bases. The arrow on the reads indicate the direction of mapping. We tested various prediction schemes. Here we will describe two prediction schemes that give the best results:

Scheme-1: For each unknown base we take a majority vote of the bases from the overlapping reads to predict the value of the unknown base. An unknown base having no overlap is substituted with the reference genome base located at the corresponding position. The number of bases having no overlap is quite low for high coverage data. For 150x coverage data with the last 30 bases unknown in the second read, there are only 2.22% unknown bases with no overlap. Moreover, only 3.64% of the unknown bases have less than 10 overlapping bases. Hence, in most of the cases the majority vote is taken among a large number of overlapping bases.

Scheme-2: In the second scheme, we predict the unknown bases of the incomplete second read by matching the known bases of the incomplete second read with the overlapping reads. The overlapping read that is matching most closely is used to predict the unknown bases. The bases of the overlapping read covering the unknown bases are used to complete the unknown bases of the incomplete second read. If the overlapping read does not cover all the unknown bases of the incomplete second read, then the remaining unknown bases are predicted using prediction scheme 1.

Figure 4 shows the effectiveness of scheme 1 and scheme 2 in completing the unknown bases in the incomplete reads. The figure shows percentage of the incomplete seconds reads that the scheme is able to complete perfectly (i.e., the completed second reads becoming exactly the same as the original second read). The original read data set has 150x coverage. The figure shows that prediction scheme 1 results in more accurate prediction of unknown bases as compared to prediction scheme 2. Therefore, in this work we will use prediction scheme 1 to predict the unknown bases.

2) Predicting unknown base quality scores: For predicting the base quality scores, we observe the fact that the slope of the average base quality score pattern generated by Illumina machines for the first read and the second read is nearly the same. We used FastQC [12] to plot the average base quality score of the first and second read across the read length. Figure 5 shows a plot of the average base quality score values across the entire read for the 150x original read dataset. It clearly shows that the slope of the average base quality score pattern of the first read and the second read is nearly the same.

To predict the base quality scores of the unknown bases of the second read, we modeled the base quality score pattern of the last $n\_unknown$ bases of the first read with a piece-wise linear function. The number of "pieces" in our piece-wise linear model is equal to $n\_unknown - 1$. Assuming that the slope of the base quality score pattern of the second read is same as the average of that of the first read, our piece-wise linear model is able to correctly predict the unknown base quality scores of the second read.

3) Predicting additional SAM fields: In the workflow WF-2 (Section III-B, apart from predicting the unknown bases and their corresponding base quality scores, we also need to predict some other SAM fields in the read prediction stage. These are: A. Mapping position of the completed second read B. CIGAR string of the completed second read.

A. Mapping position of the completed second read: In both workflows WF-1 and WF-2, mapping using BWA-MEM is the first stage. Therefore, if the incomplete second read is mapped on the reverse strand of the reference genome, then its mapping position in the SAM file will always be $n\_unknown$ positions ahead of the original second read, assuming that there are no deletions and soft clipping in the last $n\_unknown$ bases of the original second read. In our prediction of the mapping position, we assume that there are no deletions and soft clipping in the last $n\_unknown$ number of bases of the original second read, and hence, subtract $n\_unknown$ from the mapping position of the incomplete second read to form the mapping position of the completed second read. If the mapped incomplete second read has soft clipping at the beginning of the read we do not perform this operation. Figure 6 shows a plot of the correctly mapped reads in the completed read dataset. The original read dataset has 150x coverage. A read is regarded as *correctly mapped* if its mapping position is same as the mapping position of the read in the original read dataset. The percentage of correctly mapped reads are shown for two cases: 1) *150*x*_wf1* representing the case of WF-1, which is the percentage of correctly mapped reads after the completed read dataset is remapped using BWA-MEM, and 2) *150*x*_wf2* representing the case of WF-2, which is the percentage of correctly mapped reads in the completed read dataset, in which the mapping positions of the completed second reads are predicted using the method describe above. The plot shows that the reads in WF-2 have very high mapping accuracy. On the other hand, remapping the reads after the prediction of the unknown bases of the second read, as done in WF-1, greatly reduces mapping accuracy.

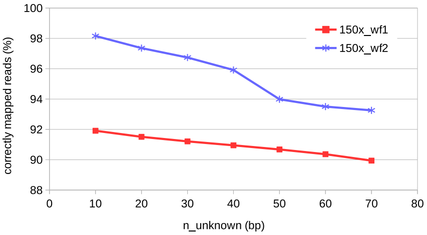
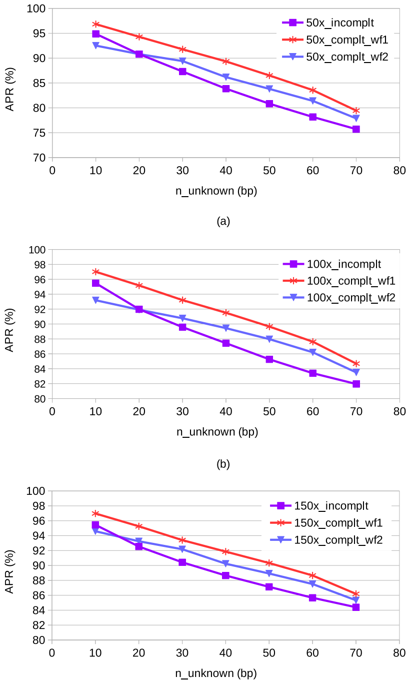


Fig. 6: Percentage of correctly mapped reads



(a)

(b)

(c)

Fig. 7: (a) Effectiveness plot for 50x coverage data. (b) Effectiveness plot for 100x coverage data. (c) Effectiveness plot for 150x coverage data.

B. CIGAR string of the completed second read: In WF-2, we reevaluate the CIGAR string of the completed second read by performing a semi-global alignment between the completed second read and the substring of reference genome. Let T be the reference genome and $T[a, b]$ be its substring starting from reference position a and ending at position b. The substring of the reference genome used in the semi-global alignment is $T[p-10, p+qlen+10]$, where p is the predicted mapping position of the completed second read and the $qlen$ is the length of the completed second read. If the mapped incomplete read is soft clipped at either end, the CIGAR string is not reevaluated. Instead, the first or last operation in the CIGAR string of the incomplete second read is extended by $n\_unknown$ depending upon the mapping strand of the incomplete second read. If the incomplete second read is mapped on the reverse strand, the first CIGAR operation is extended and vice versa.

IV. RESULTS

For evaluating our proposed scheme we implemented the read prediction stage in practice. Our read prediction stage is capable of predicting the unknown bases, unknown base quality scores and the additional SAM fields as described in Section III-C. The prediction of additional SAM fields is only required in WF-2. Our prediction stage requires the reference genome and base quality profile of the first read as input in addition to the SAM file of the DNA reads mapped and sorted w.r.t. mapping position. We used UCSC hg19 as the reference genome. We also used *1000G_phase1*, *dbsnp_138* and *Mills_and_1000G_gold_standard* as the known SNP and indel sites for the `BaseRecalibrator` stage. All the Picard and GATK tools are run with default settings. BWA-MEM also is run with default settings except the use of `-M` option, essential for Picard compatibility.

We first run the GATK pipeline with original read dataset to generate *orig* the set of variant calls (VCF file) that we compare our techniques with. We then clipped the last tens of bases of the original second reads to generate incomplete read dataset. The set of variant calls (VCF file output) of the GATK pipeline computed for the incomplete read dataset will be called as $incomplt$. To test our prediction scheme, we predict the unknown bases, unknown base quality scores and some additional SAM fields (only in WF-2). The sets of variant calls generated by running WF-1 and WF-2 of Figure 2 are called as $complt\_wf1$ and $complt\_wf2$, respectively. *Precision* and *recall* (sensitivity) can be defined as:

$$precision = \frac{TP}{TP + FP} \times 100\% \tag{1}$$

$$recall = \frac{TP}{TP + FN} \times 100\% \tag{2}$$

where TP, FP and FN are the true positives, false positives and false negatives, respectively, . In order to evaluate the effectiveness of the predictive analysis workflows defined in this paper, we use the area under the precision-recall (APR) curve as our metric. APR indicates the effectiveness of a pipeline in identifying as much as possible correct variants (high TP) while identifying as little as possible incorrect variants (low FP). In the ideal case, APR is equal to 100%, and the closer the APR is to 100%, the more effective the workflow is. This definition of APR is the same as the one used by [13] to evaluate the effectiveness of various variant calling pipelines. The APR is calculated for $complt\_wf1$, $complt\_wf2$ and $incomplt$, with respect to *orig*. We use `RTG` tools [14] to calculate the precision-recall graph. This is then further used to evaluate the APR of our workflows.

Figure 7 shows the effectiveness in terms of APR of $incomplt$, $complt\_wf1$ and $complt\_wf2$ with respect to *orig*, while clipping 10, 20, .. up to 70 bases of the original second read. Figures 7a, 7b and 7c show the APR plot for 50x, 100x and 150x coverage data, respectively. *Scheme-1* explained in Section III-C1 is used to predict the unknown bases in the second read. The figures show that the APR decreases almost linearly with increasing number of unknown bases. At the same time, the overall APR of all sets of variant calls increases as the data coverage is increased from 50x to 150x. The figures also show that WF-1 is the most effective workflow to accurately call variants of incomplete reads, consistently achieving a higher APR than WF-2 and $incomplt$ for all cases. For an increasing $n\_unknown$, the APR of WF-2 gets gradually closer to that of WF-1, but never actually reaching it. Although WF-2 has a much higher read mapping accuracy than WF-1 (according to Figure 6), Figure 7 shows that WF-1 has a better APR than WF-2 for all cases. This degradation in APR of WF-2 as compared to WF-1 can be attributed to the prediction scheme (scheme-1 of Section III-C1) that we used to predict the the unknown bases. In scheme-1, we take a majority vote of the bases from the overlapping reads to predict the value of the unknown base. This majority vote may cause a true variant to be overshadowed and hence, being missed in WF-2. On the other hand, in WF-1 the unknown bases of the incomplete second read are predicted, and then the completed read dataset is remapped to the reference genome. This causes some of the reads to be mapped to a different position than the initial mapping and hence, do not overshadow a true variant.

Figure 8 shows a plot of the difference between the APR of WF-1 and $incomplt$ for a range of $n\_unknown$. As pointed earlier, the figure shows that WF-1 is always better than $incomplt$. The difference is initially small but increases with increasing $n\_unknown$, then peaks at around

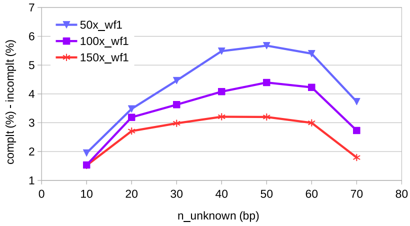


Fig. 8: The difference between the APR of WF-1 and $incomplt$ for a range of $n\_unknown$ and coverage values.

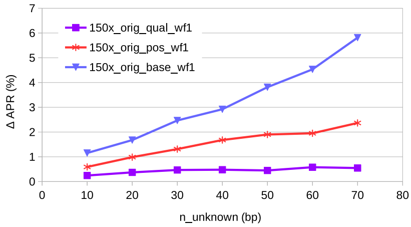


Fig. 9: Increase in the APR after assuming original values for mapping position, unknown base quality score or unknown base value.

$n\_unknown = 50$, and finally falls back down. The plot also shows that the difference in the APR of WF-1 and $incomplt$ is much higher at lower coverage than at higher coverage. For $n\_unknown = 50$, the difference is 5.7% and 3.2% for 50x and 150x coverage, respectively. This means that with increasing coverage the APR of $incomplt$ increases at a much higher rate than WF-1. Therefore, our method of predicting unknown bases and their corresponding base qualities has a bigger comparative impact on the APR with lower coverage as compared to $incomplt$.

We also studied the effect of accurate prediction of different parameters of the incomplete second read. We make three different assumptions in WF-1. 1) *orig_pos*: We assume that the mapping position of the reads going into the read prediction stage of WF-1 is exactly the same as mapping position of the reads in the original read dataset. The unknown bases and the corresponding base quality scores are predicted as described in Section III-C1 (scheme-1) and III-C2, respectively. 2) *orig_qual*: We assume that the unknown base quality scores of the incomplete second read are predicted with ideal accuracy (i.e., they are exactly the same as in original second read). The unknown bases of the incomplete second read are predicted as described in Section III-C1(scheme-1). 3) *orig_base*: We assume that the unknown bases of only those incomplete second reads in which the mapping positions are correct (same as that of original second read), are predicted with ideal accuracy (i.e., they are exactly the same as in original second read). The mapping positions of the incomplete second reads are predicted using the same method as described in Section III-C3. Unknown base quality scores of the incomplete second read are predicted as described in Section III-C2. Figure 9 shows the increase in APR for each of the three cases. $\Delta APR = 150\text{x}\_y\_wf1 - 150\text{x}\_complt\_wf1$, y is *orig_pos*, $orig\_qual$ or $orig\_base$. Figure 9 shows that the assumption of accurate prediction of the unknown base values ($orig\_base$) causes a much higher increase in APR as compared to the other two assumptions. For $orig\_qual$, the increase in APR is quite small. Hence, there is a very little room for improvement in our method of predicting the unknown base quality scores. For $n\_unknown = 50$, accurate prediction of unknown base quality scores and read mapping positions cause an increase of 0.45% and 2% in the APR, respectively. On the other hand, accurately predicting the value of the unknown bases of only those incomplete second reads which have been mapped correctly can help to increase the APR by 3.8%. Therefore, we can conclude that the method of predicting unknown bases can be further improved to achieve more effectiveness.

V. CONCLUSION

In this paper, we proposed a predictive genome analysis approach based on the idea of starting the genome analysis before the DNA sequencing is completely finished. We introduced an additional stage in the GATK pipeline to predict the unknown bases and their corresponding base quality scores, due to incomplete DNA sequencing. Two workflows were proposed to achieve this purpose. The results showed that our method of predicting the unknown bases and quality scores achieves more than 90% similarity with the analysis performed on the full dataset for 50 unknown bases (slashing more than a day of sequencing time).

We also measured the impact of accurate prediction of unknown bases, unknown base quality scores and the read mapping position to improve the effectiveness of the workflows in identifying the variants. Results show that our base quality and read mapping position prediction scheme is highly accurate. with 50 unknown bases, ideal prediction of the value of the base quality scores and read mapping position gives only 0.45% and 2% higher similarity with analyzing the full dataset, respectively. However, accurately predicting the value of those unknown 50 bases can achieve a 3.8% higher similarity, meaning that more effective base prediction methods can achieve an even higher analysis accuracy.

ACKNOWLEDGMENT

This work is supported by the Faculty Development Program of the University of Engineering and Technology Lahore, Pakistan.

REFERENCES

[1] K. Wetterstrand, "DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP)," Available at: www.genome.gov/sequencingcosts, 2016, Accessed [15 October, 2016].

[2] A. McKenna *et al.*, "The genome analysis toolkit: A mapreduce framework for analyzing next-generation dna sequencing data," *Genome Research*, vol. 20, no. 9, pp. 1297–1303, 2010.

[3] G. van der Auwera, "GATK Best Practices," https://software.broadinstitute.org/gatk/best-practices/, 2016.

[4] N. Ahmed, V. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars, "Heterogeneous Hardware/Software Acceleration of the BWA-MEM DNA Alignment Algorithm," in *ICCAD'15*, 2015, pp. 240–246.

[5] A. Tarasov *et al.*, "Sambamba: fast processing of NGS alignment formats," *Bioinformatics*, 2015.

[6] J. Peltenburg, S. Ren, and Z. Al-Ars, "Maximizing Systolic Array Efficiency to Accelerate the PairHMM Forward Algorithm," in *BIBM'16*, 2016, pp. 758–762.

[7] H. Mushtaq and Z. Al-Ars, "Cluster-based apache spark implementation of the gatk dna analysis pipeline," in *BIBM*, Nov 2015, pp. 1471–1477.

[8] Illumina, "Illumina HiSeq-2500 System Specifications," https://www.illumina.com/documents/products/datasheets/datasheet_hiseq2500.pdf, 2015.

[9] H. Mushtaq, F. Liu, C. Costa, G. Liu, P. Hoftsee, and Z. Al-Ars, "SparkGA: A Spark Framework for Cost Effective, Fast and Accurate DNA Analysis at Scale," in *ACM-BCB*, August 2017, pp. 148–157.

[10] Z. Al-Ars, N. Ahmed, and K. Bertels, "Early DNA Analysis Using Incomplete DNA Datasets," Patent, European Patent no. NL 2017750 (pending), 2016.

[11] G. Highnam *et al.*, "An analytical framework for optimizing variant discovery from personal genomes," *Nature Communications*, vol. 6, no. 6275, 2015.

[12] S. Andrews, "FastQC: a quality control tool for high throughput sequence data," http://www.bioinformatics.babraham.ac.uk/projects/fastqc/, 2010.

[13] S. Hwang *et al.*, "Systematic comparison of variant calling pipelines using gold standard personal exome variants," *Scientific Reports*, vol. 5, no. 17875, 2015.

[14] J. Cleary *et al.*, "Comparing variant call files for performance benchmarking of next-generation sequencing variant calling pipelines," *bioRxiv*, 2015.

6

Conclusions and Recommendations

After having described our optimized and accelerated algorithms for fast genome analysis, in this chapter we present the major takeaways of the research work described in this thesis.

Seed-and-extend algorithms are an integral part of data-intensive genome analysis applications. Therefore accelerated solutions for these algorithms will help to reduce the time and cost of DNA analysis pipelines. In this thesis, we presented optimized and accelerated seeding as well as extension algorithms and applied them to enhance the performance of various genome analysis applications. We demonstrated that GPUs are a good platform to accelerate maximal exact match (MEM) seeding algorithms. We also optimized a MEM computation algorithm and showed that rewriting the algorithm in an intelligent manner helps to reduce the memory accesses in the memory-bound algorithm, boosting up its performance. Similarly, for the extension, we demonstrated that seed-extension via sequence alignment can be performed much faster on GPU as compared to CPU. We developed a high performance GPU library for sequence alignment and showed that it can be easily integrated into genome analysis applications. We also performed comparisons of different seeding and extension algorithms for DNA read mappers with the help of our generic seed-and-extend mapper. Our analysis shows that maximal exact matching algorithms combined with local alignment give the best results for read mapping. Finally, we presented a method to perform variant calling on partial sequencing data to reduce the DNA sequencing delay bottleneck. Our approach gives an accuracy of more than 90% while reducing the sequencing time by one full day.

The outline of this chapter is as follows: Section 6.1 describes the main contribution of the thesis as well as the lessons learned during the research, on a chapter-wise basis. Section 6.2 presents the recommendation for future research work.

6.1. Conclusions

Genome analysis has a variety of applications in everyday life. The current progress in DNA sequencing will shape the future of life science technology ranging from nutrition to clinical diagnostics. Current sequencing platforms generate huge amounts of DNA data. Therefore, fast and low cost DNA analysis is essential to achieve the envisioned goals of DNA sequencing. In this thesis, we provide various solutions to overcome the challenges in achieving fast analysis of high-throughput DNA sequencing data. We presented software optimized and GPU accelerated DNA analysis algorithms based on the seed-and-extend paradigm. Furthermore, we discussed methods to avoid the large sequencing delay caused by sequencing machines needed to read DNA information. In the following, we describe our major research contributions and observations.

Chapter 2: Fast Seeding Algorithms

Maximal exact matching (MEM) and super-maximal exact matching (SMEM) seeds are used in many seed-and-extend algorithms due to their high accuracy. Their computation time may become a bottleneck in DNA analysis applications. In Chapter 2 we presented GPU accelerated and software optimized maximal exact matching seeding algorithms. The following conclusions can be drawn from Chapter 2.

1. Computation of maximal exact matching seeds is a highly parallelizable task which makes it a good candidate for acceleration on GPUs.
2. Apart from NVBIO, there is no other GPU acceleration available for the computation of maximal exact matching seeds using FM-index for DNA sequencing data. Furthermore, NVBIO is only suitable for short DNA reads. Besides, CPU based libraries are also not designed for sequencing data that require smaller seed lengths. Therefore, their performance drastically reduces while finding shorter seeds in the sequencing data.
3. We presented *GPUseed* a high-performance GPU accelerated API for computing maximal exact matching seeds in DNA sequences using the FM-index. The implementation extracts maximum parallelism from the computation to fully utilize the massive parallelism of the GPU. Two unique optimizations are applied which are 1) Pre-computed suffix array intervals of partial seed 2) Avoiding redundant computation.
4. We tested our library on state-of-the-art third generation long-read sequencing data against a highly optimized CPU based algorithm for computing max-

imal exact matching seeds running on 24 Intel Xeon threads. On an NVIDIA Tesla K40 GPU, GPUseed is up to 9x and 5.6x faster than CPU implementations with Pacbio and Oxford Nanopore data, respectively.

5. The acceleration results of GPUseed show that even with memory-bound algorithms, like computing maximal exact matching seeds using FM-index, good speedups could be achieved with GPUs if the algorithm is massively parallelizable.
6. Maximal exact matching seeds are computed in the BWA-MEM read mapper. The suffix array interval computation is a highly memory-bound task an takes about 30% of the total execution time with short DNA reads on BWA-MEM version 0.7.8. [44].
7. We presented a software optimized algorithm for computing the seeds in BWA-MEM. Our algorithm reduces the number of memory accesses by up to 45% resulting in a 1.7x speedup of the suffix array computation of SMEM seeds.
8. We applied our optimized algorithm for computing suffix array intervals of maximal exact matching seeds to an accelerated BWA-MEM mapper (version 0.7.8), in which the other steps of the mapper are accelerated on FPGAs. Our optimization achieved an overall application speedup of up to 2.6x.
9. In the newer versions of BWA-MEM (0.7.11 and onwards) suffix array interval computation of seeds takes up to 44% of the total execution time [44], thereby allowing more speedup with our optimized algorithm.

Chapter 3: GPU Accelerated API for Sequence Alignment

Sequence alignment is one of the most common operations in DNA analysis algorithms. It is used in the extension step of seed-and-extend algorithms and also as a standalone step (i.e. without seeding) in many applications. In Chapter 3, we presented GASAL2, a GPU accelerated API for sequence alignment. GASAL2 can perform local, global and all types of semi-global alignment; with and without traceback computation. We also showed two applications of the API. The conclusions drawn from Chapter 3 are as follows.

1. GASAL2 and NVBIO are the only two GPU accelerated libraries for the sequence alignment of DNA sequencing data. NVBIO does not support ambiguous bases ("N") and lacks the capability of allowing overlap execution of GPU and CPU.
2. NVBIO converts the bases of the sequences to be aligned from an 8-bit representation to 4 bits (known as packing) on the CPU which is extremely slow

and takes more than 80% of the total alignment time for 150 base pairs (bp) long DNA reads. In contrast, our GPU accelerated API performs the packing on GPU which is up to 750x faster than NVBIO and, hence completely eliminating the sequence packing overhead.

3. In GASAL2 we compute 64 cells of the dynamic programming alignment matrix with no more than two GPU memory accesses, thereby achieving full utilization of GPU compute resources. The accesses to the traceback matrix are coalesced allowing maximum utilization of GPU memory bandwidth
4. GASAL2 GPU alignment kernels for alignment with traceback computation are 4x faster than NVBIO on an NVIDIA Geforce GTX 1080 Ti GPU. Combined with accelerated sequence packing, this gives an overall speedup of 7x for 150 bp reads performing the local alignment. Without traceback computation, the speedup is 8.7x.
5. GASAL2 gives 8x speedup over the fastest CPU based library (SeqAn) with traceback computation while performing local alignment of 150 bp read dataset. The speedup is 2.4x without traceback computation. SeqAn is running on 56 Intel Xeon threads.
6. We used GASAL2 to accelerate the seed-extension step of BWA-MEM DNA read mapper. It is more than 20x faster than the CPU seed-extension functions with 12 CPU threads giving nearly ideal speedup for 150 bp reads.
7. We observed that considerable time is wasted due to unequal loads on CPU threads due to batch processing required for accelerating BWA-MEM on GPU. We solved this problem by using dynamic batch sizing. Our approach reduces the idle CPU time by 3x for 150 bp reads in BWA-MEM GPU acceleration. Our proposed method is generic and can be applied to any implementation where batch processing is required.
8. Using GASAL2, we can accelerate the seed-extension step of Darwin, an algorithm to compute overlaps between long DNA reads. Using an NVIDIA Tesla K40 GPU, our GPU acceleration speeds up the whole Darwin algorithm by 109x and 24x running on 8 threads of Intel Xeon and 64 threads of IBM Power8, respectively.
9. The GPU acceleration of Darwin shows that GASAL2 is also suitable for performing local alignment of long DNA reads.

Chapter 4: Comparison of Seed-and-Extend Algorithms

This chapter is based on the comparison of seeding and extension algorithms used in the modern DNA read mappers. Different combinations of seeding and extension algorithms are tested to find suitable seed-and-extend techniques for various read lengths. The following conclusions can be drawn from the chapter.

1. We presented GASE (Generic Aligner for Seed-and-Extend), a generic read mapper based on the seed-and-extend technique. GASE contains various optimized and accelerated seeding and extension algorithms. Users can choose any combination of seeding and extension algorithm to suit their requirements of speed and accuracy.
2. We showed that for reads up to 600 bp long, exactly matching fixed length seeds is less accurate than the maximal exact matching seeds and fixed length seeds with at least 1 mismatch. We also showed that BLAST-like seed extension works best with the all-SMEM seeding approach. However, local alignment is overall more accurate than BLAST-like seed extension.
3. According to our results, nov-SMEM seeding with SIMD optimized local alignment seed-extension is the best combination due to its high accuracy and more potential for optimization/acceleration for future DNA reads.

Chapter 5: Predictive Genome Analysis

DNA sequencing delay is a major bottleneck in DNA analysis pipelines. In Chapter 5, we described methods to reduce the sequencing delay. In the following, we list the conclusions drawn from this chapter.

1. We proposed methods to start the computational analysis before the sequencing is completely finished, thereby reducing the sequencing time, and predicted the unknown bases and their corresponding quality scores.
2. We presented base prediction schemes and found the consensus approach to be the best. Similarly, we predicted the quality scores by extrapolating the average base quality profile.
3. We found that for 100 bp paired-end read data with coverage of 150x, 77% of the total reads with the last 50 bases unknown are predicted correctly with our proposed scheme.
4. We measured the area under the precision-recall curve (APR) of the results generated by the GATK variant calling pipeline using our predicted reads as input. The APR is 90% for 150x coverage 100 bp paired-end data with the last 50 bases predicted with our scheme.
5. To measure the maximum possible accuracy of our approach, we compared the APR values obtained by our prediction schemes with the APR values with ideal prediction. We found that with 100% correct prediction of the last 50

unknown bases will help to increase the APR from 90% to 94%. On the other hand, our scheme for predicting the base quality scores is very accurate and no considerable gain in APR can be achieved by the ideal prediction of quality scores.

6.2. Future research directions

In this thesis, we presented high performance seeding, seed-extension, and sequence alignment algorithms. We also described methods to overcome the DNA sequencing delay in an analysis pipeline. Our proposed algorithms and methods have shown to provide good speedups over the existing schemes. However, our work has some limitations and it is still possible to achieve more performance and accuracy by enhancing the proposed solutions. In the following, we will give some recommendations for extending the algorithms and methods described in the thesis.

1. GPUs have 32-bit registers. Therefore, to maximize performance of our GPU accelerated seeding algorithm, GPUseed, only supports genomes that have 4 billion bases or below. However, many living organisms have much bigger genomes (e.g. many plants). Therefore, extending GPUseed for bigger genomes will make it applicable in the analysis of genomes larger than 4 billion bases.

2. In BWA-MEM, the suffix array is stored in an array where each entry has a size of 64 bits. However, only 33 bits of each entry are required. Therefore, packing more values in the array will help to improve the compression ratio and as a result help in speeding up the suffix array lookup.

3. GASAL2 contains GPU accelerated kernels for local, global and semi-global optimal alignment algorithms. However, many DNA analysis algorithms use non-optimal but faster alignment algorithms. Therefore, extending GASAL2 to include the GPU acceleration of these non-optimal alignment algorithms will enhance the applicability of the library.

4. The acceleration of Darwin in Chapter 3 shows that our GPU accelerated sequence alignment library for NGS data can also be applied to align long DNA reads. However, the Darwin sequence alignment algorithm is non-optimal. Therefore, it would be interesting to investigate the efficacy of GPUs in the acceleration of optimal alignment algorithms for long DNA reads produced by third generation sequencers.

5. In Chapter 4, we compared the different seeding and extension algorithms in the context of short DNA read mappers. Extending this work for long DNA read mappers may prove to be beneficial in the analysis of sequencing data produced by Pacbio and Oxford Nanopore sequencers.

6. As described in Chapter 5, to overcome the sequencing delay without losing the accuracy, correctly predicting the unknown bases is essential. Applying machine learning algorithms may help to improve base prediction.

The constant improvement in sequencing technologies is accompanied by a reduction in the sequencing cost. This trend indicates a future where DNA sequencing can become part of household applications. This will give rise to a tremendous amount of sequencing data that needs to be processed efficiently. At the same time, cloud computing is on the rise. Hence, we see accelerated processing using GPUs and FPGA of DNA sequencing data in the cloud becoming more widespread in the future. This makes accelerated applications in the cloud a viable solution for future high performance genomics. Since state-of-the-art third-generation sequencing technologies generate long DNA reads which are easy to assemble, DNA read assembly might replace read mapping. Hence, more and more genome analysis pipelines based on read assembly will appear in the future. However, short DNA reads will remain in use for clinical purposes due to their much higher accuracy and even for research due to the easy availability of large amounts of sequencing data based on short DNA reads.

Bibliography

[1] Wetterstrand KA., *DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP),* `www.genome.gov/sequencingcosts` (2019), Accessed 5 April, 2019.

[2] Pacbio, *SMRT Sequencing,* `https://www.pacb.com/smrt-science/smrt-sequencing` (2019), Accessed 13 September, 2019.

[3] R. Bumgarner, *Overview of DNA Microarrays: Types, Applications, and Their Future,* Current Protocols in Molecular Biology **101**, 22.1.1 (2013).

[4] F. Sanger, S. Nicklen, and A. R. Coulson, *DNA sequencing with chain-terminating inhibitors,* Proceedings of the National Academy of Sciences **74**, 5463 (1977).

[5] Illumina, *NovaSeq System Specifications,* `https://emea.illumina.com/systems/sequencing-platforms/novaseq/specifications.html` (2019), Accessed 2 January, 2019.

[6] S. Shaikh and O. Sumant, *Global DNA Sequencing Market,* `www.alliedmarketresearch.com/dna-sequencing-market` (2018), Accessed 27 August, 2019.

[7] K. C. A. Chan, P. Jiang, K. Sun, Y. K. Y. Cheng, Y. K. Tong, S. H. Cheng, A. I. C. Wong, I. Hudecova, T. Y. Leung, R. W. K. Chiu, and Y. M. D. Lo, *Second generation noninvasive fetal genome analysis reveals de novo mutations, single-base parental inheritance, and preferred DNA ends,* Proc Natl Acad Sci U S A **113**, E8159 (2016).

[8] C. Gonzaga-Jauregui, J. R. Lupski, and R. A. Gibbs, *Human genome sequencing in health and disease,* Annu Rev Med **63**, 35 (2012).

[9] R. Kamps, R. D. Brandão, B. J. v. d. Bosch, A. D. C. Paulussen, S. Xanthoulea, M. J. Blok, and A. Romano, *Next-Generation Sequencing in Oncology: Genetic Diagnosis, Risk Prediction and Cancer Classification,* Int J Mol Sci **18**, 308 (2017).

[10] D. Edwards and J. Batley, *Plant genome sequencing: applications for crop improvement,* Plant Biotechnology Journal **8**, 2 (2010).

[11] G. Gorjanc, M. A. Cleveland, R. D. Houston, and J. M. Hickey, *Potential of genotyping-by-sequencing for genomic selection in livestock populations,* Genetics Selection Evolution **47**, 12 (2015).

[12] W. Gu, S. Miller, and C. Y. Chiu, *Clinical metagenomic next-generation sequencing for pathogen detection,* Annu Rev Pathol **14**, 319 (2019).

[13] C. Sekse, A. Holst-Jensen, U. Dobrindt, G. S. Johannessen, W. Li, B. Spilsberg, and J. Shi, *High Throughput Sequencing for Detection of Foodborne Pathogens,* Frontiers in Microbiology **8**, 2029 (2017).

[14] M. Roberts, W. Hayes, B. R. Hunt, and S. M. Mount, *Reducing storage requirements for biological sequence comparison,* Bioinformatics **20**, 3363 (2004).

[15] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, *Alignment of whole genomes,* Nucleic Acids Research **27**, 2369 (1999).

[16] T. Smith and M. Waterman, *Identification of common molecular subsequences,* Journal of Molecular Biology **147**, 195 (1981).

[17] S. B. Needleman and C. D. Wunsch, *A general method applicable to the search for similarities in the amino acid sequence of two proteins,* Journal of Molecular Biology **48**, 443 (1970).

[18] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, *Basic local alignment search tool,* Journal of Molecular Biology **215**, 403 (1990).

[19] H. Li, *Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM,* arXiv (2013).

[20] B. Langmead and S. S., *Fast gapped-read alignment with Bowtie 2,* Nature Methods **9**, 357 (2012).

[21] H. Li, *Minimap2: pairwise alignment for nucleotide sequences,* Bioinformatics **34**, 3094 (2018).

[22] Y. Turakhia, G. Bejerano, and W. J. Dally, *Darwin: A Genomics Co-processor Provides Up to 15,000X Acceleration on Long Read Assembly,* in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18 (2018) pp. 199–213.

[23] G. Myers, *Efficient local alignment discovery amongst noisy long reads,* in *Algorithms in Bioinformatics* (2014) pp. 52–67.

[24] G. Marçais, A. L. Delcher, A. M. Phillippy, R. Coston, S. L. Salzberg, and A. Zimin, *MUMmer4: A fast and versatile genome alignment system,* PLOS Computational Biology **14**, 1 (2018).

[25] R. S. Harris, *Improved Pairwise Alignment of Genomic Dna*, Ph.D. thesis, Pennsylvania State University, USA (2007).

[26] G. Miclotte, M. Heydari, P. Demeester, S. Rombauts, Y. Van de Peer, P. Audenaert, and J. Fostier, *Jabba: hybrid error correction for long sequencing reads,* Algorithms for Molecular Biology **11**, 10 (2016).

[27] R. Poplin, V. Ruano-Rubio, M. A. DePristo, T. J. Fennell, M. O. Carneiro, G. A. Van der Auwera, D. E. Kling, L. D. Gauthier, A. Levy-Moonshine, D. Roazen, K. Shakir, J. Thibault, S. Chandran, C. Whelan, M. Lek, S. Gabriel, M. J. Daly, B. Neale, D. G. MacArthur, and E. Banks, *Scaling accurate genetic variant discovery to tens of thousands of samples,* bioRxiv (2017).

[28] R. Poplin, P.-C. Chang, D. Alexander, S. Schwartz, T. Colthurst, A. Ku, D. Newburger, J. Dijamco, N. Nguyen, P. T. Afshar, S. S. Gross, L. Dorfman, C. Y. McLean, and M. A. DePristo, *A universal SNP and small-indel variant caller using deep neural networks,* Nature Biotechnology **36**, 983 (2018).

[29] S. Kim, K. Scheffler, A. L. Halpern, M. A. Bekritsky, E. Noh, M. Källberg, X. Chen, Y. Kim, D. Beyter, P. Krusche, and C. T. Saunders, *Strelka2: fast and accurate calling of germline and somatic variants,* Nature Methods **15**, 591 (2018).

[30] L. Hasan, M. Kentie, and Z. Al-Ars, *DOPA: GPU-based protein alignment using database and memory access optimizations,* BMC Research Notes **4**, 261 (2011).

[31] NVIDIA, *NVIDIA Tesla P100,* `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf` (2016), Accessed 13 September, 2019.

[32] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, *et al.*, *Big Data: Astronomical or Genomical?* PLoS Biology **13**, e1002195 (2015).

[33] H. Mushtaq, F. Liu, C. Costa, G. Liu, P. Hofstee, and Z. 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 (2017) pp. 148–157.

[34] 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, A. McKenna, T. J. Fennell, A. M. Kernytsky, A. Y. Sivachenko, K. Cibulskis, S. B. Gabriel, D. Altshuler, and M. J. Daly, *A framework for variation discovery and genotyping using next-generation dna sequencing data,* Nature Genetics **43**, 491 (2011).

[35] C. Herzeel, P. Costanza, D. Decap, J. Fostier, and J. Reumers, *elPrep: High-Performance Preparation of Sequence Alignment/Map Files for Variant Calling,* PLOS ONE **10**, 1 (2015).

[36] D. Decap, J. Fostier, J. Reumers, C. Herzeel, and P. Costanza, *Halvade: scalable sequence analysis with MapReduce,* Bioinformatics **31**, 2482 (2015).

[37] M. Ehrhardt, R. Rahn, K. Reinert, S. Budach, P. Costanza, and J. Hancox, *Generic accelerated sequence alignment in SeqAn using vectorization and multi-threading,* Bioinformatics **34**, 3437 (2018).

[38] J. Daily, *Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments,* BMC Bioinformatics **17**, 1 (2016).

[39] L. Hasan and Z. Al-Ars, *An overview of hardware-based acceleration of biological sequence alignment,* in *Computational Biology and Applied Bioinformatics* (IntechOpen, Rijeka, 2011) Chap. 9.

[40] E. J. Houtgast, V.-M. Sima, K. Bertels, and Z. Al-Ars, *An Efficient GPU Accelerated Implementation of Genomic Short Read Mapping with BWA-MEM,* SIGARCH Comput. Archit. News **44**, 38 (2017).

[41] E. J. Houtgast, V. Sima, K. Bertels, and Z. Al-Ars, *An FPGA-based systolic array to accelerate the BWA-MEM genomic mapping algorithm,* in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)* (2015) pp. 221–227.

[42] S. Ren, K. Bertels, and Z. Al-Ars, *Efficient Acceleration of the Pair-HMMs Forward Algorithm for GATK HaplotypeCaller on Graphics Processing Units,* Evolutionary Bioinformatics **14**, 1176934318760543 (2018).

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

[44] M. Vasimuddin, S. Misra, H. Li, and S. Aluru, *Efficient Architecture-Aware Acceleration of BWA-MEM for Multicore Systems,* in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2019) pp. 314–324.

[45] J. Zhang, H. Lin, P. Balaji, and W. Feng, *Optimizing Burrows-Wheeler Transform-Based Sequence Alignment on Multicore Architectures,* in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing* (2013) pp. 377–384.

[46] N. Ahmed, V. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars, *Heterogeneous hardware/software acceleration of the BWA-MEM DNA alignment algorithm,* in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2015) pp. 240–246.

[47] E. Seeberg and T. Rognes, *Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors ,* Bioinformatics **16**, 699 (2000).

[48] J. W. Fickett, *Fast optimal alignment,* Nucleic Acids Research **12**, 175 (1984).

[49] T. Ahmad, J. Peltenburg, N. Ahmed, and Z. Al-Ars, *ArrowSAM: In-Memory Genomics Data Processing through Apache Arrow Framework,* bioRxiv (2019), 10.1101/741843.

[50] M. Vyverman, B. De Baets, V. Fack, and P. Dawyndt, *essaMEM: finding maximal exact matches using enhanced sparse suffix arrays,* Bioinformatics **29**, 802 (2013).

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

[52] J. Pantaleoni and N. Subtil, *NVBIO,* `nvlabs.github.io/nvbio` (2015), Accessed 1 October, 2017.

[53] P. Ferragina and G. Manzini, *Opportunistic data structures with applications,* in *Proceedings 41st Annual Symposium on Foundations of Computer Science* (2000) pp. 390–398.

Summary

Recent advances in DNA sequencing technology have opened new doors for scientists to use genomic data analysis in a variety of applications that directly affect human lives. However, the analysis of unprecedented volumes of sequencing data being produced represents a formidable computational challenge. The conventional CPU-only computing paradigm is not sufficient to analyze exponentially growing sequencing data in a cost-effective and timely manner. Heterogeneous computing systems with GPU and FPGA based accelerators have become easily accessible and are increasingly being used to process massive amounts of data due to their better performance-to-cost ratio than CPU-only platforms. Furthermore, highly optimized analysis algorithms are required to extract the maximum computational power of these computing systems.

Seed-and-extend based algorithms are widely used for the analysis of high throughput DNA sequencing data. These algorithms have a two-step process: seeding and extension. In this thesis, we have presented optimized and accelerated seeding and extension algorithms specifically designed to analyze massive sequencing data. We developed GPUseed, a GPU accelerated API to compute maximal exact matching seeds. We extracted maximum parallelism from the computation and applied various optimizations to achieve up to 9x speedup over a multi-threaded CPU based algorithm. We also optimized the memory-bound CPU based algorithm to compute super-maximal exact matching seeds to achieve 45% fewer memory accesses to speed up the algorithm by 1.7x. In addition, we proposed GASAL2, a GPU accelerated library to perform all types of sequence alignment algorithms in the extension step. The library fully utilizes the compute resources of the GPU by minimizing GPU memory accesses and efficient utilization of high GPU memory bandwidth. GASAL2 minimizes the CPU wait time by allowing overlap CPU and GPU execution. It achieves up to 20x speedup over a highly optimized SIMD based CPU library running on tens of threads. GASAL2 is an open source API and can be used in any bioinformatics application. As a case study, we used GASAL2 to accelerate the extension step of BWA-MEM, the most widely used DNA read mapper, to achieve nearly ideal speedup as dictated by Amdahl's law. We then extended our GPU acceleration of sequence alignment to accelerate Darwin read overlapping algorithm designed to compute overlaps between state-of-the-art third generation long DNA reads. The GPU acceleration is orders of magnitude faster than the multi-threaded Darwin CPU implementation. Next, we developed GASE a generic read mapper for seed-and-extend that contains various optimized and accelerated seeding and extension algorithms. Users can choose any algorithm for the seeding and extension

step to build a mapper that suits their requirements. Using GASE, we showed that combining non-overlapping maximal exact matches with local alignment will give the best results in terms of both speed and accuracy for long DNA reads.

The process of DNA sequencing itself is a time consuming task and requires 3-4 days to complete. The computation analysis has to wait for the sequencing to complete to start processing the sequencing data to perform specific analysis. Therefore, we developed methods to overcome the sequencing delay which helps to reduce the sequencing time by a day but still achieves 90% accuracy.

Samenvatting

Recente ontwikkelingen in DNA-afleestechnologie maken het mogelijk voor wetenschappers om genetische gegevensanalyse te gebruiken in toepassingen die grote invloed kunnen hebben op mensenlevens. Echter zijn er enorme computationele uitdagingen om deze grote volumes sequentiegegevens te analyseren. Conventionele rekenmethoden zoals het gebruiken van processoren is niet voldoende om de groeiende hoeveelheid gegevens te analyseren op een effectieve manier. Om deze uitdaging aan te gaan kunnen analyse-algoritmen versneld worden door het gebruik maken van gespecialiseerde processoren zoals GPU en FPGA samen met andere traditionele processoren in een zogeheten heterogeen computer systeem. Verder is het belangrijk om deze analyse-algoritmen te optimaliseren om de efficiëntie van heterogene computersystemen te maximaliseren.

Seed-and-extend-gebaseerde algoritmen worden veel gebruikt voor de analyse van DNA-sequentiegegevens. Deze algoritmen worden in twee stappen uitgevoerd: seeding en extention. In dit proefschrift hebben wij deze twee stappen geoptimaliseerd en versneld. Wij hebben GPUseed ontwikkeld, een GPU versnelde API om complexe seeding algoritmes snel te berekenen. Hiervoor hebben wij het parallellisme van de algoritme gemaximaliseerd en verschillende optimalisaties toegepast en hiermee een versnelling van 9x hebben bereikt. Verder hebben wij het geheugenverbruik van het algoritme geoptimaliseerd om geheugentoegang met 45% vermindert en hiermee een versnelling van 1.7x hebben bereikt. Daarnaast hebben wij GASAL2 voorgesteld, een GPU-bibliotheek voor het versnellen van verschillende sequentie analyse-algoritmen tijdens de extention stap. Deze bibliotheek maakt efficiënt gebruik van GPU onderdelen door toegang tot het GPU-geheugen te minimaliseren en daarmee een hoge geheugenbandbreedte te bereiken. Verder minimaliseert GASAL2 de CPU-wachttijd door CPU en GPU operaties te overlappen. Deze optimalisaties maken het mogelijk voor GASAL2 een versnelling tot 20x te bereiken. GASAL2 is open en publiek beschikbaar en kan worden gebruikt door derde partijen in voor verschillende bio-informatica applicaties. Als case study hebben wij GASAL2 gebruikt om de extention stap van BWA-MEM, de meest gebruikte DNA analyse-algoritme, te versnellen. Daarnaast hebben wij GASAL2 gebruikt om de sequentie-overlap algoritme in het programma Darwin te versnellen. Darwin wordt gebruikt voor DNA analyse van ultramodern derde generatie DNA-aflees machines. De GPU-versnelling is tientallen malen sneller dan de snelste bestaande Darwin CPU-implementatie. Vervolgens hebben we GASE ontwikkeld, een generieke DNA analyse-algoritme voor seed-and-extension die verschillende geoptimaliseerde en meerdere seeding- en extention-algoritmen bevat. Gebruikers kunnen combinaties van deze algoritmen kiezen en eigen variaties toevoegen om de nauwkeurigheid te testen.

Als laatste hebben wij in dit proefschrift het DNA-afleesproces geoptimaliseerd. Dit proces zelf neemt veel tijd in beslag en duurt 3 tot 4 dagen. Om dit proces te versnellen hebben wij een methode ontwikkeld om de DNA analyse-algoritmen te laten starten voor dat het afleesproces volledig afgerond is. Hiermee hebben wij het aflees tijd met een hele dag weten te verminderen en tegelijkertijd een hoge nauwkeurigheid van 90% behouden.

List of Publications

Publications related to this thesis

International Journals

N. Ahmed, T. D. Qiu, K. Bertels, and Z. Al-Ars, *GPU Acceleration of Darwin Read Overlapper for de Novo Assembly of Long DNA Reads*, (2020), Accepted for publication in *BMC Systems Biology*.

N. Ahmed, K. Bertels, and Z. Al-Ars, *GPUseed: Fast Computation of Maximal Exact Matches for Genome Analysis*, (2020), Submitted to *MDPI Genes*.

N. Ahmed, J. Lévy, S. Ren, H. Mushtaq, K. Bertels and Z. Al-Ars, *GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data*, BMC Bioinformatics 20, 520 (2019).

International Conferences

N. Ahmed, K. Bertels, and Z. Al-Ars, *Efficient GPU Acceleration for Computing Maximal Exact Matches in Long DNA Reads*, (2020), in *2020 10th International Conference on Bioscience, Biochemistry and Bioinformatics (ICBBB)*.

N. Ahmed, H. Mushtaq, K. Bertels, and Z. Al-Ars, *GPU Accelerated API for Alignment of Genomics Sequencing Data*, in *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)* (2017) pp. 510–515

N. Ahmed, K. Bertels, and Z. Al-Ars, *Predictive Genome Analysis using Partial DNA Sequencing Data*, in *2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE)* (2017) pp. 119–124.

N. Ahmed, K. Bertels, and Z. Al-Ars, *A Comparison of Seed-and-Extend techniques in Modern DNA Read Alignment Algorithms*, in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)* (2016) pp. 1421–1428.

N. Ahmed, V. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars, *Heterogeneous Hardware/Software Acceleration of the BWA-MEM DNA Alignment Algorithm*, in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2015) pp. 240–246.

Patents

Z. Al-Ars, **N. Ahmed**, and K. Bertels, *Early DNA Analysis Using Incomplete DNA Datasets*, (2017), European Patent (NL2017750).

Other publications related to genomics

International Journals

T. Ahmad, **N. Ahmed**, and Z. Al-Ars, *Optimizing Performance of GATK Workflows using Apache Arrow In-Memory Data Framework*, (2020), Accepted for publication in *BMC Genomics*.

H. Mushtaq, **N. Ahmed**, and Z.Al-Ars, *SparkGA2: Production-quality memory-efficient Apache Spark based genome analysis framework*, PLOS ONE **14**, 1 (2019).

S. Ren, **N. Ahmed**, K. Bertels, and Z. Al-Ars, *GPU accelerated sequence alignment with traceback for GATK HaplotypeCaller*, BMC Genomics **20**, 184 (2019).

International Conferences

S. Ren, **N. Ahmed**, K. Bertels, and Z. Al-Ars, *An Efficient GPU-Based de Bruijn Graph Construction Algorithm for Micro-Assembly*, in *2018 IEEE 18th International Conference on Bioinformatics and Bioengineering (BIBE)* (2018) pp. 67–72.

H. Mushtaq, **N. Ahmed**, and Z. Al-Ars, *Streaming Distributed DNA Sequence Alignment using Apache Spark*, in 2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE) (2017) pp. 188–193.

Curriculum Vitae

Nauman Ahmed

Nauman Ahmed was born in Lahore, Pakistan on 1 June 1981. He obtained his Bachelor's degree in Electrical Engineering in 2004 from the University of Engineering and Technology Lahore, Pakistan. In the same year, he joined the University of Engineering and Technology Lahore as a Lecturer. He obtained a Master's degree in Electrical Engineering in 2010 from the same university. In 2012, he was promoted to Assistant Professor. He was awarded the Faculty Development Program scholarship by the university in 2012. In 2014, he started his PhD in the Computer Engineering Laboratory of the Delft University of Technology, Netherlands under the supervision of Dr. Zaid Al-Ars. His research paper titled, ”Heterogeneous Hardware/Software Acceleration of the BWA-MEM DNA Alignment Algorithm”, published in the 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) was awarded HiPEAC Tech Transfer Award 2015. His research interests are high performance genomics and computer architecture.