Newcastle University, UK

School of Engineering (SoE)

Algorithm-Hardware Co-Design for Performance-driven Embedded Genomics

PhD Thesis

Sidharth Maheshwari

November 14, 2021

Abstract

Genomics includes development of techniques for diagnosis, prognosis and therapy of over 6000 known genetic disorders. It is a major driver in the transformation of medicine from the reactive form to the personalized, predictive, preventive and participatory (P4) form. The availability of genome is an essential prerequisite to genomics and is obtained from the sequencing and analysis pipelines of the whole genome sequencing (WGS). The advent of second generation sequencing (SGS), significantly, reduced the sequencing costs leading to voluminous research in genomics. SGS technologies, however, generate massive volumes of data in the form of reads, which are fragmentations of the real genome. The performance requirements associated with mapping reads to the reference genome (RG), in order to reassemble the original genome, now, stands disproportionate to the available computational capabilities. Conventionally, the hardware resources used are made of homogeneous many-core architecture employing complex general-purpose CPU cores. Although these cores provide high-performance, a data-centric approach is required to identify alternate hardware systems more suitable for affordable and sustainable genome analysis.

Most state-of-the-art genomic tools are performance oriented and do not address the crucial aspect of energy consumption. Although algorithmic innovations have reduced runtime on conventional hardware, the energy consumption has scaled poorly. The associated monetary and environmental costs have made it a major bottleneck to translational genomics. This thesis is concerned with the development and validation of read mappers for embedded genomics paradigm, aiming to provide a portable and energy-efficient hardware solution to the reassembly pipeline. It applies the algorithm-hardware co-design approach to bridge the saturation point arrived in algorithmic innovations with emerging low-power/energy heterogeneous embedded platforms.

Essential to embedded paradigm is the ability to use heterogeneous hardware resources. Graphical processing units (GPU) are, often, available in most modern devices

alongside CPU but, conventionally, state-of-the-art read mappers are not tuned to use both together. The first part of the thesis develops a Cross-platfOrm Read mApper using opencL (CORAL) that can distribute workload on all available devices for high performance. OpenCL framework mitigates the need for designing separate kernels for CPU and GPU. It implements a verification-aware filtration algorithm for rapid pruning and identification of candidate locations for mapping reads to the RG.

Mapping reads on embedded platforms decreases performance due to architectural differences such as limited on-chip/off-chip memory, smaller bandwidths and simpler cores. To mitigate performance degradation, in second part of the thesis, we propose a REad maPper for heterogeneoUs sysTEms (REPUTE) which uses an efficient dynamic programming (DP) based filtration methodology. Using algorithm-hardware co-design and kernel level optimizations to reduce its memory footprint, REPUTE demonstrated significant energy savings on HiKey970 embedded platform with acceptable performance.

The third part of the thesis concentrates on mapping the whole genome on an embedded platform. We propose a Pyopencl based tooL for gEnomic workloaDs tarGeting Embedded platfoRms (PLEDGER) which includes two novel contributions. The first one proposes a novel preprocessing strategy to generate low-memory footprint (LMF) data structure to fit all human chromosomes at the cost of performance. Second contribution is LMF DP-based filtration method to work in conjunction with the proposed data structures. To mitigate performance degradation, the kernel employs several optimisations including extensive usage of bit-vector operations. Extensive experiments using real human reads were carried out with state-of-the-art read mappers on 5 different platforms for CORAL, REPUTE and PLEDGER. The results show that embedded genomics provides significant energy savings with similar performance compared to conventional CPU-based platforms.

Contents

List of Figures **iv**

List of Tables **viii**

Acknowledgments **xiv**

Publications **1**

1 Introduction **2**

1.1 Motivation 2

1.2 Hypotheses 7

1.3 Contributions 7

1.4 Thesis layout 9

2 Background **11**

2.1 Whole Genome Sequencing 13

2.1.1 Genome 13

2.1.2 Overview of WGS 15

2.2 Genome Re-assembly 19

2.3 *De novo* Assembly 23

2.3.1 Overlap-Layout-Concensus (OLC) 23

2.3.2 *De Bruijn Graph* 24

2.3.3 String Graph 27

2.3.4 Shortcomings of *de novo* Assembly Approach 28

2.4 Read-Alignment Approach 28

2.4.1 Preprocessing 30

2.4.2 Filtering . . . 34
2.4.3 Verification . . . 37
2.4.4 Previous work . . . 38
2.5 Algorithm-Hardware Co-Design . . . 45
2.6 Summary . . . 46

3 Verification-aware Read Mapper for Heterogeneous Systems 49
3.1 Overview . . . 49
3.2 Background . . . 51
3.3 Methods . . . 54
3.3.1 OpenCL view of the hardware . . . 54
3.3.2 Preprocessing . . . 58
3.3.3 Verification-Aware Filtration . . . 59
3.3.4 Implementation of Myers Bitvector Algorithm . . . 63
3.3.5 Kernel Algorithm: Search and Verification . . . 65
3.4 Experimental Results . . . 67
3.4.1 Experimental setup . . . 67
3.4.2 Results . . . 71
3.5 Discussion . . . 79
3.6 Summary . . . 83

4 Dynamic Programming based Filtration 84
4.1 Introduction . . . 84
4.2 Methodology . . . 86
4.2.1 Preprocessing and Verification . . . 86
4.2.2 Dynamic Programming based Filtration . . . 87
4.3 Algorithm flowchart . . . 90
4.4 Experimental Setup . . . 91
4.4.1 Homogeneous Scenario . . . 92
4.4.2 Heterogeneous Scenario . . . 92
4.4.3 Embedded Scenario . . . 92
4.4.4 Power and Energy Consumption . . . 93

4.5 Results and Discussion . . . 93
4.6 Summary . . . 100

5 Embedded Whole Genome Read Mapping **102**
5.1 Introduction . . . 102
5.2 Methodology . . . 104
5.2.1 Memory-aware preprocessing . . . 107
5.2.2 Filtration for memory aware data structures . . . 108
5.3 Experimental setup . . . 108
5.4 Results and discussion . . . 110
5.4.1 System 1 - CPU+GPU . . . 110
5.4.2 System 2 - Odroid N2 . . . 111
5.4.3 Accuracy . . . 116
5.4.4 Power and energy consumption . . . 116
5.4.5 Performance gap and future work . . . 117
5.4.6 Challenges with OpenCL . . . 118
5.5 Summary . . . 119

6 Conclusion **120**
6.1 Contributions . . . 120
6.2 Future Work . . . 123

Bibliography **127**

List of Figures

1.1 Decreasing cost of sequencing per human genome. 3
1.2 A comparison between growth in genomic data versus improvements in computational capabilities 4

2.1 Microscopic visualisation of genome with chromosomes expanded to level of nucleotide bases which constitute the entire genome. 12
2.2 A visualisation of the DNA's double helix structure in the form of forward and reverse strand. 13
2.3 A visualisation of the whole genome sequencing process depicting sequencing, re-assembly and analysis pipelines. 14
2.4 A demonstration of the sequencing process with visualisations of amplified template, cyclic reversible termination using PCR and imagine to capture the reads. 17
2.5 An illustration of the polymerase chain reaction (PCR) while generating copies of the DNA molecule. Denaturation splits the double stranded DNA molecule into two single strands and then PCR completes the opposite strand generating identical copies of the original DNA molecule, using nucleotide bases (dNTPs), primer and polymerase enzyme. 18
2.6 A comparative visualization of the types of reads that are often generated from the sequencing processes. Typically, there are three types are reads: (a) short, (b) paired-end and (c) long reads. Repetitive sections are prevalent in genomes, with approximately 50% makeup of human and over 80% makeup of the maize genome being repeats. Repeats posit a major challenge in genome re-assembly and the type of reads being mapped can affect the mapping accuracy. 21

2.7 An overview of the workflow of *de novo* assembly approach. There are three main stages: (i) contig assembly, (ii) scaffolding and (iii) gap filling. Contigs are contiguous genomic segments, free of any gaps, obtained using overlapping segments in reads. Scaffolding are ordered and oriented contigs arranged using paired-end reads as anchors. Gap-filling involves resolving unidentified bases using consensus based on independent reads. These gaps may be important SNPs and variations. 22
2.8 A visualisation of the working-principle of *de novo* assembly approaches with the help of four unique regions (blue, violet, green and yellow) and two copies of repeated region (red). (a) Overlap-Layout-Consensous (OLC), (b) *de Bruijn* graph approach, and (c) String graphs [1]. 25
2.9 A visualisation of k-mers and q-grams. k-mers are non-overlapping genomic sections of length k while q-grams are consecutive overlapping sections of length q. q-grams are, often, referred as overlapping k-mers. . . 26
2.10 A visualization of read mapping using the read-alignment approach. . . . 29
2.11 Read mapping stages in the read-alignment approach using approximate string matching algorithms. 30
2.12 Visualisation of the hashing process by converting *q-grams*, derived from the reference genome, to numerical indexes using hash functions. All positions of occurrence of a *q-gram* in the reference genome is appended in the entries linked to the corresponding index. 31
2.13 Visualisation of the pigeonhole principle where a read is divided into six equal length non-overlapping *k-mers*. If the read needs to be mapped with an edit distance of five then the pigeonhole principle states that one of the six *k-mers* will be error free and match exactly in the reference. 35
2.14 Visualisation of the q-gram lemma filtering approach used by RazerS3 [2]. This method counts the number of *q-grams* that match the projection of the parallelogram on the reference. If the count reaches a minimum threshold τ, then the projection is the candidate location. The reference is divided into equal-sized overlapping parallelograms and the process is repeated for each parallelogram. 36

2.15 State-of-the-art read mappers targeting different hardware platforms viz. FPGA, GPU and CPU. Segregation has been performed on basis of data structures used to store reference genome. CPU oriented mappers are further segregated into *best-mappers* and *all-mappers*. 38
2.16 Metrics that determine the algorithm-hardware co-design approach with the available design space choices. 45

3.1 OpenCL programming model with memory hierarchy. 54
3.2 Visualization of Burrows-Wheeler Transform and its usage in compression. 57
3.3 Visualization of preprocessing methodology of CORAL for a small sequence: `GAAATCGZATCATZACCGTG$` using FM-Index and suffix arrays. We store the tally matrix, suffix array and the modified **F** array to be used for querying *k-mers* in the filtration stage. 58
3.4 Visualization of the searching method using FM-Index and suffix arrays. We search for pattern: `ATC` in the text: `GAAATCGZATCATZACCGTG$` in three cycles and then show how the search can be extended if the pattern size increases on dynamically to `AATC`, using four cycles. 60
3.5 Algorithm for the CORAL kernel. 65
3.6 Average number of verifications per read using different filtration schemes for real data sets, viz. ERR012100_1 ($n = 100, \delta = 5$) and SRR826460_1 ($n = 150, \delta = 7$), on chr2. NVA - non verification-aware, VA - verification-aware and VA+A - verification-aware along with approximation. The approximation used, here, limits the maximum number of verifications per *k-mer* to 1000. 79
3.7 The minimum amount of private memory, in bytes, used by each workitem in the CORAL kernel for different read lengths. 82

4.1 Demonstration of pigeonhole principle for ($n = 100, \delta = 5$), where n is read length and δ is error. *K-mers* with their respective number of candidate locations are displayed. The vertical lines are the optimal dividers, identified in filtration stage, to minimize the total number of candidate locations. The dots represent one of the four bases: {`A, C, G, T`}. 87

4.2 Demonstration of memory optimised dynamic programming based filtration algorithm for parameters ($n = 100, \delta = 5$). δ iterations are required to obtain optimal dividers for $\delta + 1$ *k-mers*. In the end, the optimal dividers are found starting from the last one using backtracking. 88
4.3 Flowchart of the REPUTE kernel. 90
4.4 Mapping time for different distributions of workloads on CPU and GPU for ($n = 150, \delta = 5$) and minimum *k-mer* length of 22. X-axis shows the number of reads, out of 1 million, mapped by each GPU and the remaining reads are mapped on the CPU. 98
4.5 Mapping time for different minimum *k-mer* lengths with same distributions of workloads on CPU and GPU. CPU mapped 820,000 reads and GPU mapped 90,000 reads each with the read configuration of ($n = 100, \delta = 4$). . 99

5.1 Overview of the read mapping process. There are three stages: Preprocessing, filtration and verification. Reference genome is the input to the preprocessing and verification stages. The are other inputs to the filtration stage such as reads file, read length, edit distance, k-mer length, constants, etc. 105
5.2 Demonstration of proposed low memory footprint (LMF) tally matrix along with the tally offset array to help obtain the missing information in the tally LMF matrix during filtration in $O(1)$ time. 106
5.3 Visualization of bit-vector operation to obtain the desired element of the tally matrix using tally LMF matrix and tally offset array. 109
5.4 Comparison between average number of candidate locations per read verified by PLEDGER and Hobbes3. Values presented for $n = 150$ and $\delta = 7$ 117
5.5 Linear normalization of mapping times for PLEDGER if it reports same number of candidate locations as Hobbes3. Values presented for $n = 150$ and $\delta = 7$. 118

List of Tables

3.1 Characteristics of state-of-the-art read mappers along with CORAL, proposed in this chapter. *Other* in the table implies a combination of multiple data structure, search algorithms and heuristics, the one, specifically, employed by BWA-MEM. 53

3.2 Charaterization of number of occurrences of *k-mers* for $k = 16, 17, 18, 19, 20, 21, 22$ in chromosome 2. Values given are in percentage approximated to the nearest decimal. 62

3.3 The results of mapping three sets of 500,000 simulated reads, with different maximum edit distances viz. 3,4 and 5, to chromosome (chr) 2 on the CPU. RazerS3, Hobbes3, GEM and CORAL-cpu reported up to 3500 mapping per read on System 1 and 100 mapping locations per read on System 2. FEM, Yara, however, by default report all the mapping positions and BWA-MEM was configured to report all mapping positions. T and A are abbreviations for time and accuracy and have been reported in seconds and proportional percentage of accurate mapping against the total number of reads. 73

3.4 The results of mapping three sets of 500,000 simulated reads, with different maximum edit distances viz. 3,4 and 5, to chromosome (chr) 21. RazerS3, Hobbes3, GEM and CORAL-cpu reported up to 3500 mapping per read on System 1 and 100 mapping locations per read on System 2. CORAL-all, also, produce 100 mapping locations per read but executes on CPU and both the GPUs, simultaneously. FEM, Yara, however, by default report all the mapping positions and BWA-MEM was configured to report all mapping positions. T and A are abbreviations for time and accuracy and have been reported in seconds and proportional percentage of accurate mapping against the total number of reads. 74

3.5 The results of mapping 1M real reads to chromosome (chr) 2 on the CPU. RazerS3, Hobbes3, GEM and CORAL-cpu reported up to 3500 mapping per read on System 1 and 100 mapping locations per read on System 2. FEM, Yara, however, by default report all the mapping positions and BWA-MEM was configured to report all mapping positions. T and A are abbreviations for time and accuracy and have been reported in seconds and proportional percentage of accurate mapping against the total number of reads. 75

3.6 The results of mapping 1M real reads to chromosome (chr) 21. RazerS3, Hobbes3, GEM and CORAL-cpu reported up to 3500 mapping per read on System 1 and 100 mapping locations per read on System 2. CORAL-all, also, produce 100 mapping locations per read but executes on CPU and both the GPUs, simultaneously. FEM, Yara, however, by default report all the mapping positions and BWA-MEM was configured to report all mapping positions. T and A are abbreviations for time and accuracy and have been reported in seconds and proportional percentage of accurate mapping against the total number of reads. 76

4.1 The results of mapping 2M real reads to chromosome (chr) 21 on the CPU. T represents mapping time in seconds and A represents accuracy, measured in accordance with Section 4.4.1. Hobbes3, GEM and REPUTE-cpu reported up to 1000 mapping per read. Yara, however, by default reports all the mapping positions and BWA-MEM, being a *best-mapper*, is configured to report all mapping positions. RazerS3 is used as the gold standard with 100 outputs per read. 95
4.2 The results of mapping 2M real reads to chromosome (chr) 21 on the CPU + GPU. T represents mapping time in seconds and A represents accuracy, measured in accordance with Section 4.4.2. Hobbes3, GEM and REPUTE-all report up to 100 mapping per read. Here, REPUTE-all, however, distributes the workload on both CPU and GPUs. Yara, by default, reports all the mapping positions and BWA-MEM, being a *best-mapper*, is configured to report all mapping positions. RazerS3 is used as the gold standard and reports 100 outputs per read. 96
4.3 Read mapping on the HiKey970 SoC. T - time in seconds and A - accuracy, measured in accordance with Section 4.4.3. 97
4.4 Energy consumption in accordance with Section 4.4.4. 100

5.1 Mapping time(in seconds) for 1M real reads, of length $n = 100$ and error $\delta = 5$, to chromosome (chr) 1-22, X and Y on system - 1 (CPU+GPU). PLEDGER-all distributes workload over CPU and GPU in 4:1 ratio while others execute only on the CPU. 112
5.2 Mapping time(in seconds) for 1M real reads,, of length $n = 150$ and error $\delta = 7$,, to chromosome (chr) 1-22, X and Y on system - 1 (CPU+GPU). PLEDGER-all distributes workload over CPU and GPU in 4:1 ratio while others execute only on the CPU. 113
5.3 Mapping time(in seconds) for 1M real reads, of length $n = 100$ and error $\delta = 5$, to chromosome (chr) 1-22, X and Y on system - 2, Odroid N2 platform. 114
5.4 Mapping time(in seconds) for 1M real reads,, of length $n = 150$ and error $\delta = 7$,, to chromosome (chr) 1-22, X and Y on system - 2, Odroid N2 platform. 115
5.5 Energy consumption in accordance with Section 5.4.4. 116

List of Acronyms

ALU - Arithmetic Logic Unit
ASM - Approximate String Matching
ASIC - Application Specific Integrated Circuit
BWT - Burrows–Wheeler transform
CPU - Central Processing Unit
CKS - Cheap *K-mer* Selection
chr - Chromosome
CRT - Cyclic Reversible Termination
DSP - Digital Signal Processors
DP - Dynamic Programming
EOF - End-Of-File
ESM - Exact String Matching
FPGA - Field-Programmable Gate Arrays
GPU - Graphical Processing Unit
HW - Hardware
HDL - Hardware Description Language
HTS - High-Throughput Sequencing
HRG - Human Reference Genome
IoT - Internet of Things
LINE - Long Interspersed Nuclear Elements
LTR - Long Terminal Repeats
LMF - Low-Memory Footprint
MPSoC - Multiprocessor System-on-a-Chip
SGS - Second-Generation Sequencing
OpenCL - Open Computing Language

OLC - Overlap Layout Consensus

PCR - Polymerase Chain Reaction

P4 - Predictive, Preventive, Personalized, and Participatory

PIM - Process-In-Memory

RG - Reference Genome

SBC - Single Board Computer

SNP - Single-Nucleotide Polymorphism

SMS - Single-Molecule Sequencing

SINE - Short Interspersed Nuclear Elements

STR - Simple Tandem Repeats

SIMD - Single-Instruction Multiple-Data

WGS - Whole Genome Sequencing

Acknowledgments

I would like to express my gratitude to Rishad Shafik, my supervisor, for his wisdom, support, help and guidance throughout my postgraduate studies. He educated me to become a better researcher and engaged me in productive discussions that helped me develop novel ideas. He helped me learn how to write high quality research articles and encouraged me to participate in conferences and summer schools to enhance my knowledgebase. He has helped me whenever I have faced challenges in research and outside while living in a new and culturally different country. I would like to thank Alex Yakovlev, my second supervisor, for helping me improve the quality of my research. Through his vast experience and knowledge, he engaged me in numerous discussions and helped me gauge my ideas against their practical implementations and benefits. Rishad and Alex have helped me take my PhD in the right direction with a vision of making an impact in the society. They have been very understanding and supportive during the pandemic times which has been difficult times for everyone. I am grateful to Ian Wilson, my third supervisor, for his valuable inputs during discussions from a bioinformatician's perspective. My research is interdisciplinary in nature and Ian's help and feedback have been crucial to our implementations.

I, also, extend my thanks to Amit Acharyya from IIT Hyderabad, whose has been guiding me since my undergraduate. He sailed me through the entire research cycle starting from literature review to idea generation and to implementation and publishing a quality contribution. Along with his student Venkateshwarlu Gudur, we have performed collaborative research and published several articles together during my PhD. I would, also, like to thank Gudur for his help and contribution. I would like to thank Tousif Rehman for his contributions in evaluating OpenCL-based FPGA implementation of our proposed work. I would, also, like to thank the members of Microsystems laboratory, the administration and the office bearers of electrical engineering department

in the Merz court for their continuous support. I would like to thank the Newcastle University for providing me an opportunity to study in a beautiful, friendly and a wonderful environment.

I am thankful to my parents, Madhu Shree Chitlangia and Kishan Maheshwari, for all the love, support, prayers, patience and sacrifices they have made for me. I would like to thank my sister, Shikha Saboo, for her motherly love and support throughout my life. I would like to thank my wife, Shristy Pandit, who was always there to listen to me, help me, share love and advice throughout my PhD. I would like to thank my friends Abhinav Agarwal and Anurag Modi for always being there for me and helping me in all ways possible. I would like to thank all of my friends whom I cannot mention here for brevity. Lastly, I prostrate to my *Isth* Bhagwaan Shri Ram, who is *Parambrahm* himself and his will and grace guides my existence.

This research was supported by EPSRC, EPSRC STRATA and EPSRC DTP. My PhD scholarship was sponsored by SAgE Faculty Doctoral Training Awards Scheme (DTA) 2016 and School Research Scholarship (SRS) 2016.

Publications

Journal publications

1. **Sidharth Maheshwari**, V. Y. Gudur, R. Shafik, I. Wilson, A. Yakovlev and A. Acharyya, *CORAL: Verification-aware OpenCL based Read Mapper for Heterogeneous Systems*, in IEEE/ACM Transactions on Computational Biology and Bioinformatics, doi: 10.1109/TCBB.2019.2943856.

2. V. Y. Gudur, **Sidharth Maheshwari**, A. Acharyya, R. Shafik, *An FPGA based Energy-Efficient Read Mapper with Parallel Filtering and in-situ Verification*, in IEEE/ACM Transactions on Computational Biology and Bioinformatics, 2021. (Accepted)

Conference publications

1. **Sidharth Maheshwari**, R. Shafik, I. Wilson, A. Yakovlev and A. Acharyya, *REPUTE: An OpenCL based Read Mapping Tool for Embedded Genomics*, in Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 2020, pp. 121-126, doi: 10.23919/DATE48585.2020.9116238.

2. **Sidharth Maheshwari**, R. Shafik, I. Wilson, A. Yakovlev, V. Y. Gudur and A. Acharyya, *PLEDGER: Embedded Whole Genome Read Mapping using Algorithm-HW Co-design and Memory-aware Implementation*, in Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 2021. (Accepted)

3. V. Y. Gudur, **Sidharth Maheshwari**, R. Shafik and A. Acharyya, *Accelerated Filtering and in situ Verification for Energy-Optimized Genome Read Mapping*, 2020 IEEE International Symposium on Circuits and Systems (ISCAS), Sevilla, 2020, pp. 1-5, doi: 10.1109/ISCAS45731.2020.9180620.

Workshops and summits

1. ACACES 2019: 15th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems. Fuiggi, Italy.

2. Arm Research Summit 2018, Cambridge, UK.

3. 2018 ACTION for Impact Programme: 3-day residential programme for budding innovators. To help researchers transform ideas to commercial products/services. Venue: Slaley Hall, Northumberland, UK.

4. 2017 4th International Summer School on Resource-Aware Machine Learning, TU Dortmund, Germany. Low-power and low-energy ML Hardware solutions. Courses involved both teaching and practical sessions.

5. 2016 Introduction to Learning and Teaching in Higher Education programme at Newcastle University. Required for demonstrating to undergraduate and masters students in laboratory sessions.

Chapter 1

Introduction

1.1 Motivation

With over 6000 single-gene genetic conditions and many other diseases involving genetic variants, medicine has become the foremost application of genomics. Genomics is central to the undergoing transformation of medicine from reactive to proactive forms. The envisaged proactive forms of medicine will be predictive, preventive, personalized, and participatory (P4) [3, 4]. P4 medicine involves the examination of an individual´s complete genetic makeup to predict health prospects. It requires tracking of vast number of samples for data analysis to find diagnostics and therapeutics for early reversal of the disease trajectory [5]. The aim of P4 medicine is early detection, prevention and developing new strategies for looking at diseases, making vaccines and providing affordable healthcare using personalized medicine [6, 7]. The intentions of P4 medicine, also, corroborates with the endeavor of translational genomics, which aims to adopt the discoveries made in genetic research to clinical practice and include whole genome sequencing (WGS) pipelines as a part of routine tests performed in the hospitals [8, 9]. To spur innovations for P4 medicine and enable translation of genetic understanding to healthcare, huge repositories of genetic data is needed from a wide variety of patients, in massive numbers. It will incur huge infrastructural costs to setup dedicated genome sequencing data centers along with development, maintenance and use of

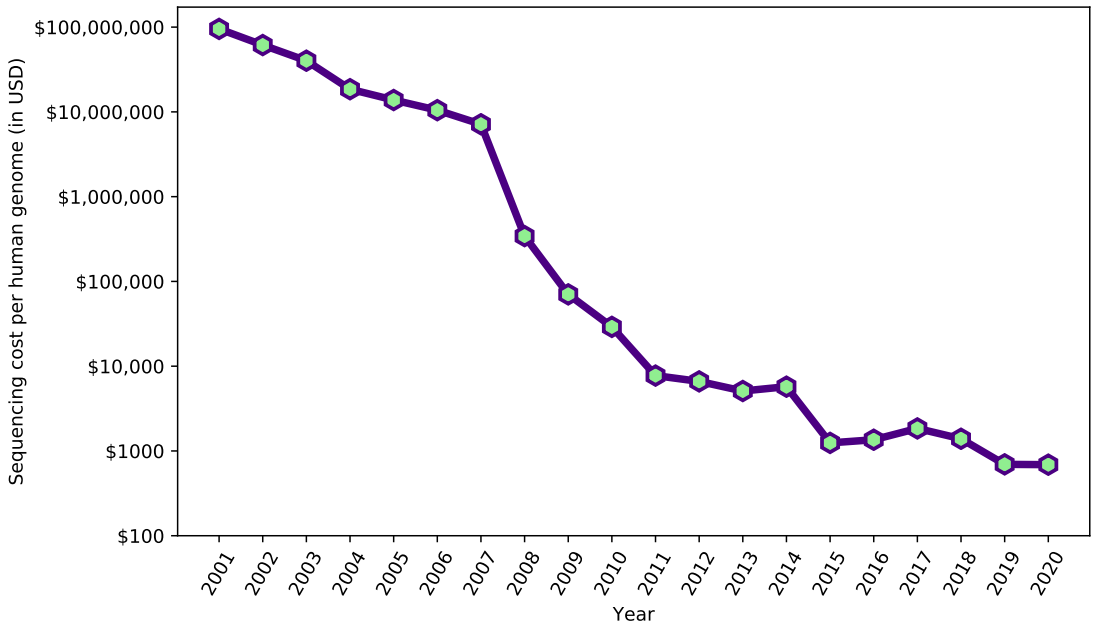

Figure 1.1: Decreasing cost of sequencing per human genome. The data was obtained from [10] and the sequencing costs are plotted in logarithmic scale.

massive computing facilities to process, analyze, store, transmit and integrate data [6]. Translation of genetic data will require development of advanced computational tools to reassemble the original genome of the individuals and run analysis cycles to extract information.

Obtaining genome is an essential prerequisite to genomics, which is performed using the sequencing machines. The decreasing cost of sequencing, since, the Human Genome project has played a significant role in enabling translational genomics, as shown in Fig. 1.1. With the advent of high-throughput sequencing (HTS) and the continued advancements in sequencing technologies, the cost of sequencing a human genome is reported to be $689 as of August 2020 [10]. This, however, does not account for 'non-production' activities, most notably, the genome assembly and analysis pipelines of the WGS. Even though HTS has enabled sequencing large numbers of individuals, it has made genomics one of the largest contributors to Big Data [11–13]. Recent trends indicate that genomics is on the path to become largest data producer in the coming decade with an estimated 100 million to 2 billion human genomes to be sequenced by 2025 [14]. The computational hours required to process the existing data already surpasses our

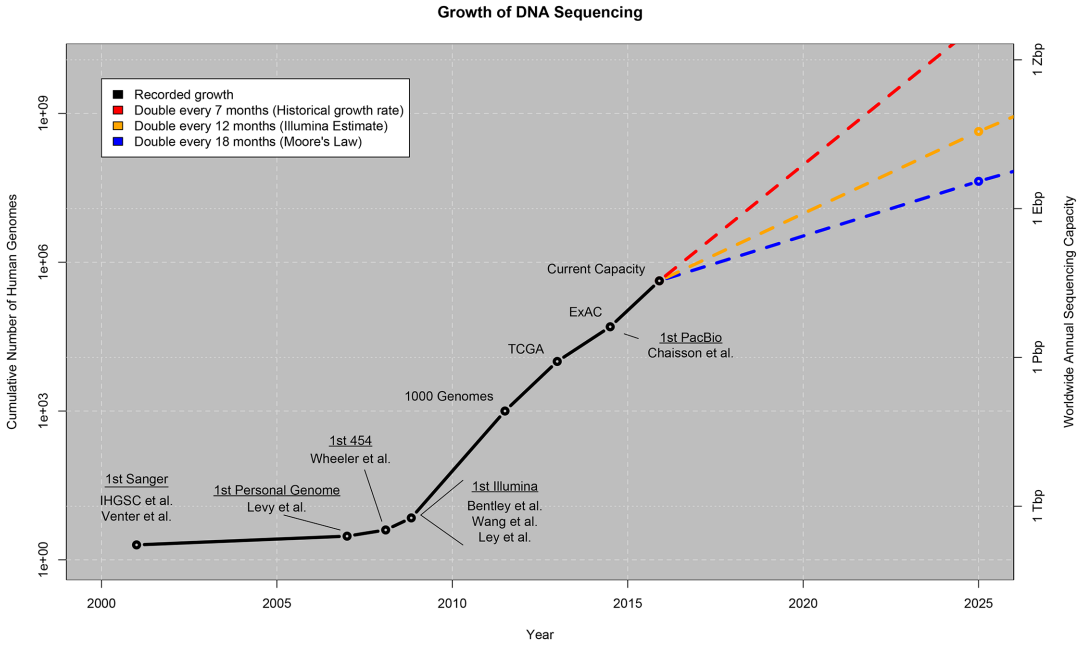

Figure 1.2: The plot shows recorded and estimated growth in the number of sequenced genomes and sequencing capacity compared to Moore's Law which indicates the improvements in computational capacity [13].

available computational capabilities [13, 15]. Fig. 1.2 shows the recorded and estimated growth in the number of genomes sequenced and sequencing capacity in comparison to the Moore's law, which indicates the improvements in computational capacity. To understand the computational requirements for accurate reassembly of human genome after the sequencing process, let's consider the NHGRI estimate [10] which suggests that mapping the human genome with over 3000 megabases (Mb; a million bases) will need a 30-fold coverage while using Illumina sequencing machines. It implies that to obtain the individual's genome, 90,000 Mb $\equiv$ 900 million single-end reads of length $n = 100$ needs to be mapped to the reference genome with an assumed error-rate of 5% i.e. edit distance $\delta = 5$. From our experiments using the Hobbes3 read mapper [16], we have found that it will require over 85 hrs on a workstation with Intel i7-8750H 6-core CPU. This excludes the downstream analysis that involves diverse range of approaches including the approximate string-matching algorithms [13]. We can see that the pace of innovation in genomic data creation is much higher than that of genomic informatics; this widening gap can be addressed with novel hardware solutions to computing genomic data.

Genomics has become a major contributor to Big Data with archives of raw sequencing data doubling every 18 months [17]. Studies based on large sequencing data are, largely, conducted centrally in genomic data centers or using cloud computing services provided by privately owned data centers. The new industrial giants, the tech firms such as Google, Alphabet, Amazon, Twitter and Facebook are establishing more and bigger data centers around the world to process Big Data. The rate of growth in electricity consumption of data centers is, already, higher than the growth in worldwide electricity consumption [18]. The compute units in the servers and the cooling systems together consume about 43% of total electricity consumption [19, 20]. This has led to research and adoption of energy-efficiency measures in cooling and power provisions systems and development of novel hardware and software solutions to make computing more energy-efficient [19, 21]. With the growing demand from genomics, both the economic and environmental considerations of establishing new computing infrastructures can no longer be ignored. The regional and remote locations, often, suffer from unreliable internet connectivity, limited facilities and funding, to establish data centers or high-performance servers to address computing and storage demands [22]. The processing cores used in servers and workstation are complex and optimized for floating-point operations while genomic computations are integer-based operations. Energy efficiency of data centers are only as good as that of the processors they are made of — and there is scope for improvement with the use of modern heterogeneous systems and low-power embedded platforms. With the aim of data-centric hardware implementation, the performance and energy tradeoffs of using simpler embedded cores for genomic computations needs to be determined. This can lead to using affordable high-performance embedded clusters for large-scale genomic computations with minimal programming effort.

There are various categories of hardware computing devices such as central processing unit (CPU), graphical processing unit (GPU), field-programmable gate arrays (FPGA) and digital signal processors (DSP). These devices can be found in off-the-shelf platforms, provided by a range of electronics manufacturers such as Intel, AMD, Nvidia, ARM and Xilinx. They can be found either solo or in different combinations such as CPU + GPU, CPU + GPU + FPGA or CPU + FPGA. Most modern computing

systems including embedded platforms and many supercomputers, are heterogeneous and have a combination of CPU + GPU available on the same platform [23]. On the other hand, state-of-the-art bioinformatics tools including the read mappers [2, 16, 24–29], have focused on algorithmic innovations and software optimizations targeting, mainly, the CPU. There are many communications available that have focused on acceleration of genomic algorithms on either GPU or FPGA, as summarized in [30]. It is, however, arduous and challenging to rewrite and tailor these implementations, for changes in parameter and portability, as these platforms have different architectures and, often, require vendor specific software and languages to program and use them. A cross-platform standalone tool capable of mapping reads using different devices, simultaneously, for effective performance gains in a heterogeneous system is not available in the literature.

Embedded platforms, such as the single board computers (SBCs), have been designed keeping power and energy consumption as critical design parameter. They are, typically, powered using batteries and have a compact size. A plethora of SBCs are available off-the-shelf owing to the tremendous growth in Internet-of-Things (IoT) devices [31]. Using SBCs offer significant advantages such as low cost and maintenance requirements. They are a prominent candidate for locating computing and storage resources at end-user premises, therefore, aiding to the rising data privacy concerns following the developments in the field of genomics. In this thesis, we focus on the genome reassembly pipeline of the WGS, which is a prerequisite and a primary step in obtaining the genomic data. Reassembly is performed by mapping reads to the reference genome, which involves approximate string-matching algorithms and data structures that are commonly found in all computational pipelines of genomics. Genome is composed of four bases, viz. adenine, cytosine, thymine and guanine, represented as characters (`A C G T`) and requires integer-based operations for processing. For that purpose, simpler cores found on the embedded platforms may be better suited than complex general purpose CPU cores. This thesis opens a new research dimension of embedded genomics by attempting to prove the two hypotheses mentioned in Section 1.2. The solutions and results obtained can be extrapolated to other computational pipelines to mitigate the rising energy-consumption and performance concerns of this emerging Big Data contributor.

1.2 Hypotheses

The following hypotheses are the problem statements that this thesis attempts to address. The main objective is to reduce the energy consumption in mapping the entire human genome without compromising on performance. This will establish that the computational pipelines of WGS can be implemented on a low-power and memory restricted embedded platform to reduce energy consumption.

1. Using the OpenCL framework, a cross-platform read mapper targeting heterogeneous platforms can run parallel kernel executions on multiple devices, simultaneously, to enhance performance.

2. Using Algorithm-Hardware Co-design to map reads on an embedded platforms will reduce energy consumption.

1.3 Contributions

The main contributions of this thesis are as follows:

- **CORAL** - **C**ross-platf**O**rm **R**ead m**A**pper using openc**L**

  Using OpenCL programming framework, a cross-platform implementable read mapper is proposed. CORAL is capable of parallel kernel executions on multiple OpenCL conformant devices such as the CPUs and GPUs, making it first of its kind read mapper for heterogeneous platforms. Today, majority of platforms manufactured by different vendors comply with OpenCL standards [32]. This mitigates the need for restructuring or rewriting the tools to target any particular device and able to achieve high performance using all the available hardware resources, simultaneously.

  CORAL automatically determines the number of workitems (or threads) in a workgroup for a particular device based on user given workload allocation, distributes the workload to various devices, executes them in a task-parallel fashion and combines the output. We write the host code in `Python` and kernel in `C` using OpenCL primitives. We use PyOpenCL rather than conventional `C`-based

OpenCL, as an scripting language requires low programming effort. `Python` enables fast modifications and prototyping. This work attempts to prove hypothesis (1). CORAL is available online at: https://github.com/nclaes/coral

- **REPUTE** - **RE**ad ma**P**per for heterogeneo**U**s sys**TE**ms
  A novel dynamic programming (DP) based read mapping algorithm is presented for executions in memory-restricted embedded platform. Similar to CORAL, it uses PyOpenCL framework and is capable of parallel kernel executions on multiple devices, simultaneously. What distinguishes it from CORAL is its improved accuracy and performance due to novel DP based approach.

  REPUTE presents a first prototype implementation of read mapping on an embedded platform. It maps reads to chromosome (chr) 21 of the human genome. Central to this implementation is the Algorithm-HW co-design approach to design the REPUTE kernel targeting a memory restricted platform. It proposes a low-memory footprint kernel for simpler cores of SBCs and kernel-level optimizations for high performance. REPUTE demonstrates significant energy savings compared to general purpose workstations with comparable performance and same accuracy. This work attempts to prove hypothesis (1) and (2). REPUTE is available online at: https://github.com/nclaes/REPUTE

- **PLEDGER** - **P**yopencl based too**L** for g**E**nomic workloa**D**s tar**G**eting **E**mbedded platfo**R**ms
  As human genome is over 3000 Mb long, the size of data structures required for mapping reads to larger chromosomes such as chr 1 and 2 exceed the memory capacity available on embedded platforms. This prohibits mapping of the whole genome in memory-restricted environments. PLEDGER aims to optimize the read mapping algorithm for memory-restricted hardware platform to enable translational genomics. It proposes a novel preprocessing algorithm which generates memory-aware data structures. Using Algorithm-HW co-design, the PLEDGER kernel proposes a DP based read mapping approach to use the memory-aware data structures making it capable of mapping the entire genome on a single SBC with $>$3.6 GB RAM. It provides the flexibility of mapping reads to one, many

or all chromosomes of the human genome viz. chr 1-22, X and Y. Similar to CORAL and REPUTE, it uses PyOpenCL framework and is capable of parallel kernel executions on multiple devices, simultaneously. To improve performance, we tailor the algorithm for the target memory-restricted platform using bit-vector operations and localized variable optimizations to minimize the memory footprint of the kernel.

PLEDGER is a standalone tool which can complete the entire reassembly process starting from preprocessing to mapping and verification on an embedded platform for the entire human genome. It showcases significant energy savings and comparable performance with respect to general purpose workstations. This work attempts to prove hypothesis (2). PLEDGER is available online at: https://github.com/chitlangia/pledger

1.4 Thesis layout

This thesis is organised as follows:

Chapter 1 - Introduction. This chapter briefly discusses the importance of genomics, its continuous growth and advent to a major Big Data contributor. It presents the existing and future trends of usage of genomics and its role in on-going transformation of medicine from reactive to P4 form. It presents the bottleneck to translational genomics from the computational front with associated performance and energy requirements. This motivates us to present hardware based solution backed with Algorithm-HW co-design approach to mitigate cost, performance and energy requirements. In the end, it summarises the contributions.

Chapter 2 - Background. An overview of the whole genome sequencing is presented. The sequencing and reassembly pipelines of WGS are discussed in detail outlining the existing algorithms with their merits and demerits. Then, the read mapping approach and its three stages: Preprocessing, Filtration and Verifications are discussed in detail along with the approximate string-matching algorithms and associated data structures. Following, this we present a detailed report on the existing state-of-the-art read mappers with their methodologies and data structure. In the end, we present the data structures

and associated methods used in the contributions of the thesis.

Chapter 3 - Verification-aware Read Mapper for Heterogeneous Systems. In this chapter, a cross-platform OpenCL based read mapper, CORAL, for heterogenenous system is presented. The proposed preprocessing, filtration and verification stages of CORAL are discussed in detail with visualizations. CORAL is validated against state-of-the-art read mappers using performance and accuracy metrics, when executed on CPU and GPU using real human reads are presented.

Chapter 4 - Dynamic Programming based Filtration. This chapter proposes a novel dynamic programming based filtration methodology which significantly enhances the performance and accuracy of the read mapper. The filtration scheme is realised in a read mapper, called REPUTE, which implements all the three stages of reassembly in a LMF kernel using Algorithm-HW Co-design along with series of kernel-level optimizations. A prototype implementation of read mapping on an embedded platform is presented. The results indicate considerable improvements in performance compared to state-of-the-art mappers and significant energy savings on embedded platforms compared to general purpose workstations.

Chapter 5 - Embedded Whole Genome Read Mapping. This chapter presents a tool for whole genome read mapping on a memory-restricted embedded platform called PLEDGER. PLEDGER proposes a memory-aware data structures suitable for low-memory environments and uses Algorithm-HW Co-design to develop a performance optimised LMF kernel. It showcases considerable performance gains compared to state-of-the-art read mappers with similar accuracy. It demonstrates that embedded genomics has the potential to mitigate the growing energy demand due to growth in genomic data.

Chapter 6 - Conclusions. The contributions of the study discussed in this thesis are summarised, and future research areas for the embedded genomic pipelines for high-performance, affordable and energy efficient genomic computation is suggested.

Chapter 2

Background

This chapter presents an overview the Whole Genome Sequencing (WGS) process and discusses the background and state-of-the-art tools available for genome re-assembly. Section 2.1 begins with the description of genome and its molecular structure, followed with an overview of the WGS pipelines. Then, the working principles of second generation of sequencing technology, the next-generation sequencing or high-throughput sequencing, is described. The section ends with an introduction to the recent third generation sequencing technology.

In Section 2.2, the fundamental genome re-assembly approaches are introduced viz. *de novo* and the read-alignment approach. The complexities in the structure of genome which affect the re-assembly process are discussed along with the types and usage of reads available from different sequencing technologies.

In Section 2.3, an overview of the *de novo* assembly approach is presented. *De novo* assembly relies on graph-based data methods such as overlap-layout-consensus, *de Bruijn* and string graphs. Notable assemblers using each of the graph based approaches are are presented. FInally, the shortcomings of *de novo* approach is discussed.

In Section 2.4, an overview of the read alignment approach is presented. A detailed discussion on the algorithms and data structures ensues, which includes the different stages involved in the read mapping process. A review of the state-of-the-art read mappers is presented along with their algorithmic approaches. Finally, Section 2.6

summarises the chosen approaches used in the contributions of this thesis.

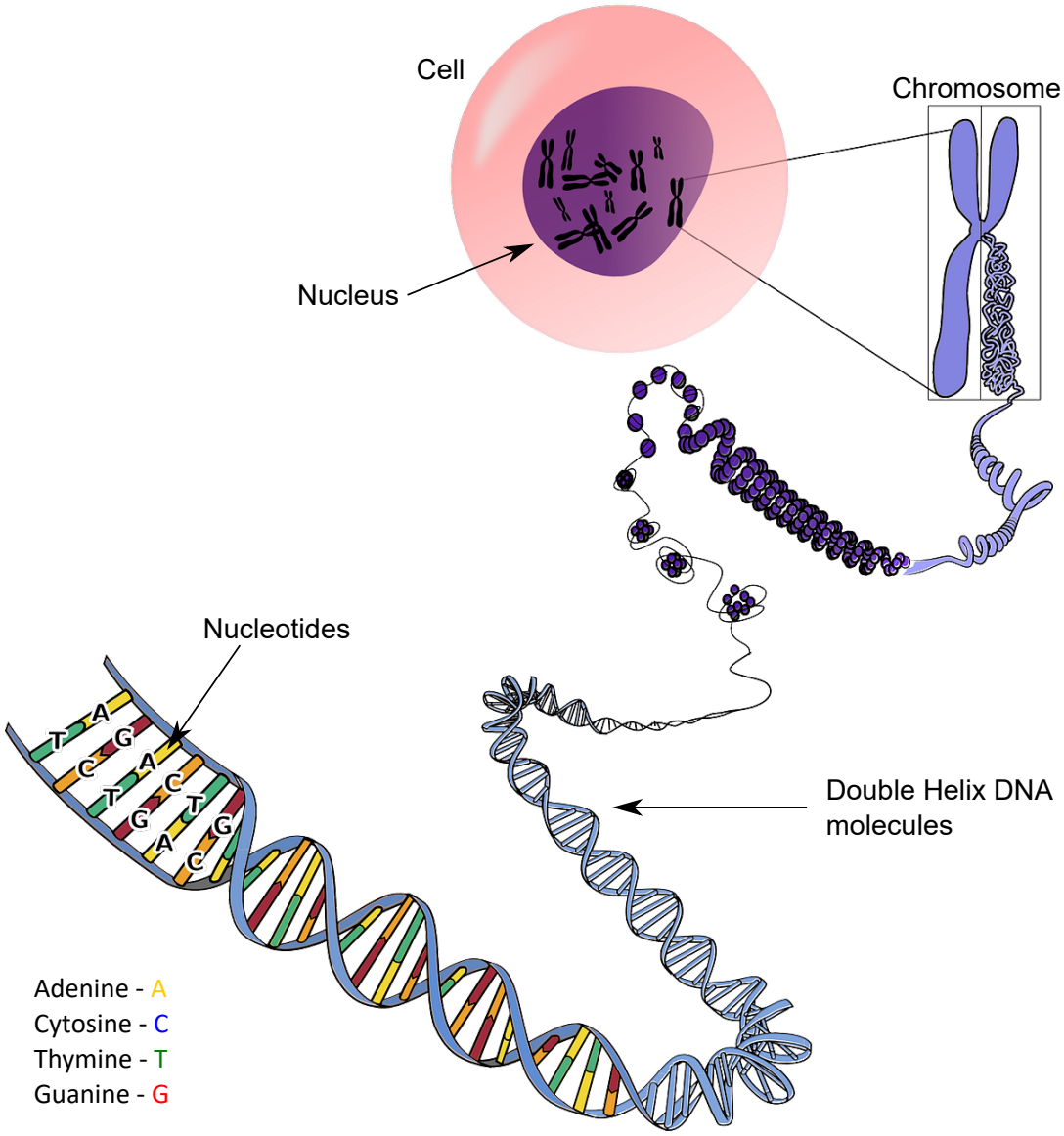

Figure 2.1: Microscopic visualisation of genome with chromosomes expanded to level of nucleotide bases which constitute the entire genome. Original image source: OpenClipart-Vectors, via pixabay.com

Figure 2.2: A visualisation of the DNA's double helix structure in the form of forward and reverse strand. Forward strand is complement of the reverse strand where the base `A` is complement to `T` $A \leftrightarrow T$ and base `C` is complement to `G` $C \leftrightarrow G$. A genome strand is read from 5′ to 3′ direction, which means left to right for forward and right to left for reverse strand.

2.1 Whole Genome Sequencing

2.1.1 Genome

Genome is the complete genetic material required to build and maintain an organism including all the chromosomes and genes. Fig 2.1 presents a visualisation of genome and its composition. Genome is found in the nucleus of a cell as a collection of chromosomes. Chromosomes are very long strand of millions of DNA molecules in a compactly coiled up structure with the help of proteins called histones, that support its structure. The DNA molecules are composed of nucleotide bases viz. Adenine, Cytosine, Thymine and Guanine, joined together in a double helix structure. These nucleotide bases can be represented using alphabets $\sum = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$, respectively, and, hence, genome can be stored in the form of a text with over 3.2 billion characters. Fig 2.2 visualises a representation of the double helix structure of the DNA molecule as two strands. By convention, for a reference chromosome, one whole strand is designated the "forward strand" and the other the "reverse strand". Visually, the sequence of a strand is, typically, read in the 5′-3′ direction i.e. for the forward strand, it will be read from left-to-right, and for the reverse strand it means right-to-left. The relation between the two strands is that both are complement of each other, as in, the base `A` is complement to `T` ($A \leftrightarrow T$) and base `C` is complement to `G` ($C \leftrightarrow G$). Thus, provided with one of the strands, the other can be generated, with some rare exceptions when either of the strands will have mutations not present in the other in the case of diploid or polyploid genomes due to heterozygous single-nucleotide polymorphism (SNP) and indels [33–36].

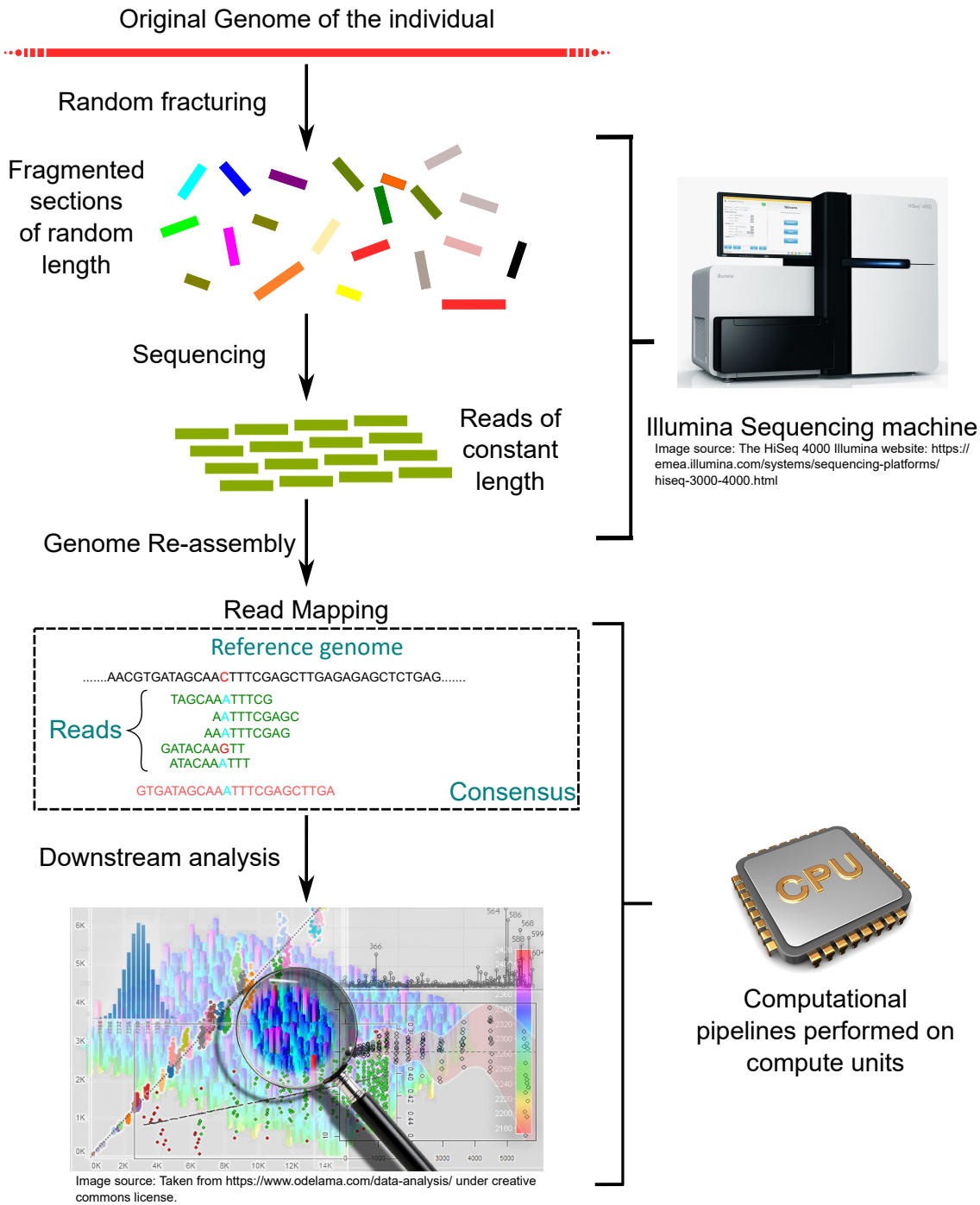

Figure 2.3: A visualisation of the whole genome sequencing process depicting sequencing, re-assembly and analysis pipelines.

2.1.2 Overview of WGS

Ever since the advent of next generation sequencing, there have been continuous advances and inventions of new sequencing technologies. Still the modern instruments are capable of reading only small segments of the genomes of most organisms, ranging from approximately 100 base pairs (bp) to several tens of kilobases [37]. By contrast, the human genome comprises more than 3 Gb long, and even small organisms such as bacteria have genomes spanning over millions of bases. Reconstructing an entire organism's genome sequence, thus, requires gluing together many small sequence fragments. WGS is a comprehensive method to obtain and analyse the entire genome of an organism. Broadly, it consists of three pipeline stages: sequencing, re-assembly and downstream analysis. Recent jargon refers re-assembly and downstream analysis together as Genome Analysis. The discussion on downstream analysis pipeline is not within the scope of this work and is part of the future work which is presented in Section 6.2. The sequencing stage aims to obtain and identify the structure of an organism's or individual's genome and is performed in a variety sequencing machines from different vendors including Illumina, Qiagen, PacBio, 10x Genomics and Oxford Nanopore Technologies [37, 38]. Although, the techniques and processes employed by each sequencing machine manufacturer differ, the underlying approach for second generation sequencing (SGS), typically, involves template preparation, cyclic reversible termination (CRT) and imaging [39]. High-Throughput Sequencing (HTS) refers to 2^{nd} generation and above in this thesis.

Template preparation involves taking the original genome from the sample of an individual or organism and randomly breaking them into smaller fragments, as visualised in Fig. 2.3. These single strand small fragments are then amplified by producing millions of copies to form an immobilised cluster in a small region on the top of a surface, as shown in Fig. 2.4. Once the template formation is complete, CRT step is initiated where polymerase chain reactions (PCR) takes place with special terminal primers. PCR reaction replicates the organism's genome during cell division, where the DNA molecule is split into single strands and an identical replica of the original double stranded DNA molecule is generated using PCR, as shown in Fig. 2.5. PCR is a matured

technology and, therefore, we leave it to the interested readers to look of more sources online [40, 41]. Here, in each CRT step, the PCR aims to attach one complementary nucleotide base, per step, in the opposite strand in all the clusters. This nucleotide base is a fluorescent labeled terminal primer which prevents PCR from adding more than one base and at the same time reflects a particular color depending on the nucleotide base in $\sum$. In the next step, imaging is performed to identify the base attached in the CRT step by capturing the reflected lights from each cluster, as shown in Fig. 2.4. Cluster formation is necessary so that the reflected light has sufficient intensity for accurate imaging by the camera. The terminal part of the primer is then removed and the CRT and imaging steps are repeated till the entire genomic fragment is sequenced.

At the end of sequencing run, the reads are trimmed to fixed length and stored with quality scores depending on the intensity of light captured during the imaging step. The output reports the quality for each base in a read indicating the quality of base call during the sequencing process. A higher score indicates that the base was identified accurately. The sequencing process is error-prone as during the CRT step some primers may not link successfully or more than one primer gets attached. This dephasing leads to miscalls at the end of reads due variations in intensity during the imaging step. Thus, the reads obtained upon sequencing are, often, not identical to the original random fragmented sections and may include substitutions, deletions and insertions [36,39]. These errors are called sequencing errors as they are caused due to the sequencing technology.

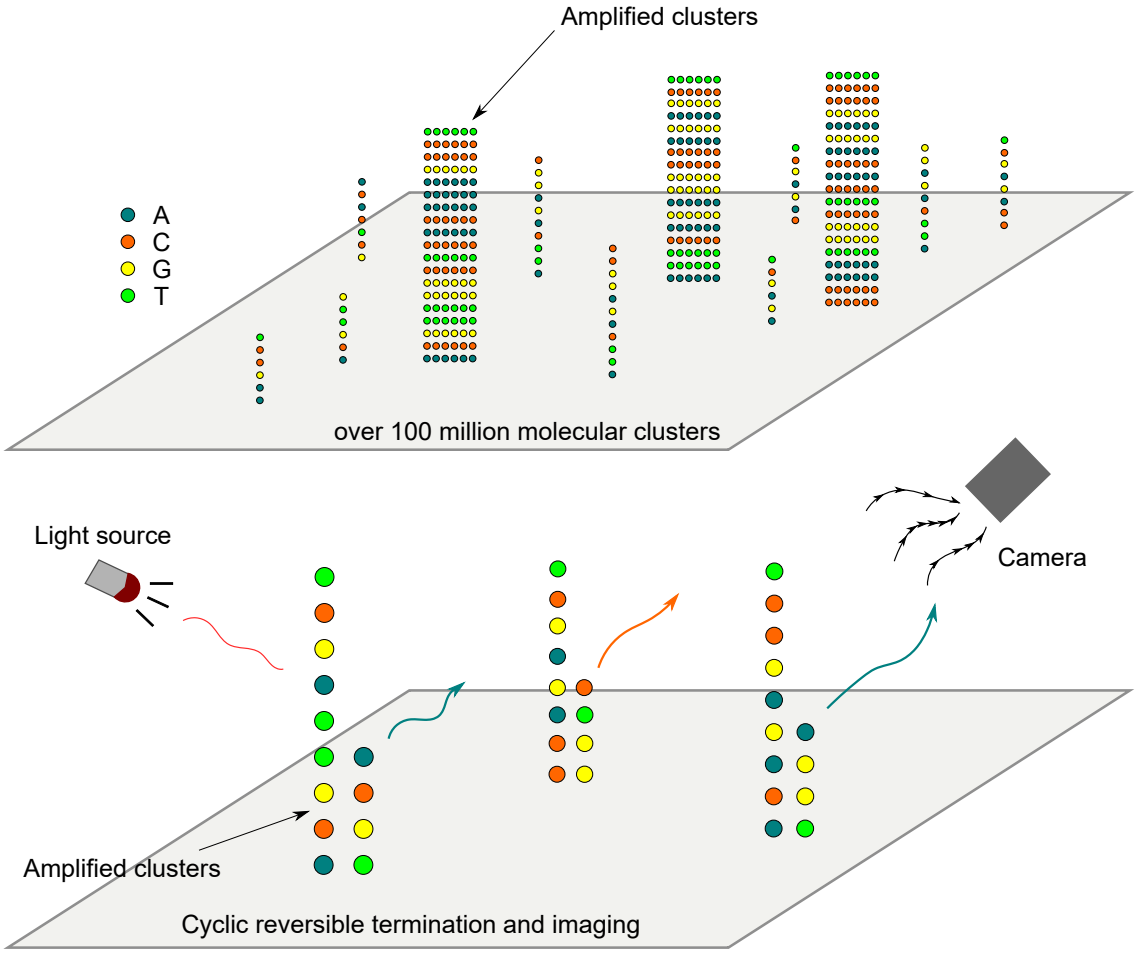

Figure 2.4: A demonstration of the sequencing process with visualisations of amplified template, cyclic reversible termination using PCR and imagine to capture reads. PCR is performed using a special terminal primer which prevents it from adding more than one fluorescent labelled nucleotide base to the opposite strand, per cycle, and reflects a particular light depending on the base added. By capturing the reflected light from the amplified clusters the structure of the reads are identified.

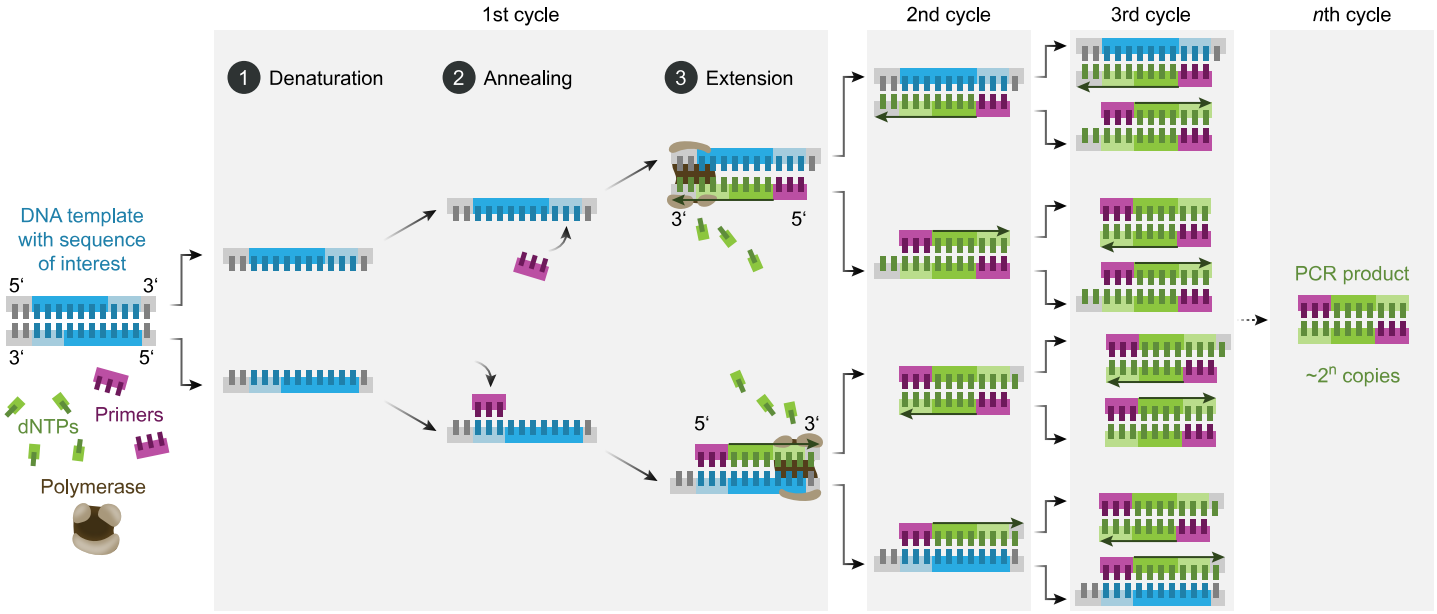

Figure 2.5: An illustration of the polymerase chain reaction (PCR) while generating copies of the DNA molecule. Denaturation splits the double stranded DNA molecule into two single strands and then PCR completes the opposite strand generating identical copies of the original DNA molecule, using nucleotide bases (dNTPs), primer and polymerase enzyme [40]. With each cycle the total number of replicas double. Image source: Enzoklop, CC BY-SA 4.0, via Wikimedia Commons.

There are various different types of template preparation techniques offered by different sequencing machine vendors. SGS technologies are template-based shotgun sequencing technology. SGS involves oversampling of original genome by generating a very large number of small-length reads by amplification of the genome during sequencing process, hence, categorised as HTS. As SGS focuses on generating large numbers of short reads than longer reads, it has significantly reduced the cost of sequencing at the expense of processing massive amounts of data at the later stages. They are characterised by highly parallelised operations, high throughput, higher yield, simpler operation leading to lower cost per read and high accuracy but, unfortunately, shorter reads. Recently, a third generation of sequencing technology based on single-molecule sequencing (SMS) has emerged. SMS technology allows less bias and more homogeneous genome coverage as it lacks PCR amplification. It is real-time sequencing as opposed to SGS which is paused after each CRT step. The most important feature of SMS technology is that it offers longer reads which can help close gaps in current reference assemblies and resolve ambiguous regions in the genome. SMS technologies, however, lack in yield, generate high error rate and have high cost per base which currently impeded its usage in large scale sequencing project [37, 42–45]. A detailed discussion on the latest trends in the sequencing technology is beyond the scope of this thesis, interested readers can refer to [36, 37, 39, 42–46].

2.2 Genome Re-assembly

The purpose of genome re-assembly is to rearrange the raw reads in a manner indicative of the original genome they come from. This can be performed using either of the following approaches: read alignment or the *de novo* assembly. The genome of a novel species, with no prior knowledge of the source DNA sequence, is assembled *de novo*. If an organism's genome is already assembled once before and the reference genome is available then read alignment approach results in faster re-assembly. The decision of using either of the approaches is based on the intended biological application, types of reads, cost, effort and time considerations. The details of the re-assembly approaches will be discussed in the following sections, this section outlines challenges to genome

re-assembly.

Genomes tend to have complex structures which, often, require tremendous time and research to understand. This becomes more challenging with genomes of complex organisms such as mammals, where the genomes are very long and complex. Most genomes contain a certain proportion of repetitive DNA elements, called repeats, throughout its length [45,47], particularly in mammalian genome where repeats account for 25%–50% of its entire genome . Repeats constitute about 50% of the human genome and more than 80% of the maize genome. They are categorised as long interspersed nuclear elements (LINEs), short interspersed nuclear elements (SINEs), long terminal repeats (LTRs) and simple tandem repeats (STRs) [48]. Earlier repeats were referred to as "junk," however, recent studies have shown that they are involved in intrinsic biological processes, including genome expansion, speciation, evolutionary adaptation, generation of genetic variation, and epigenetic regulation [49–51].

The importance of repeats require that these regions are not collapsed or ignored during re-assembly. They, however, make genome re-assembly extremely difficult, often, causing misarrangements or gaps. Repeats yield fragments with highly similar sequences that originate from different places in the genome during sequencing causing nonuniform read depth and resulting in copy loss or gain in the assembly [45, 52, 53]. The extent to which assemblers are confused by repeats depend on the types, length and accuracy of the reads obtained from the sequencing process. Fig. 2.6 presents a comparative visualisation of how three different types of reads viz. single-ended short, pair-ended and long reads offer certain advantages while re-assembling genome, especially, the repeating sections. The repetitive regions shorter than a sequencing read can be automatically resolved. Complications arise when repeats are longer than read lengths, then, to identify the repeat from where a read has originated, read should overlap adjacent non-repeating section. Paired-end reads are two reads sequenced from opposite ends of the same fragment which may be several kilo bases long. Pair-ended reads can provide some respite as they cover a larger portion with increased chances of one end of the read overlapping the non-repeating section as shown in Fig. 2.6(b). Longer reads are preferred to resolve large repeating regions, however, they are accompanied with higher error rates. Higher error rates lead to multiple mapping

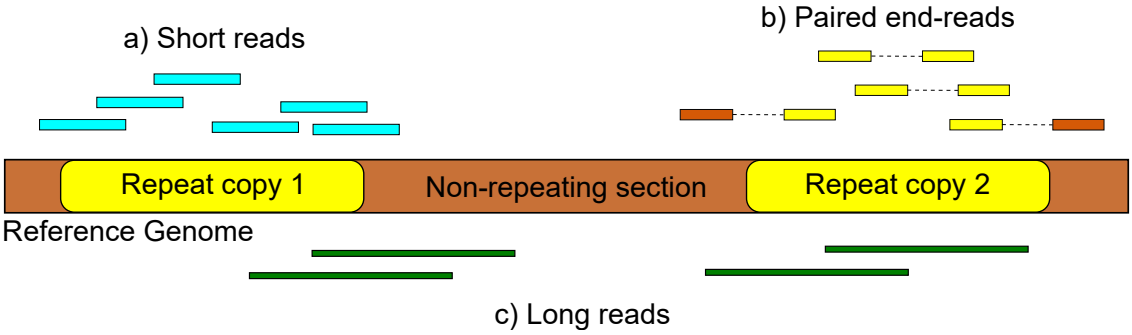

Figure 2.6: A comparative visualization of the types of reads that are often generated from the sequencing processes. Typically, there are three types are reads: (a) short, (b) paired-end reads and (c) long. Repetitive sections are prevalent in genomes, with approximately 50% makeup of human and over 80% makeup of the maize genome being repeats. Repeats posit a major challenge in genome re-assembly and the type of reads being mapped can affect the mapping accuracy.

locations and decreases the chance of convergence to a unique solution [53]. Repeats and the associated complexity it brings in the assembly process had been a matter of contention and vigorous debate over the feasibility of assembling the entire human genome from shotgun sequence data [54, 55]. Even after two decades of the arrival of first human reference genome, the remaining gaps and complexities are continuously being resolved with advancements in the sequencing technology. Quite often more than one type of reads are used to assemble and refine the genome.

Ideally, longer accurate reads are desired for high quality assembly, however, this gap is, yet, to be filled by any single sequencing technology. Before the advent of SMS technology pair-ended reads were used to resolve the gaps, present due to repeats, upon assembling small reads. Recently, the human genome was assembled using long reads obtained from nanopore sequencing technology [56], which is one of the technology under the SMS category. However, due to high error rate of nanopore technology, single-ended small but accurate reads from Illumina were used to resolve the gaps and improve basecalling [44]. Illumina is currently the leader in the SGS industry and most library preparation protocols are compatible with the Illumina system. In addition, it offers the highest throughput of all platforms and the lowest per-base cost [45]. In this thesis, genome re-assembly has been performed using short single-ended accurate reads of length 100-150. The methodology proposed in this thesis is translational and can be

extended to pair-ended and longer reads.

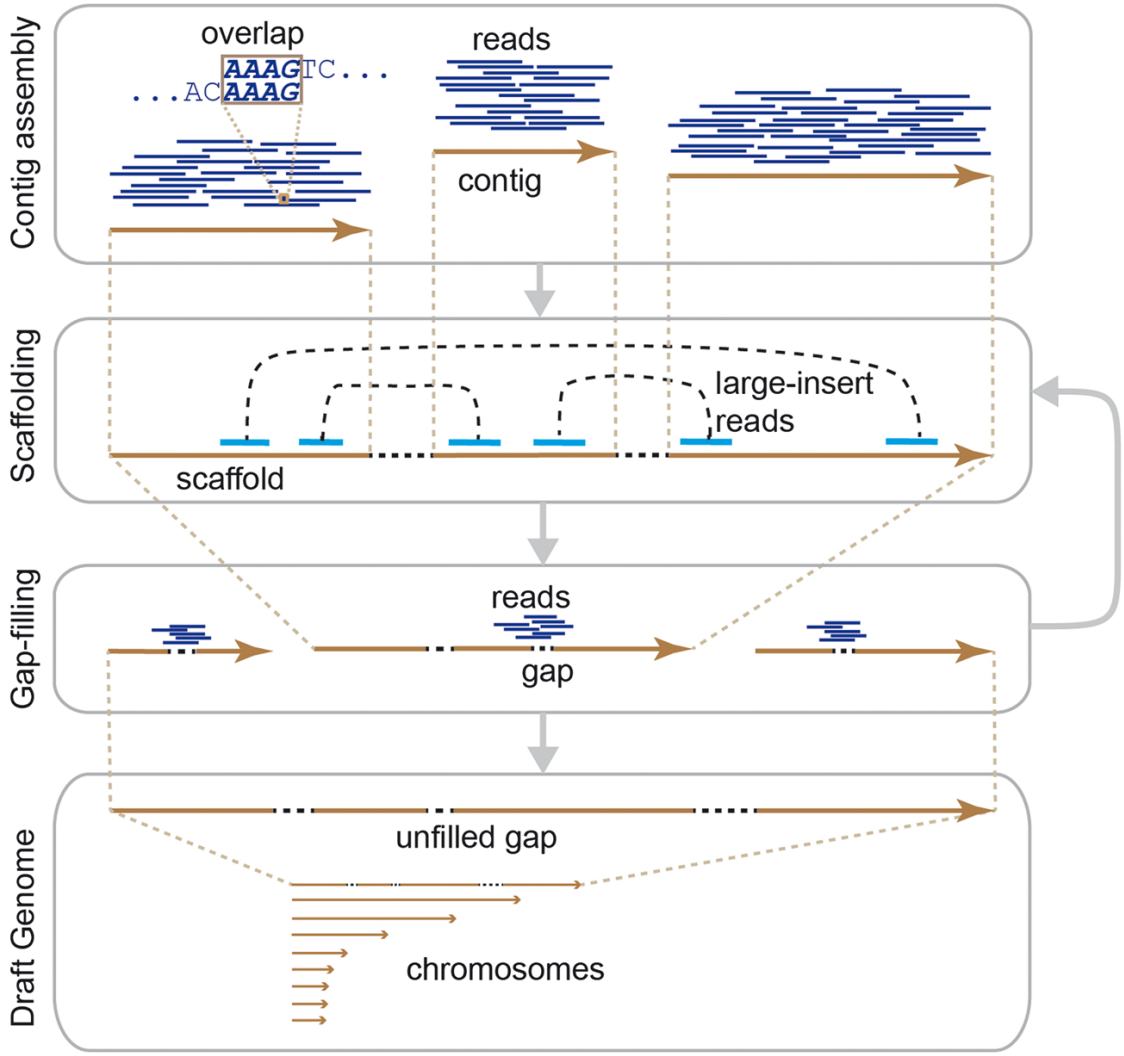

Figure 2.7: An overview of the workflow of *de novo* assembly approach. There are three main stages: (i) contig assembly, (ii) scaffolding and (iii) gap filling. Contigs are contiguous genomic segments, free of any gaps, obtained using overlapping segments in reads. Scaffolding are ordered and oriented contigs arranged using paired-end reads as anchors. Gap-filling involves resolving unidentified bases using consensus based on independent reads. These gaps may be important SNPs and variations. [52]

2.3 *De novo* Assembly

De novo assembly approach is similar to solving a giant jigsaw puzzle with a massive number of pieces. However, the pieces, here, are reads with errors, making it even more challenging as exact matching algorithms are not applicable. As shown in Fig. 2.7, there are, mainly, three stages in the *de novo* assembly approach: (i) contig assembly, (ii) scaffolding and (iii) gap filling [1, 45, 52, 57, 58]. Contigs are continuous (or contiguous) genomic segments, free of any gaps, obtained using a set of overlapping read segments that together represent a consensus region of genome. Contigs are formed using graph-based assembly algorithms based on three basic graph frameworks viz. overlap-layout-consensus (OLC) graph [59], *de Bruijn* graph [60] and string graph [61]. These graph frameworks will be discussed later in subsections 2.3.1, 2.3.2 and 2.3.3. Scaffolding are sets of ordered and oriented contigs with the approximate distances between contigs estimated by traversing paired-end sequences that anchor to different contigs. The sequences that are linked are typically contiguous sequences corresponding to read overlaps, as shown in Fig. 2.7. Gap-filling involves resolving unidentified bases using consensus based on independent reads. These gaps may be important SNPs and variations. To enhance the quality of the assembly, the scaffolding and gap-filling steps are, often, performed iteratively to close the gaps by re-processing latent information in the raw reads, until no scaffolded contigs remain or no additional gaps can be resolved [62, 63]. .

2.3.1 Overlap-Layout-Concensus (OLC)

OLC is the simplest and the earliest used graph-based model, which presents each read as a node and connects the nodes if they overlap, as shown in Fig. 2.8(a). Alternative formulations have a pair of vertices for each read with one representing the start, other its end and the edge representing the read's sequence. Construction of OLC, typically, involves three main stages. First, overlaps between all reads are detected. Second, the graph is constructed, then contigs are formed by iteratively merging overlapping reads until a read at a repeat boundary is detected leading to a repeat that is unresolved and collapsed into a single copy, as shown with the split colour bar in Fig. 2.8(a). Finally,

a consensus sequence is inferred. To account for sequencing errors, imprecise read overlaps are allowed [45,52,57].

The most computationally intensive stage in OLC is to find overlaps among reads. A naive approach would use dynamic programming based alignment between all possible combinations of reads. However, this would require $\mathcal{O}(N^2)$ time, where N is the number of sequenced bases, and, hence, are used for very small genomes. Faster approaches include building data structures such as indexes constructed using *k-mers*. *K-mers* are non-overlapping or overlapping sections of the read or genome of length k, as shown in Fig. 2.9. While *q-grams* of length q are consecutive overlapping *k-mers* with relatively small values of q. A read with the length of l can be divided into $(l-k+1)$ overlapping *k-mers*. The index build by extracting *k-mers* can be used to identify reads having common *k-mers* and then using dynamic programming the overlapping can be verified. This technique drastically reduces the search space.

With the computational bottleneck of expensive dynamic programming steps, OLC could not succeed with massive amounts of short-reads from HTS. Even with indexing the number of spurious matches overwhelmed the computation. The size of the resulting graph proved problematic and the number of edges grew quadratically with the depth of coverage and number of repeats making the resulting graph impractical to solve [57]. The first human genome was constructed primarily using OLC algorithms, and notable OLC-based assembly methods include parallel contig assembly program (PCAP) [64], Arachne [65] and Celera [66]. In contrast to short-reads, an overlap graph for the low-throughput and high-error rate long reads from SMS technology, generally, forms a much less complex graph. Some notable works using OLC-based assembler for long reads can be found in [67,68].

2.3.2 *De Bruijn Graph*

For high-throughput short-read sequence data, *de Bruijn* graph approach is used more often. In this approach, the reads are broken into *q-grams* (or overlapping *k-mers*) and distinct *q-grams* are added as vertices to the graph while the *q-grams* from the adjacent positions in a read are linked by an edge, as there is an overlap with $q-1$ bases, as depicted in Fig. 2.8(b). The assembly problem can then be formulated as finding a

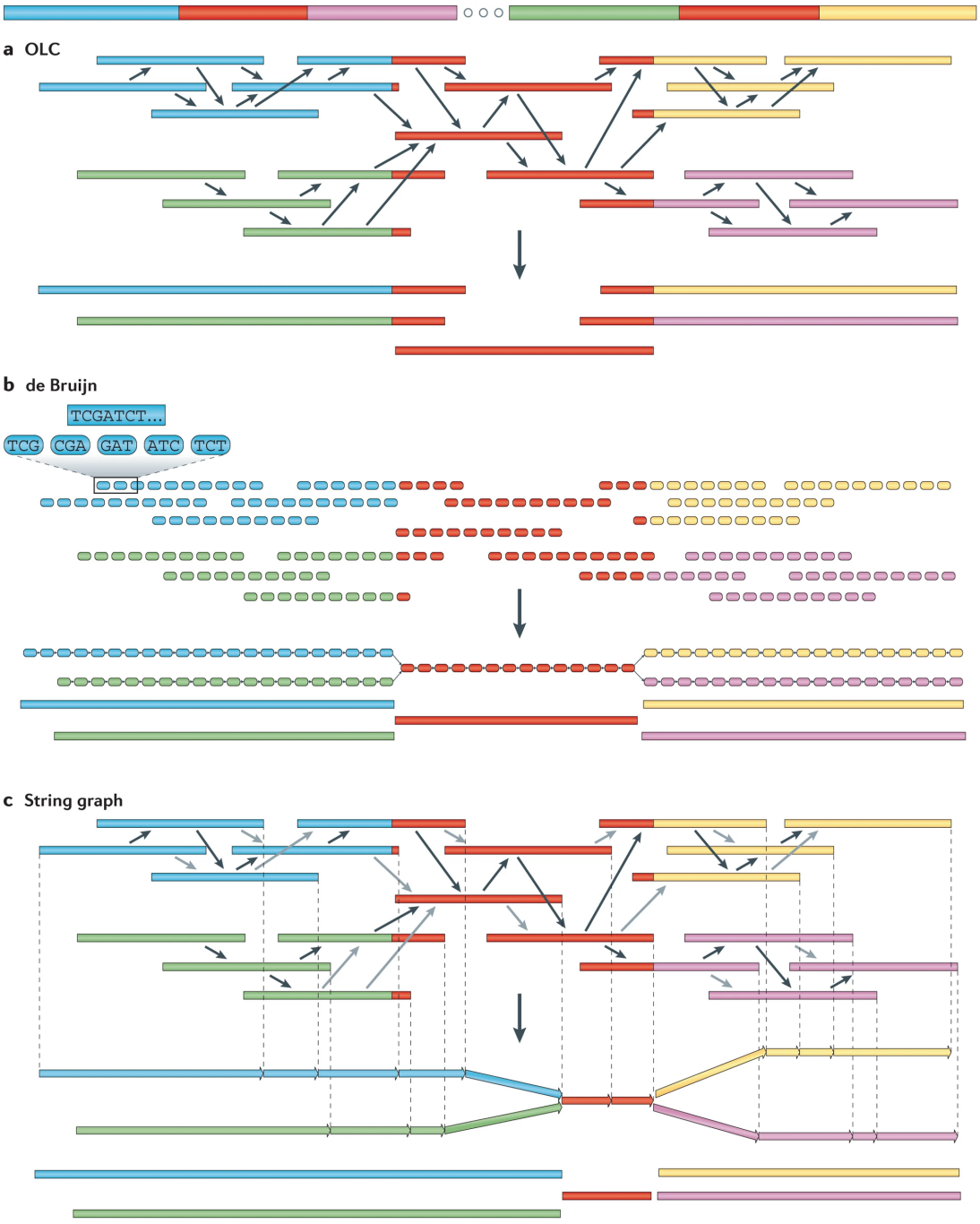

Figure 2.8: A visualisation of the working-principle of *de novo* assembly approaches with the help of four unique regions (blue, violet, green and yellow) and two copies of repeated region (red). (a) Overlap-Layout-Consensous (OLC), (b) *de Bruijn* graph approach, and (c) String graphs [1].

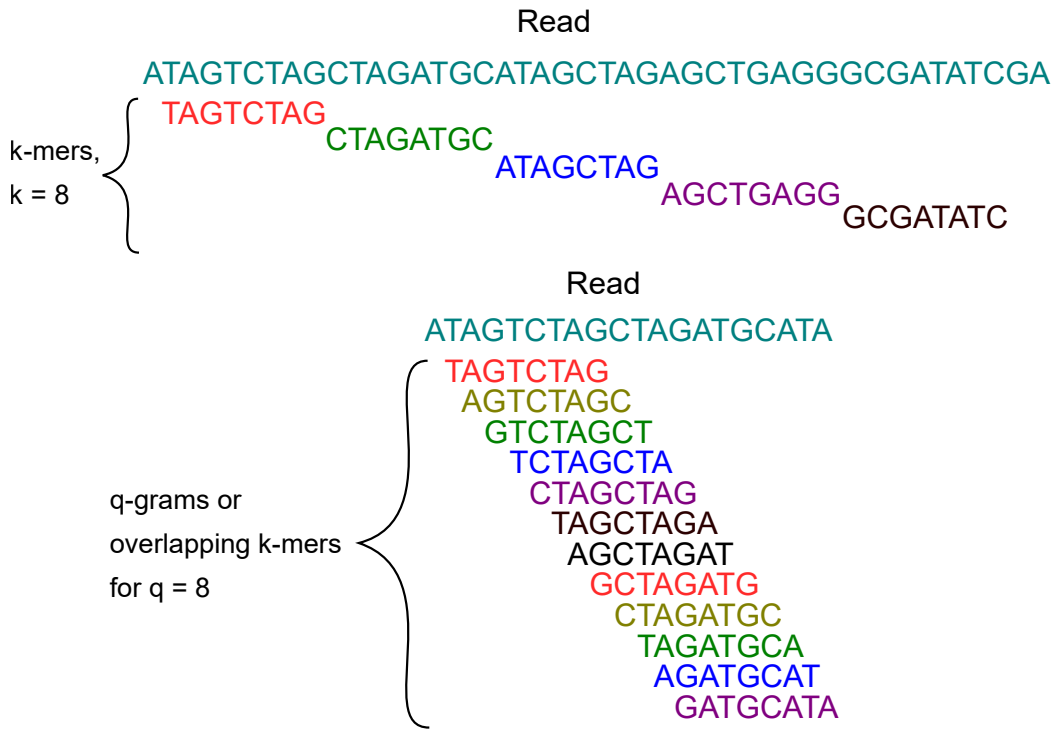

Figure 2.9: A visualisation of k-mers and q-grams. k-mers are non-overlapping genomic sections of length k while q-grams are consecutive overlapping sections of length q. q-grams are, often, referred as overlapping k-mers.

walk through the graph that can be either a Hamiltonial or an Eulerian path [45]. In Hamiltonian approach, the *q-grams* are nodes while in an Eulerian approach they are the edges. Hamiltonian approach is similar to OLC approach where the genome is re-assembled by traversing the Hamiltonian paths through the graph by visiting all nodes, only, once. The Hamiltonian path problem is an NP-complete problem when the number of nodes is not trivial with the computational complexity of $\mathcal{O}(m \times 2^n)$, where m is the total number of nodes, and n is the number of branching nodes [45]. Notable *de novo* assemblers using Hamiltonian approach are SOAPdenovo [69], Abyss [70] and velvet [71].

The Eulerian path problem assembles the genome by finding Eulerian paths that traverse all edges, each of which is visited only once without simplification in polynomial time $\mathcal{O}(n^2)$ [45]. Practically, the Eulerian and Hamiltonian traversal through the graphs are obscured due to sequencing errors. As reported in [57], even if an Eulerian path

through the graph is found, it may not reflect an accurate sequence of the genome because of the presence of repeats, as there are a potentially exponential number of Eulerian traversals of the graph, only one of which is correct. Generally, assemblers attampt to contruct unambigous contigs without any gaps and then use scaffolding steps to connect the unbranching regions of the graph, as shown in Fig. 2.8(b). Notable *de novo* assemblers using Eulerian *de Brujin* graph approach are ALLPATHS [72], SPAdes [73], IDBA-UD [74] and EPGA2 [75].

The Eulerian de Bruijn graph based assemblers generally perform better in the assembly of a large genome than the Hamiltonian graph based assemblers. As it does not require to find overlaps between read pairs, absence of expensive dynamic programming based alignment steps significantly reduce the computational burden compared to OLC. The graph can be constructed by two passes over the data, in which, the first one extracts the *q-grams* and second, extract the adjacent overlapping *q-grams* to form the edges, in case of the Eulerain path approach. Additionally, *de Bruijn* do not require to store the pairwise overlaps and have a useful property of collapsing the repeats where all copies of repeats are represented as a single copy with multiple entry and exit points. This provides a concise representation of the structure of genome [57], as shown in Fig. 2.8(b). Using suitable choices of data structures, various algorithms have been proposed to efficiently re-assemble genome using *de Bruijn* graphs [45, 57].

2.3.3 String Graph

The string graph was introduced by Myers in 2005 [61]. He observed that the advantages of a *de Bruijn* graph can be obtained from OLC graph by performing two transforms. First, by removing the duplicate reads that may contain distinct elements of same or its reverse-complement sequence and contained reads which may be reads that are a substring of some other reads or their reverse complements, and second, by removing the transitive edges from the graph. A graph is created with a vertex for the endpoint of every read. Edges are created both for each unaligned interval of a read and for each remaining pairwise overlap. Vertices connect edges that correspond to the reads that overlap, as shown in Fig. 2.8(c) [57]. The string graph shares several properties of *de Bruijn* graphs without the need of breaking reads into *q-grams* [57]. They are, often, used

with long reads with high error rates generated from SMS technology as overlap-based approaches are more suitable than the *de Bruijn* graph-based methods [45]. String graphs, also, use advanced data structures such as FM-Index [76], which will be discussed in Section 2.4.1. Some notable string graph based genome assemblers are [77–79].

2.3.4 Shortcomings of *de novo* Assembly Approach

The memory requirements of *de novo* assembly, in general, and *de Bruijn* assembly, in particular, are prohibitive. As there is approximately one *q-gram* for every base in a genome, the *de Bruijn* graph has billions of vertices for large genomes such as mammals [57]. Sequencing errors compound this problem as they lead to several types of structures making convergence to true genomic sequence difficult. Each error in read produces up to q erroneous *q-grams*. Sequencing errors along with small natural variations in repeats, typically, leads to bulges, tips and erroneous connections in the graph which further complicates resolving the graph. The requirements of memory (or RAM) and storage, and long computation times are the major bottleneck of *de novo* approach. Assembling human genome using *de novo* approach can take several days to weeks using servers or clusters with up to 512 GB RAM. The details of various tools along with their reported computation times and resource consumption are summarised in [45].

2.4 Read-Alignment Approach

If any species for which a high-quality assembled genome sequence already exists, such as humans and most model species, then there exists a computationally faster and more efficient method compared to *de novo* assembly approach, called the read-alignment approach. This approach aims to re-assemble a genome by comparing reads to the reference genome of the organism to identify the correct position from which they originated during the sequencing process. This is possible because genomes of organisms from the same species are extremely similar, e.g., two unrelated humans have genomes that are approximately 99.8% similar [36]. Therefore, reference genomes can be used as a template or guide to assist in piecing reads together. The increased reliability of the reference genome, additionally, provides a universal standard against

Figure 2.10: A visualization of read mapping using the read-alignment approach.

which research findings can be reported, globally [80]. Fig. 2.10 visualises mapping of reads to the human reference genome (HRG) where reads consisting of sequencing errors and natural variations is mapped against the HRG using approximate string matching algorithm. Read 1 in Fig. 2.10 matches with the HRG except for one base demonstrating a substitution, while read 2 demonstrates two substitutions. The variations may range from single point mutations to short insertions or deletions (indels) to even larger-scale complex variations spanning thousands of nucleotides or more, although, whether the variations can be resolved or not depends on the length of the reads as discussed in section 2.2. Fig. 2.11 shows the three main stages in the read alignment approach: proprocessing, filtering and verification. Reference genome is used in the first and the last stage of the pipeline.

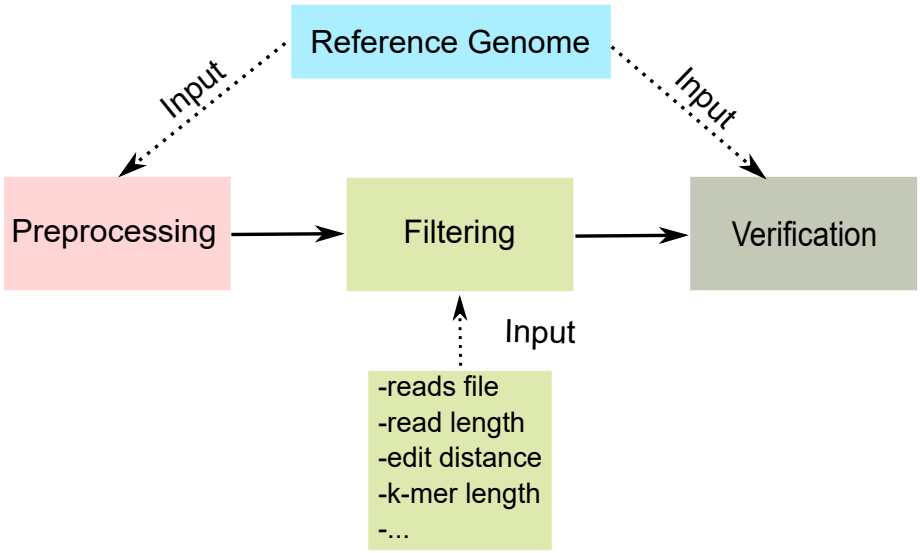

Figure 2.11: Read mapping stages in the read-alignment approach using approximate string matching algorithms

2.4.1 Preprocessing

Indexing strategies involve the storage of reference genome or read sequences or both in an intricate way, often, in the form of auxiliary data structures [81]. Preprocessing reference genome instead of sequenced reads is advantageous as computation of auxiliary data structures for reference will be required once while indexing of sequence reads will be conducted for each sample, separately [82]. Since the advent of HTS, the massive amounts of reads generated makes indexing them impractical, hence, mostly the reference genomes are indexed. A naive algorithm would use dynamic programming to align a read of length n to a HRG of length m, however, this becomes impractical for large sequences as its time complexity ($\mathcal{O}(m \times n)$) is proportional to the lengths of sequences and will grow quadratically. Thus, instead of scanning the entire HRG for each read, approximate string matching algorithms seeks the candidate locations in HRG, where a read is most likely to match. To facilitate such an approach, quite often, HRG is preprocessed and stored in the form of data structures so that the candidate locations for reads are obtained quickly. Most modern read mappers preprocess using an offline approach which is performed once and then reused for all input reads, as the HRG does

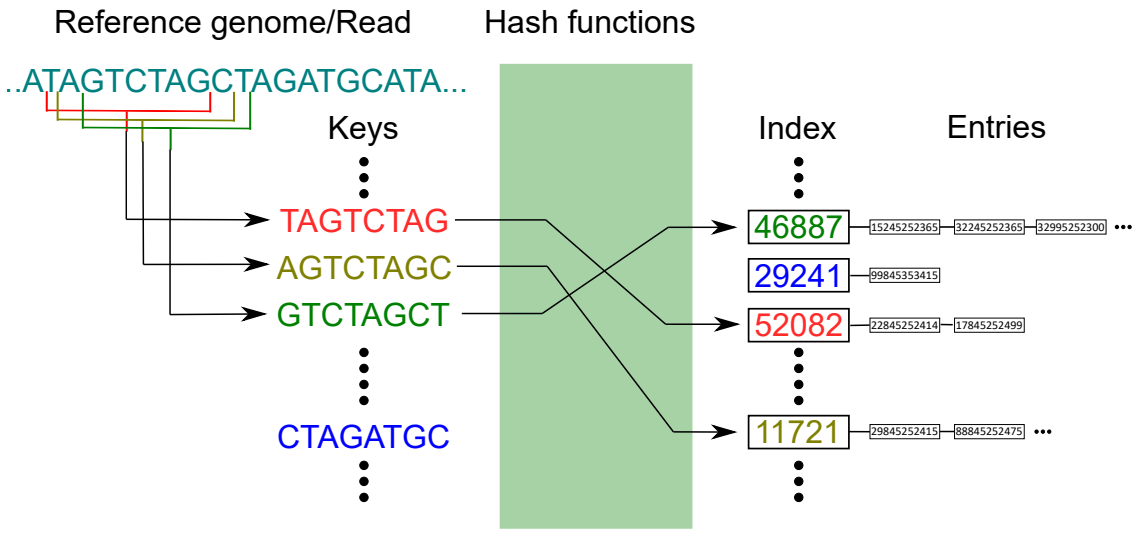

Figure 2.12: Visualisation of the hashing process by converting *q-grams*, derived from the reference genome, to numerical indexes using hash functions. All positions of occurrence of a *q-gram* in the reference genome is appended in the entries linked to the corresponding index.

not change unless updated. The preprocessing strategy used by state-of-the-art read mappers can be broadly grouped in two categories: algorithms employing hash tables and algorithms using suffix/prefix tries with its faster and efficient variants [81–83].

Using hash tables to preprocess HRG

In the context of read mapping, hashing is, also, known as *q-gram* indexing [2, 81]. The general idea around hashing involves the extraction of overlapping *k-mers* or *q-grams* from the reference genome, referred to as keys, which are encoded into a numerical index to point to a memory address. The memory address holds the position of occurrence of the *k-mer* in the HRG. Figure 2.12 demonstrates the hashing of overlapping *k-mers* extracted from the reference, which are encoded to generate indexes pointing to memory locations that contain the entries holding the positions of occurrences of the *k-mers*. Hashing is a matured technology and to encode a character sequence (*k-mer*) into a numerical, many hashing functions are available. However, there are three important factors which determine the choice of hash functions: collision, performance and memory requirements of index or hash table [83]. Due to massive length of the HRG with significant repeat sections, depending on the length k there will be millions of *k-mers*

that will be encountered more than once. This will lead to collisions as similar *k-mers* will generate same index. Secondly, depending on the hash function being used to encode the *k-mers*, the length hash tables can significantly impact the performance and memory requirements to store the HRG.

The most commonly used hash function to generate *k-mer* index uses 2-bit encoding [27, 84]. As genomes are composed of 4 nucleotide bases, they can be uniquely encoded using 2-bits: {`A:00, C:01, G:10, T:11`}. Each unique *k-mer* can, thus, produce a unique integer for an index, for example, `TACCTAGGAT` ⇔ `11000101110010100011` ⇔ `810147`. The number of entries in the *k-mer* index is 4^k and grows exponentially with k. The frequencies of *k-mers*, however, decrease with increasing k [85]. The size of the entire index can thus be given as $B_I \times (4^k + m - k + 1)$, where B_I is the size of integer in bytes and m is the length of genome. Longer *k-mers* result in fewer collisions as frequencies of *k-mers* decrease, however, it results in longer index lengths increasing the memory requirements of the data structure. For each collision, more entries are added to the index as shown in Figure 2.12. These entries are the locations of the *k-mer* in the HRG and the list is often known as the inverted list [16, 84].

Using suffix/prefix tries to preprocess HRG

Suffix/prefix tries are data structures that represent the set of suffixes or prefixes of a given string to enable fast matching. The advantage of using a trie is that multiple identical copies of a substring in the reference collapse on a single path, hence, alignment is only needed to be done once, whereas with a typical hash table index, an alignment must be performed for each copy [81]. The disadvantage of using a trie is the large amount of memory required to store them and for very large data sets this solution results in long computation times [82]. A trie takes $\mathcal{O}(n^2)$ space where n is the length of the reference. It becomes impractical to build tries for even small bacterial genomes [81]. Based on tries, data structures with better performance and memory efficiency have been proposed such as suffix trees, suffix array [86], enhanced suffix array [87], Burrows-Wheeler Transform (BWT) [88] and FM-Index [89]. The choice of these data structures are independent of the search algorithm used. An algorithm built for FM-Index will, in principle, work with suffix tree/trie demonstrating the similarity between the data

structures [81].

A suffix tree is a compressed trie of all suffixes of the given text. It was proposed to reduce the memory requirements during construction and storage of the data structure in comparison to tries. However, even the most efficient implementations require 12-17 bytes per nucleotide making them impractical for usage with large genomes [81]. A suffix array is closely related to suffix tree such that it can be constructed by performing a depth-first traversal of a suffix tree if edges are visited in the lexicographical order of their first character. It contain integers that represent the starting indexes of all the suffixes of a given string, after all the suffixes have been sorted. Any suffix tree based algorithm can be switched to use a suffix array with additional information and solves the same problem in the same time complexity [87]. Suffix arrays along with auxiliary arrays are used in the enhanced suffix array approach and offer improved space requirements, simpler linear time construction algorithms and improved cache locality [90]. The enhanced suffix array approach enables storing large genomes in the memory by taking 6.25 bytes per nucleotide base [82]. It offers time complexities better than the suffix array and similar to that of the suffix trees.

The FM-index is the most widely used method among the trie-based data structures as it offers small memory footprint. It is constructed with the help of BWT algorithm, which was originally developed for data compression. The BWT matrix with the Last-First array [88] consist of a tree structure including all suffixes and prefixes in the string. Searches can be done in constant time, however, it has a drawback, as the string becomes longer, the tree size also grows massively. To reduce memory requirements further, FM-Index was proposed, composing of auxiliary data structures along with the suffix array derived from the BWT matrix. The suffix array provides the locations for alignment in the HRG and when constructed using BWT altered sequence, it offers an advantage of reduced computation time [82]. The memory footprint of FM-Index can be as low as 0.5-2 bytes per nucleotide and offers a constant time backward search identical to that of a trie [81]. A detailed discussion on the construction of BWT matrix and extraction of FM-Index auxiliary tables along with the suffix array from the BWT matrix is discussed in Section 3.3.2.

2.4.2 Filtering

Filtering aims to quickly prune the reference genome to exclude large regions where no match for the read can be found. It uses the preprocessed data structures and performs approximate string search to identify the candidate locations where the reads may be located. These candidate locations identify short regions in the reference where the read can align within a given edit distance. Candidate locations are, generally, found by essentially reducing an approximate string matching problem to an exact matching problem. This is done by employing filters which divide the reads into *k-mers* or *q-grams* and search them in the reference without errors. *K-mers* and *q-grams* are, also, known as seeds or signatures in the literature. Regions that do not share the seeds are filtered out. Selecting a good combination of seeds is crucial to the performance of read mapping. A seed with low frequency of occurrence in the reference will produce fewer candidate locations to align and will reduce the overall mapping time. The novelty in the algorithms proposed in the read mapping literature, mainly, comes from the unique ways of selecting seeds. Seeds generated during filtration can be classified as overlapping and non-overlapping and can of fixed or variable-length. Depending on the heuristics used on the seeds the filtration can be lossy or lossless.

A lossless filtration guarantees to detect all possible candidate locations, while a lossy filtration attempts to detect a majority of them. The efficiency of lossy filtration is measured by two parameters, usually, called selectivity and sensitivity [91]. The sensitivity estimates the false negatives, where candidate locations of interest were missed by the filter. Selectivity, on the other hand, estimates the false positives where candidate locations that do not actually represent a solution are detected. Lossless filtration, as the name indicates, is 100% sensitive, hence, only the selectivity parameter is used, which measures the filtration efficiency [91]. Use of gapped seeds (spaced seeds or gapped *q-grams*) have been found to significantly improve the sensitivity of lossy filtration and provide order of magnitude faster filtering time. Gapped seeds are subsets of q characters in some fixed non-contiguous shape, instead of contiguous substrings and allow internal mismatches. The shape of the gapped seeds, however, significantly affect the performance and the best shapes are selected hueristically, are

Figure 2.13: Visualisation of the pigeonhole principle where a read is divided into six equal length non-overlapping *k-mers*. If the read needs to be mapped with an edit distance of five then the pigeonhole principle states that at least one of the six *k-mers* will be error free and match exactly in the reference.

rare and often possess no apparent regularity [92]. Lossy filtration does not guarantee a global optimum alignment for a read and most state-of-the-art read mappers rely on lossless filtration for accurate mapping. For detailed discussion of lossy filtration and literature using gapped seeds, interested readers can refer to [81, 83, 91, 92].

Lossless filters, typically, use one of the two lemmata: pigeonhole lemma/principle [36] and *q-gram* lemma [93]. The pigeonhole principle states that if a read with δ errors is divided into $\delta + 1$ non-overlapping *k-mers*, then at least one *k-mer* will be without error. Simply put, δ errors cannot be found in more than δ pieces of a read with $(\delta + 1)^{th}$ piece being error free, as shown in Fig. 2.13. The position of the error free *k-mer* in Fig. 2.13 was randomly chosen as the error profile of the read is not known until the mapping process is over. A pigeonhole filter, therefore, to map a read with an edit distance of δ divides the read into $\delta + 1$ non-overlapping seeds and, in parallel, searches them in a scan over the reference. The expectation is that the error free *k-mer* will match exactly at a position where it may have originated during the sequencing process. Using all $\delta + 1$ *k-mers* ensures full sensitivity during filtration.

In contrast, the *q-gram* lemma considers matching all overlapping *k-mers* or *q-grams* of a read to the reference. This methods counts and gives a lower bound threshold τ of the number of *q-grams* that match with an edit distance δ to the reference. The threshold τ is determined by the worst case scenario, where δ errors are distributed equidistantly across the read affecting $\delta\dot{q}$ of the $n - q + 1$ *q-grams*, and is given by $\tau(n, \delta, q) = n - q(\delta + 1) + 1$, where n is the length of the read [36, 93]. The reference is divided into overlapping regions of fixed length and each read and region is associated with a counter which counts the matching *q-grams* and updates as *q-gram* slides over read

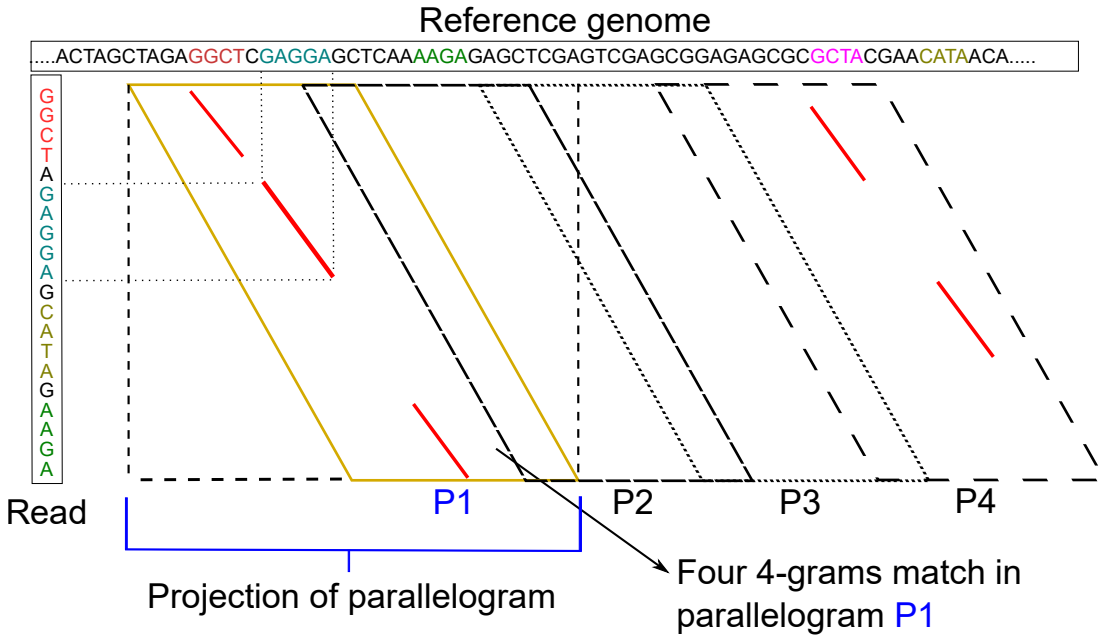

Figure 2.14: Visualisation of the q-gram lemma filtering approach used by RazerS3 [2]. This method counts the number of *q-grams* that match the projection of the parallelogram on the reference. If the count reaches a minimum threshold τ, then the projection is the candidate location. The reference is divided into equal-sized overlapping parallelograms and the process is repeated for each parallelogram.

or the reference. The *q-gram* counting filters differ mostly in the size and the shape of the overlapping region and the way matches are counted [36]. Fig. 2.14 demonstrates the filtration mechanism based on *q-gram* lemma used by RazerS3 [2] where the overlapping regions are determined by the projection of the parallelograms.

The pigeonhole principle and *q-gram* lemma offer certain advantages over each other. The selectivity of the *q-gram* lemma is higher than pigeonhole because the candidate location chosen by the former will require τ *q-grams* to match while for the latter just one *k-mer* is enough. For example, in Fig. 2.14, for $n = 20, \delta = 3$and $q = 4$, the threshold $\tau = 20 - 4(3+1) + 1 = 5$, which implies that even with four matching *4-grams* in the first parallelogram, its projection does not qualify as a candidate location. The pigeonhole principle is faster as only non-overlapping $\lfloor \frac{n}{k} \rfloor$ *k-mers* need to be searched compared to searching $(n - q + 1)$ *q-grams* for *q-gram* lemma. However, due to lack of selectivity pigeonhole principle produces greater number of candidate locations leading to more

expensive verification cycles (dicussed in Section 2.4.3). Thus, selecting low-frequency seeds becomes, all the more, important to improve performance of pigeonhole filter. Seeds, in the form of *q-grams* or *k-mers*, selected by different filtration algorithms can be of fixed lengths or variable length. The difference between both will be explained in Section 2.4.4 and throughout the thesis while discussing our contributions.

2.4.3 Verification

The objective of the verification stage is to perform an alignment of the read to the candidate locations identified in the filtration stage with the aim to identify the exact position of origin in the reference. Alignment can be of three types: global, local and semi-global. A global alignment is defined as the end-to-end alignment of two strings while a local alignment of two strings r and g is the alignment of substrings of r and g. Local alignment is performed to find regions of high local similarity because, normally, we do not know the boundaries of genes and is helpful in searching for a small read in a large genomic section. Semi-global alignment is used to perform local alignment in a global alignment setting such that string r is allowed to align with a substring of g. Conventionally, the seed-and-extend method implies that candidate location identified using the seed through filtration is extended (or verified) using semi-global alignment using a dynamic programming algorithm. Global and local alignments are, usually, performed using Needleman and Wunsch algorithm [94] and its variants like Smith–Waterman algorithm [95, 96]. For edit distance verification using semi-global alignment, most modern read mappers use the banded version of Myers bit-vector algorithm [97] implemented by Hyyrö [98]. Banded Myers bit-vector algorithm is the fastest algorithm in practice for the δ differences problem working in $\mathcal{O}(nm/w)$ time, where w is the word size in bits and n and m are lengths of the read and substring of the reference at the candidate location. We discuss our banded Myers bit-vector implementation in Section 3.3.4.

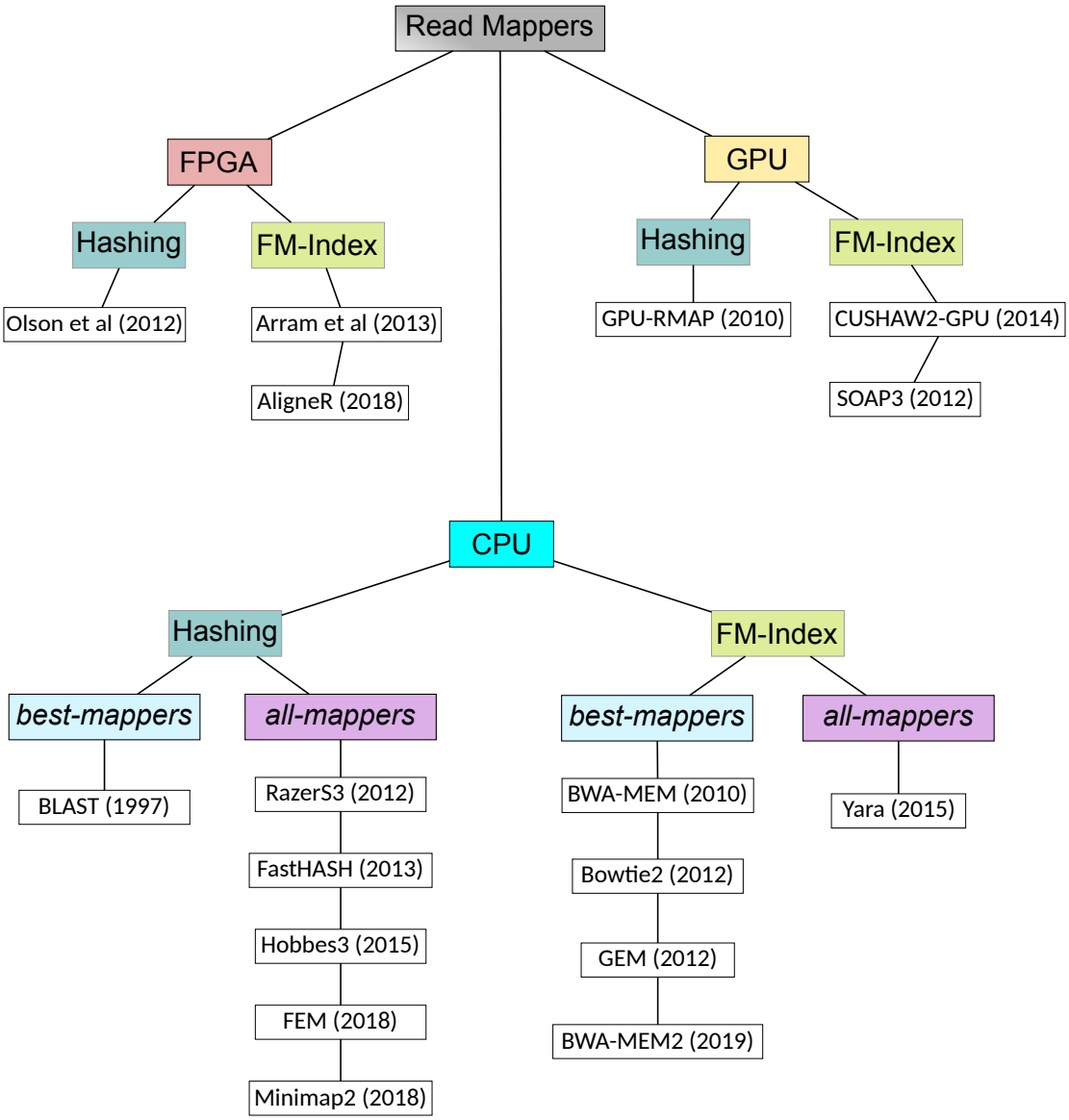

Figure 2.15: State-of-the-art read mappers targeting different hardware platforms viz. FPGA, GPU and CPU. Segregation has been performed on basis of data structures used to store the reference genome. CPU oriented mappers are further segregated into *best-mappers* and *all-mappers*.

2.4.4 Previous work

Read mapping is a compute intensive process due to the approximate string matching problem and the massive amounts of data involved, with the latter making it a memory intensive process as well. It may be performed multiple times for more insights by

varying mapping parameters such as edit distance, and to focus more on a certain genomic section for confirmation and accuracy. As discussed in Section 2.1.1, it is important to map reads to both forward and reverse strands of the HRG because the reads obtained from forward and reverse strand may, also, differ due to natural variations. For acceleration of genome re-assembly using read alignment approach, several hardware based solutions using FPGA and GPU have been proposed. Fig. 2.15 presents the recent state-of-the-art contributions towards performance oriented and energy efficient reassembly using read alignment approach where mappers targeting FPGA, GPU and CPU are shown. Below, we segregate the read mappers according to their targeted category of hardware.

Application specific and programmable hardware

An application specific integrated circuit (ASIC) or a programmable hardware is used when specialised accelerators/co-processors are required to perform a specific task or set of tasks for better performance and/or energy savings. This is possible with an optimised hardware design to perform a task better compared to existing implementations such as a software tool running on the CPU. The hardware implementations are explored when a data-centric workload becomes a significant portion of the overall workload, which is currently the case with genomics. Genomic algorithms consume large amounts of energy for its computation in data centers as quantitatively shown by Jason *et al* [99] using many tools associated with genome analysis. FPGA is a programmable logic that enables granular optimisations at the hardware level with ability to overcome limitations of fixed architecture processors and software implementations. It is used to prototype algorithms to achieve one or more desired design metric such as performance, power, energy and accuracy. FPGA-based implementations can offer significant energy savings due to the flexibility in designing parallel hardware compared to fixed hardware of CPU and GPU, where optimisations are possible only at the software level [100].

Olson *et al* [101] proposes FPGA implementation using hash-based data structure with filtering and verification stages derived from BFAST software [102]. The preprocessing is performed offline on a host and streamed to FPGA. Arram *et al* [103] implements seed-and-extend strategy using FM-Index on FPGA, where the preprocessing is

performed offline and transferred to FPGA. It implements separate processors for exact string matching (ESM) and approximate string matching (ASM). If a read fails in the ESM then it is forwarded to ASM using homegeneous, heterogeneous, and runtime reconfigurable designs. AlignerR [104] uses FM-Index to implement process-in-memory (PIM) hamming distance unit using resistive RAM memory. The main objectives of PIM is reduce memory accesses and energy consumption. AligneR presents results from simulations by modeling the power, latency and area of ReRAM arrays by NVSim and ReRAM CAMs by NVSim-CAM.

A detailed review of FPGA-based implementations of genome reassembly can be found in [30] and mentions of latest literature can be found in [105, 106]. Genome reassembly is, generally, difficult to implement on the FPGA due to large memory footprint of the data structures. FPGA implementations, often, find it challenging to include indels, which limits the mapping accuracy. The programming effort involved with FPGA is very high and requires knowledge of hardware description languages (HDL), software suites and electronic devices. Such skills are, often, not common in the end-users including geneticists and bioinfomaticians. In addition, any change in parameters, such as the read length, reference genome and edit distance, will require rewriting of the program followed with optimisation and validation cycles. For example, Arram *et al* [107], implements FM-Index based algorithm, however, limits itself to two mismatches excluding indels, therefore, limiting the mapping accuracy. To the best of our knowledge, there is no impetus on designing ASIC for genomic workloads as the characteristics of raw data is changing rapidly with advances in the sequencing technology. Along with data, the genome analysis tools are evolving as well [108] which discourages efforts and expenditure towards ASIC as they will be rendered useless.

GPU

GPUs are parallel computing machines with 100-1000s GPU-cores, which are basically arithmetic logic units (ALU), capable of performing massive number of operations, concurrently. They are optimized to accelerate floating-point operations using single-instruction multiple-data (SIMD) instructions. Unlike FPGA, the internal architecture of GPU is fixed and, hence, its optimum usage is dependent on the algorithm and

the programming effort. The amount of memory available on GPUs is, often, more than FPGA but fixed and limited by the vendor. GPU-based mappers have been extensively researched to reduce the mapping time. Many GPU-based implementations are extensions of CPU-based algorithms and focus on intratask parallelism to accelerate a part of the mapping algorithm, often highly parallel and compute intensive, rather than the whole algorithm.

CUSHAW [109] employs a quality aware heuristic to perform a bounded search using FM-Index data structures. It supports, only, substitutions and not indels. CUSHAW2-GPU [110] is an optimised implementation for CUDA-enabled GPUs. It focuses on inter-task parallelism where CPU and GPU both perform read mapping concurrently. GPU-RMAP [111] is a GPU adaptation of hashing based RMAP algorithm [112] optimised for Nvidia GPUs. It preprocesses the reads to create the hash table and divides genome into several independent segments. The segments are then scanned in parallel, and a score is assigned to the mapping locations of all the reads. Binary search algorithm is used on the hash table. The hash table is transferred to GPU after sequential construction in the CPU. With the increase in the number of reads the overall computational speed decreases. SOAP3 [113] is based on BWT data structure and is capable of handling only mismatches. It heuristically determines reads which produces many divergent branches and process them using CPU. It uses only the global memory which limits its performance. SegAlign [114] is a GPU accelerated whole genome aligner which is used in comparative genomics to understand genomes of different species together. It is not a read mapper, however, it use the same seed-filter-extend strategy and accelerates seeding and filtration stage of LASTZ [115] on GPU and extends on CPU cores. It employs hash based data structure for seeding and is capable of roping in multiple GPU nodes for speedups. A detailed discussion on other state-of-the-art GPU-based implementations is presented in [30, 106].

In comparison to FPGA, GPU requires less programming effort and time to deployment, however, it has been observed to provide lower performance and energy efficiency for read mapping [30]. Mapping tasks are, often, memory intensive and require frequent accesses to global memory with poor locality attributes. GPUs are not good at handling divergent instructions and its programming, often, requires vendor-specific software

for implementation and managing SIMD threading [116]. A major bottleneck to GPU-based read mappers is that the dynamic programming based verification stage has dependencies which are difficult to compute efficiently in parallel and can be handled better with a CPU core. Additionally, with the change in GPU architecture the mapper requires reprogramming or tailoring to efficiently use its capabilities. Although GPUs are categorised as power hungry devices, they are available in most modern heterogeneous systems including general purpose computers, data centers, and embedded platforms such as smartphones. The widespread availability calls for their energy-performance profiling to find if they can perform genomic computations fast enough to produce energy savings despite high power requirements.

CPU

Conventionally, read mappers can be classified as *best-mappers* and *all-mappers*. *Best-mappers* aim to directly find the 'best' location for mapping the read while *all-mappers* attempt to enlist all set of locations within a given edit distance. The selection of either of the classes depends on the need of downstream analysis. In many applications, it is desirable to find all mapping locations such as ChIP-seq experiments [117], RNA-seq transcript abundance quantification, CNVs (copy number variation) calling and detecting structural variants [27, 118, 119]. *Best-mappers* employ heuristics to select the 'best' mapping location over the others for a read. The heuristics may use the read quality score obtained from sequencing or penalising indels differently than substitutions during alignment. Ideally, *best-mappers* should outperform *all-mappers* in mapping times and selectivity as *all-mappers* will need to perform more expensive DP-based verification cycles, however, the results published by state-of-the-art *all-mappers*, such as, RazerS3 [2] and Hobbes3 [16] demonstrate otherwise.

Similar to FPGA and GPU-based read mappers, we can segregate CPU-based read mappers on the type of preprocessing methodology used. Generally, most read mappers have been first implemented and/or optimised for CPU and then adapted to FPGA and GPUs. Hash table based mappers include BLAST [120, 121], RazerS [122], FastHASH [123], RazerS3 [2], Hobbes3 [16], FEM [27] and Minimap2 [124]. BLAST was one of the most widely used tool before the advent of HTS. It implements seed-and-extend

strategy using hash lookup tables, refines the results using Smith–Waterman DP-based local alignment algorithm and reports statistically significant alignments. RazerS is free from preprocessing stage and constructs a q-gram index in real-time using the read sequences. It employs *q-gram* lemma filtering approach proposed in SWIFT [93] and depending on a user given sensitivity value provides a tradeoff between mapping time and sensitivity (mapping accuracy). Details of SWIFT implementation of *q-gram* lemma was discussed in Section 2.4.2. RazerS3 [2] introduces an option to use pigeonhole principle with sensitivity control. It employs shared memory parallelism where the sets of reads are assigned to threads during the filtration stage and the candidate locations are dumped in a global space. The same threads, then, verify the candidate locations not, essentially, in the same order while keeping a track of the thread ids the read belongs to. This dynamic reallocation of reads to thread in the verification stage balances the load and keeps the thread busy achieving better performance.

FastHASH [123] employs two heuristics in the filtration stage to reduce the number of verification cycles viz. Adjacency filter and Cheap *K-mer* selection (CKS). Adjacency filter is an adaptation of pigeonhole principle, which divides the read into N parts and only the candidate location which are present adjacently in the location list of $(N - \delta)$ corresponding adjacent *k-mers* are verified. It, basically, means that $(N - \delta)$ *k-mers* will be error free and hence, all true candidate locations should be present in the location list of that many *k-mers*. Adjacency filter induces additional computational burden as it conducts binary search through location lists of N *k-mers* and depending on lengths of location lists (occurrence frequency), it can grow quadratically. To prevent selection of *k-mers* with high frequency, CKS sorts all possible *k-mers* using quicksort and selects them based on their frequency. CKS aims to reduce the computational burden of Adjacency filter.

Hobbes3 [16] is the latest read mapper in the Hobbes series which follows an approach similar to FastHASH. Similar to Adjacency filter, Hobbes3 divides reads in $(\delta + 2)$ *k-mers*, which would leave at least two *k-mers* error-free. Hence, a true candidate should be adjacently found in the location list of at least two *k-mers*. CKS is a heuristic based approach where fixed length seeds are used and the seeds which are least frequent are selected, however, Hobbes3 implements a DP-based filtering approach

which selects variable length seeds after exploring the combinations of *k-mers* that yield least verification cycles. FEM [27] is a recently proposed mapping tool which constructs a succinct hashing index with low memory footprint to preprocess the reference genome. It employs pigeonhole principle and uses DP-based seed selection approach from Hobbes to select optimum k-mer lengths to minimise the total number of candidate location ensuring full sensitivity. The succinct hash index provides considerable lower memory footprint than Hobbes3. Minimap2 [124] reports the first N mapping locations per read. It it capable of mapping both short and long reads, however, the performance for for mapping the long reads is better than the short reads. Minimap2 demonstrates speedups of 2-3$\times$ over Bowtie2 and BWA-MEM for simulated short reads but the mapping times are not compared when real reads were used. The accuracy reported is similar to *best-mappers*: Bowtie2 and BWA-MEM.

State-of-the-art FM-Index and BWT based read mappers include BWA-MEM [25,125], Bowtie2 [26], GEM [28], Yara [24] and BWA-MEM2 [126]. Among these BWA-MEM, Bowtie2 and GEM are *best-mappers* while Yara is an *all-mapper*. FM-Index based mappers use backward search algorithm to find candidate locations for seeds, discussed in detail in Section 3.3.3. Bowtie2 enhanced Bowtie by allowing indels and quality aware backtracking heuristic. Using SIMD instructions available on modern CPUs, it improved the performance of DP-based verification stage. BWA-MEM employs DP based filtration and heuristics to report best mapping locations for the reads. A detailed comparison of Bowtie2 and BWA-MEM is presented in [127]. BWA-MEM2 enhances the BWA-MEM for Intel processor architecture by improving cache utilisation, memory allocation, prefetching and algorithm. It demonstrates a 2.4$\times$ speedup over BWA-MEM. GEM employs heuristics to find the best matching position for a read. It uses adaptive *seeds* to reduce the total number of candidate locations during filtration and uses Myers bit-vector algorithm for verification. Yara [24] uses pigeonhole along with approximate seeds to increase specificity of filtration and stores the reference genome using FM-Index and suffix arrays. This thesis focuses on *all-mappers* which includes the mapping locations reported by *best-mappers* provided they run in full sensitivity mode. In this thesis, we have compared our contributions to state-of-the-art *best-mappers*, viz. BWA-MEM [25], GEM [28], Yara [24], and *all-mappers*, viz. RazerS3 [2], Hobbes3 [16] and

FEM [27]. We have not compared with BWA-MEM2 and Minimap2 as they got published recently compared to the timeline of the thesis, however, the speedups reported over BWA-MEM by this thesis is better than BWA-MEM2 and Minimap2.

Resource metrics	Computation Design Space	Performance metrics
• Energy • Cost	• Hardware choices – CPU, GPU, FPGA, Embedded platform, ASIC, ... • Data structure choices – Hash index, FM-Index, Suffix Array,.. • Algorithmic choices – Pigeonhole/q-gram filter, fixed/variable length seeds, seed extraction method, ... • Parameter choices – short/long read, hamming/edit distance, genome section/chromosome/whole genome, ... • Implementation choices – Programing language/framework, parallel/pipelined, precision, approximate arithmetic, ...	• Throughput • Accuracy

Figure 2.16: Metrics that determine the algorithm-hardware co-design approach with the available design space choices.

2.5 Algorithm-Hardware Co-Design

The new hardware systems being designed perform well for many existing model algorithms but they do not include genomic algorithms, yet. Likewise, the genomic tools have been developed and used mainly on general purpose computers. Only recently engineers are exploring genomic workloads on various hardware platforms for speedups and energy efficiency. Very few work exist on emerging embedded platforms which offer low power/energy characteristics. In this thesis, we use the algorithm-hardware co-design to establish a feedback between data-centric computing and embedded platforms to develop an embedded genomic solution. Fig. 2.16 presents the components of algorithm-hardware co-design where the resource and performance metrics determine the choices made in computation design space.

In this thesis, we use CPU, GPU and embedded boards as our targeted hardware platforms. The tools proposed in the thesis can work on all the three platforms as it uses OpenCL computing framework. Even though GPUs are power hungry devices they are available in most modern heterogeneous computers. Thus, in the context of existing computing infrastructure GPU should be explored to find if energy can be saved by

lowering computation time at the cost of power. To program Nvidia GPU, generally, CUDA toolkit [128] is used as it is tailored to its architecture while OpenCL can be used with GPUs from other vendors, as well, such as AMD and Intel. This is because OpenCL views every OpenCL conformant device [32] with the same hardware and software level abstractions. OpenCL allows task-level parallelism with concurrent kernel executions on multiple devices with low programming effort compared to CUDA. We discuss OpenCL in Chapter 3 in greater detail. To learn programming using OpenCL the readers can refer to the book authored by Matthew Scarpino [129].

We use embedded platforms as it a modern computing platform emerged as a consequence of internet revolution. Today, they occupy space in almost every field of engineering and massively used in daily lives of people in the form of consumer electronic devices. It is specifically designed for low power and low energy scenarios but are capable of working as single board computers (SBC). It can be used as portable handheld device to be used at point-of-care scenario of P4 medicine discussed in Chapter 1. A portable handheld genomic device can mitigate privacy concerns arising due to massive growth in data and its applications. To prove the 2^{nd} hypothesis of Chapter 1, we need to examine if simpler ARM cores on embedded platforms can utilise energy better than complex Intel cores in CPU for integer-based operations of genomic workloads.

In this thesis, we propose algorithms that use pigeonhole principle with variable length seeding to reduce mapping time. Using pigeonhole principle, fewer *k-mers* are pruned and variable length seeding increases specificity producing fewer candidate locations. The reference genome is indexed using FM-Index, Suffix Array and additional auxiliary data structures proposed in the thesis to reduce the memory footprint. The tools are validated using both simulated and real human reads of length 100-150 mapped with an edit distance of $\leq$5%.

2.6 Summary

A tremendous amount of research is being conducted on the WGS pipelines, especially, since the advent of HTS. The sequencing technology is continuously progressing with

novel technologies cropping up. The newer technologies focus on longer read lengths with fewer errors. Recent developments in single molecule technology is being termed as third generation sequencing technology capable of producing very long reads. However, challenges persist from the error profile of such long reads as they have high error rates. Long reads with high error rates need novel solutions from the computational front. SGS reads are shorter in length but have lower error rates. The shortcomings of short length reads are compensated by their sheer numbers through amplifications during the sequencing process. A detailed discussion on the SGS technology has been presented in this chapter.

Low cost sequencing has made data easily available for sharing and research leading to ample opportunities on improving the computational pipelines of WGS. Genome reassembly algorithms have been evolving to keep up with the growth in data. Two broad categories of reassembly approach exists viz. *de novo* and read alignment approach. *De novo* approach does not require a reference template genome to pre-exist and directly assembles the reads using the overlapping between them. This process involves the construction of large graphs and is memory and compute intensive. It is usually used for new species or in cases with focused requirements. The fundamental algorithms used to solve the *de novo* problem were discussed in this chapter with their advantages and shortcomings. A discussion on the state-of-the-art *de novo* assemblers was, also, presented in this chapter.

The read alignment approach assumes that a reference template of the genome of a species exists to which the reads obtained from fresh samples can be mapped. This significantly reduces the computational burden, since, the structure of genome is already known. The mapping is performed using approximate string matching algorithms that permit differences between reads and the reference to account for sequencing errors and natural variations. Most read mappers in this category, broadly, navigates through three stages: preprocessing, filtering and verification. Preprocessing stores reference genome or the reads in the form of data structures that permit quick pruning which mapping. Filtering employs algorithms which navigates through the preprocessed data structures to identify the candidate locations where reads are likely to align. Verification conducts an in-depth alignment of the read to the region of the reference genome

identified by the candidate location allowing for mismatches and indels. Read alignment approach is significantly faster than the *de novo* approach, however, with the availability of massive amounts of data the need for efficient mapping implementations persists. Various hardwares such as CPU, GPU and FPGA have been explored to speed-up the mapping process. This chapter presents a comprehensive review of state-of-the-art implementations from the last two decades with their achievements and shortcomings. This chapter paves the path to the major contributions of the thesis which attempt to mitigate the shortcomings.

Chapter 3

Verification-aware Read Mapper for Heterogeneous Systems

3.1 Overview

Central to obtaining genomic data is the process of whole genome sequencing (WGS), which involves collection of sample, sequencing and reassembly of genome followed with downstream analysis to extract relevant information. The computational pipelines of genomics includes genome reassembly and its analysis. Initially, most of the focus was on optimising the computational pipelines for high performance targeting CPUs in general purpose computers, workstations, servers and data centers. Numerous contributions proposed novel approximate string algorithms and its associated data structures to reassemble the genome. These algorithms were then optimised and implemented on CPUs demonstrating order of magnitude speedups over older counterparts. With the maturity of SGS technology, the read sizes have increased with changes in the error profile. For recent SGS reads, read mapping process needs to include insertions and deletions (indels) along with mismatches sighting high accuracy, making genome reassembly more challenging. In attempt to shorten the gap between data production and processing rate, numerous emerging hardware solutions have been explored which includes implementations targeting FPGA and GPU. A detailed discussion on this was provided in Section 2.4.4.

A large proportion of genomic data, today, is generated in centralised government

or commercial genomic centers. As of November 2021, there are only 13 NHS Genomic Medicine Centres [130] in the UK. The processing of the genomic data is performed at the genomic centers or may be outsourced to a commercial data center via cloud computing. In either of the cases, the computing machines employ state-of-the-art high-performance many-core systems and incur high energy consumption [99]. To realise the P4 scenario, the installation and maintenance of such high-performance systems will be a bottleneck in setting up of affordable genomic healthcare infrastructure. Additionally, most modern platforms available are heterogeneous and consists of at least CPU and GPU together. To the best of our knowledge, no read mapper has been reported to, concurrently, use both CPU and GPU. As discussed in Section 2.4.4, often a part of the mapping tool is accelerated on the GPU and not the entire pipeline. Hence, WGS computation pipeline requires the exploration of a platform independent computing framework to use all the available resources on the system for effective performance gains.

This chapter presents a **C**ross-platf**O**rm **R**ead m**A**pper using openc**L** (CORAL) to map reads on any OpenCL conformant device. Today, a majority of platforms manufactured by different vendors comply with OpenCL standards [32]. We use OpenCL framework [129] as a baseline to design the CORAL kernel and apply a series of algorithmic optimisations to subdue memory constraints. The aim is to enhance the portability of our aligner across various devices and platforms mitigating the need for restructuring or rewriting. CORAL is equipped to launch kernels, simultaneously, on all the available compute units, provided enough memory is available, to distribute the workload and achieve enhanced performance. With this feature, we address the limitations encountered in multi-device heterogeneous systems, ranging from servers to workstations, where all devices can map reads concurrently to speedup the process. The CORAL algorithm is fully sensitive and capable of reporting all mapping positions, however, the actual number of mappings reported is, mainly, limited due to the memory allocation restrictions imposed by OpenCL. We elaborate on this further in Section 3.4.1. CORAL is verification-aware as it dynamically adapts the *k-mer*[1] length during filtration to reduce verification costs. It employs FM-Index [89] backward search to detect the number of candidate locations for a particular *k-mer* and in accordance with a threshold,

[1] *k-mer* is a subsection, of length *k*, of a read or genome.

it extends the *k-mer* to reduce the number of candidate locations to be verified. The candidate locations are obtained from suffix array data structure [86], preprocessed using reference genome, and are verified *in-situ* using banded Myers bit-vector algorithm [97, 98]. CORAL, automatically, determines the number of workitems (or threads) in a workgroup for a particular device based on user given workload allocation, distributes the workload and executes them in a task-parallel fashion. The host code is written in `Python` and kernel in `C` using `PyOpenCL` primitives. `Python` enables low programming effort and fast prototyping. Using CORAL, this chapter addresses the first hypothesis stated in Section 1.2. CORAL is available at: https://github.com/nclaes/CORAL

3.2 Background

In Section 2.4.4, a detailed discussion on state-of-the-art read mappers targeting different devices was presented. This chapter compares CORAL with both *best-mappers*, such as BWA-MEM [25], Bowtie2 [26] and GEM [28], and *all-mappers*, such as RazerS3 [2], Yara [24], Hobbes3 [16] and FEM [27]. Table 3.1 shows the characteristics of aforementioned state-of-the-art read mappers. These mappers mentioned are based on read-alignment approach which assumes the availability of the reference genome. As discussed in Section 2.4.4, they can be categorised as *best-mappers* and *all-mappers* which report the best and all mapping locations. In addition, *all-mappers* can, also, be tuned to report first-n mapping locations. This approach has three stages: preprocessing, filtration and verification, which were discussed in detail in Section 2.4 using Fig. 2.11. To set the context, this chapter describes the stages briefly. The preprocessing stage uses data structures such as hashing, FM-Index [89] and suffix arrays [86] to store the reference genome. Filtration uses the preprocessed data structures and performs approximate string search of the reads with the reference genome. It prunes the reference genome using q-gram lemma or pigeonhole principle [36] to identify candidate locations where the reads may be located. The verification stage identifies the exact mapping location of the read. Most modern mappers employ banded Myers bit-vector algorithm [97, 98], which is a variant of semi-global dynamic programming. As verifications are expensive and runtimes of dynamic programming increase exponentially with the length of the strings, efficient filtration techniques are desired to narrow down the search space and

reduce the total number of candidate locations per read. Every aligner, thus, employs a filtration methodology in order to extract low-frequency *k-mers* from the read, to reduce mapping time.

Several investigations have been reported on performance and power-driven explorations of simultaneous kernel execution on CPU and GPU. Prakash *et al* [131] use OpenCL to run benchmarks, concurrently, on CPU and GPU cores of Odroid XU3 embedded platform. The goal is to exploit heterogeneity for better power-performance tradeoffs. They test it using Polybench benchmark suite [132] which contains general-purpose computing workloads. In [133], the authors present an investigation of simultaneous kernel execution on both CPU and GPU in a fused CPU-GPU architecture with shared LLC, mainly targeting the Intel platforms as they are the only one that, currently, supports OpenCL 2.0's fine grained SVM. They dynamically allocate the workitems, of Rodina benchmark suite, on devices to maximize performance. Dynamic work-item allocation is an ability provided in OpenCL 2.0 standard. Singh *et al* [134] propose an energy-efficient run-time thread mapping and partitioning methodology for concurrent applications on Odroid XU3. All aforementioned works investigate the scheduling of kernels and partitioning of workitems between CPU and GPU on different platforms. They use workloads from standard benchmarking suites and attempt to improve performance-power tradeoffs. CORAL, on the other hand, is an application specific tool cross-platform read mapper. It utilizes 2×Nvidia GTX 590 GPUs with 4×Intel CPU for high performance without additional programming effort.

Table 3.1: Characteristics of state-of-the-art read mappers along with CORAL, proposed in this chapter. *Other* in the table implies a combination of multiple data structure, search algorithms and heuristics, the one, specifically, employed by BWA-MEM.

Read mappers	Cross-platform	Type			Preprocessing			Filtering		
		Best	All	First-n	Hashing	FM-Index	Suffix Array	Pigeonhole	q-gram lemma	Other
RazerS3			✓		✓			✓	✓	
Hobbes3			✓		✓			✓		
Yara			✓			✓	✓	✓		
FEM			✓		✓			✓		
GEM		✓				✓	✓	✓		
BWA-MEM		✓				✓	✓			✓
CORAL	✓			✓		✓	✓	✓		

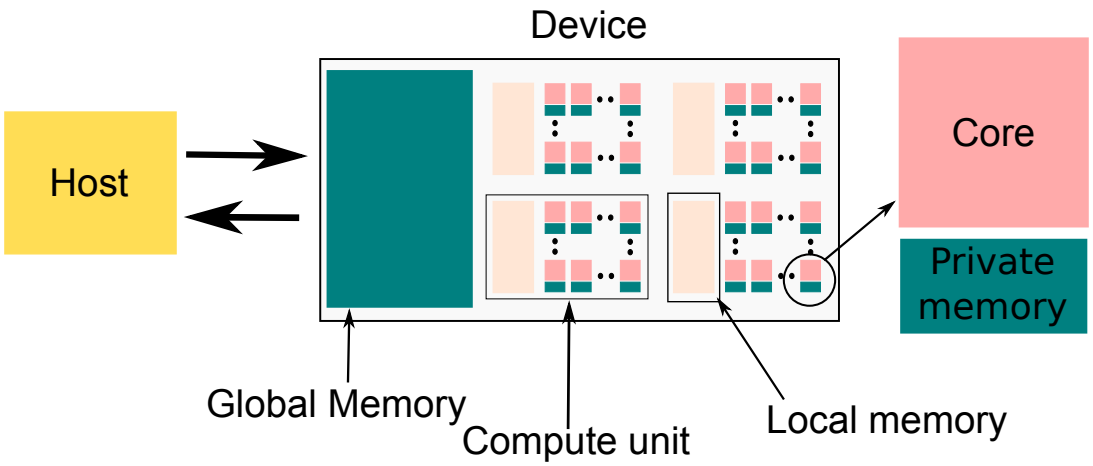

Figure 3.1: OpenCL programming model with memory hierarchy.

3.3 Methods

In Section 3.3.1, OpenCL's unique way of looking at any hardware, which enables CORAL to be used across different devices and platforms, is discussed. Section 3.3.2 presents the preprocessing approach which stores the HRG using FM-Index and suffix array. Details on the advantages of verification-aware filtration is provided Section 3.3.3. Section 3.3.4 discusses the banded Myers bit-vector algorithms [2, 97, 98] used for verification. Finally, Section 3.3.5 elaborates the CORAL algorithm and its implementation approach.

3.3.1 OpenCL view of the hardware

Fig. 3.1 visualizes how OpenCL [129] views a compatible hardware. From an execution standpoint, it recognizes two computational divisions viz. host and device; and three layers of memory with different access rights viz. global, local and private memory. Host is a software abstraction which issues instructions and data to the hardware device for execution. The host and device communicate data through the global memory i.e. host cannot access the local and private memory of the device directly but rather through the global memory. The host and device need not be different platforms like a CPU-GPU pair or CPU-FPGA pair, it can be CPU-CPU pair where the host and device share the same global memory and compute resources in different time slots, meaning the host

can launch kernels on the device its running along with other available devices. As we know, the memory hierarchy in CPU consists of off-chip RAM and on-chip caches. Most CPUs have three levels of caches viz. level 1 (L1), level 2 (L2) and level 3 (L3), with increasing size and access times. These caches hold data which are accessed frequently, recently or both to improve the runtime of programs. GPU, in general, have "CPU-like" cores with multiple arithmetic logic units, cache and registers. Depending on the vendor, the internal architecture of the core varies along with the definition of GPU-core, however, all "CPU-like" cores in modern GPUs have registers and cache, generally, referred as L1 cache. OpenCL, generally, recognizes off-chip RAM and the L1 cache as global and local memory, respectively. Private memory, generally, are the registers available to the cores. Host issues instructions in the form of kernel and workitems (or threads) execute them using different data elements. Workitems are equally divided into workgroups, with each workgroup occupying a single compute unit during execution. The workitems within a workgroup execute on all the available cores in the compute unit. As all compute units have separate L1 caches, all the workitems inside a workgroup share the local memory. The private memory, however, is only accessible to the workitem deployed on the core.

It is imperative to consider the memory capacities at all levels while designing the kernel. An efficient kernel minimizes private memory usage and the intra-data movements between private, global and local memory. There is no limit on the number of workgroups, however, the maximum number of workitems allowed in a workgroup depends on the device specification along with the private and local memory consumed by the kernel. Generally, GPUs require low memory footprint kernels to achieve higher utilizations by engaging a greater number of workitems in a workgroup. Thus, designing of kernels with memory constraints require series of algorithmic optimization with respect to hardware. Our proposed CORAL kernel requires 380-480 bytes of private memory, depending on the read size, and does not use the local memory. A discussion on the scaling of memory footprint of the kernel with read size is presented in Section 3.5. We discuss about the scaling of memory footprint of CORAL kernel with read lengths in detail in Section 3.5. Discarding the use of local memory in the kernel enhances the portability of CORAL as the size of local memory varies across different devices. Each

workitem executes the kernel for a single read and performs the following operations: loading read in the private memory, identifying the candidate locations for both forward and reverse strand, performing *in situ* verification and writing the verification result back to global memory. Asynchronous execution is possible as each workitem executes a single read, independent of the others.

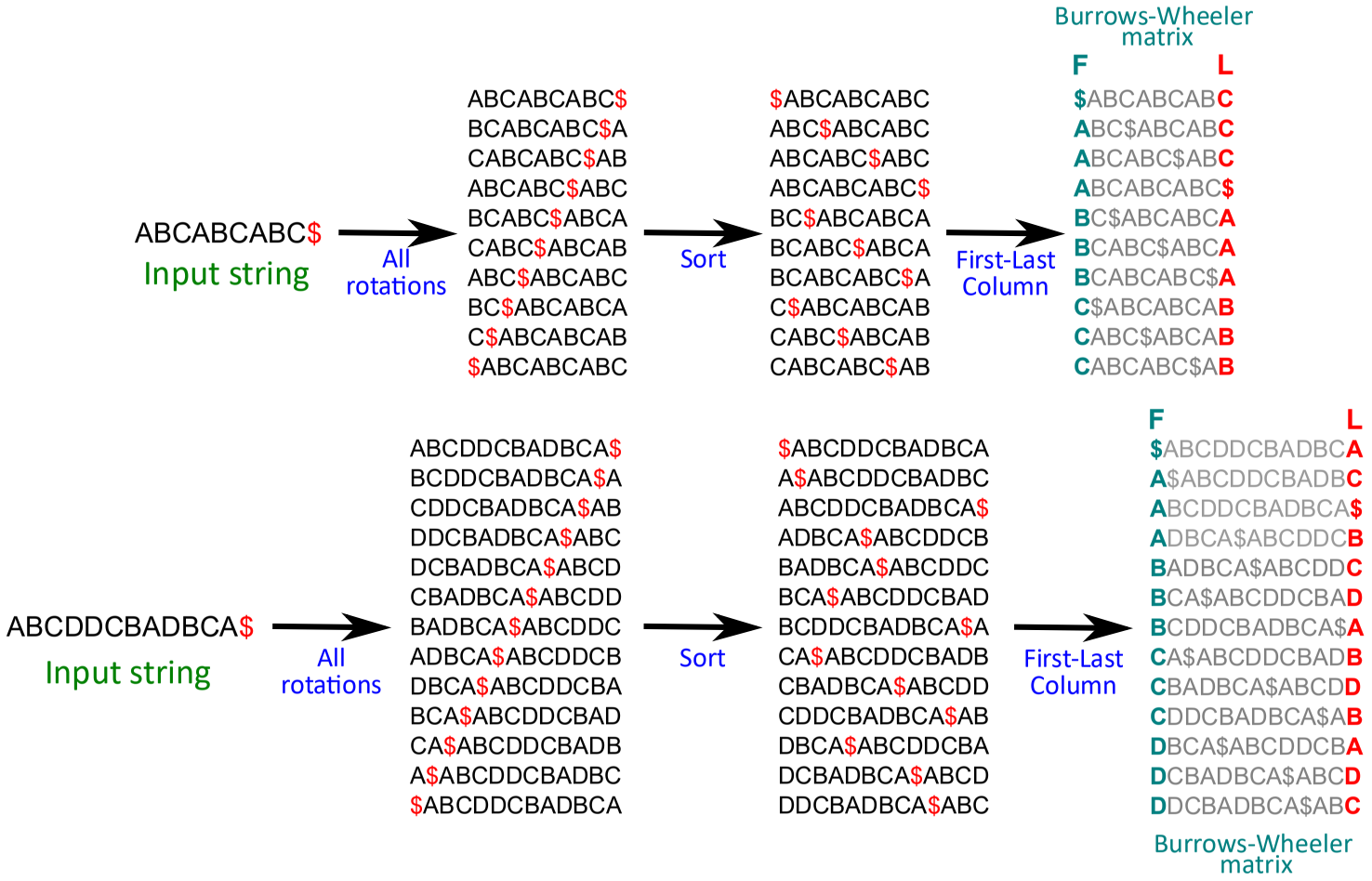

Figure 3.2: Visualization of Burrows-Wheeler Transform and its usage in compression. BWT lexicographically sorts the permutations of the string and stores only the first(**F**)-last(**L**) arrays. Both **F** and **L** arrays can be stored using run-length coding, if possible, thus compressing the string. The **F** and **L** arrays for string `ABCABCABC$` can be stored as **F**=`$3A3B3C` and **L**=`3C$3A3B`. However, for string `ABCDDCBADBCA$`, BWT doesn't produce any compression as **L**=`AC$BCDABDBADC`.

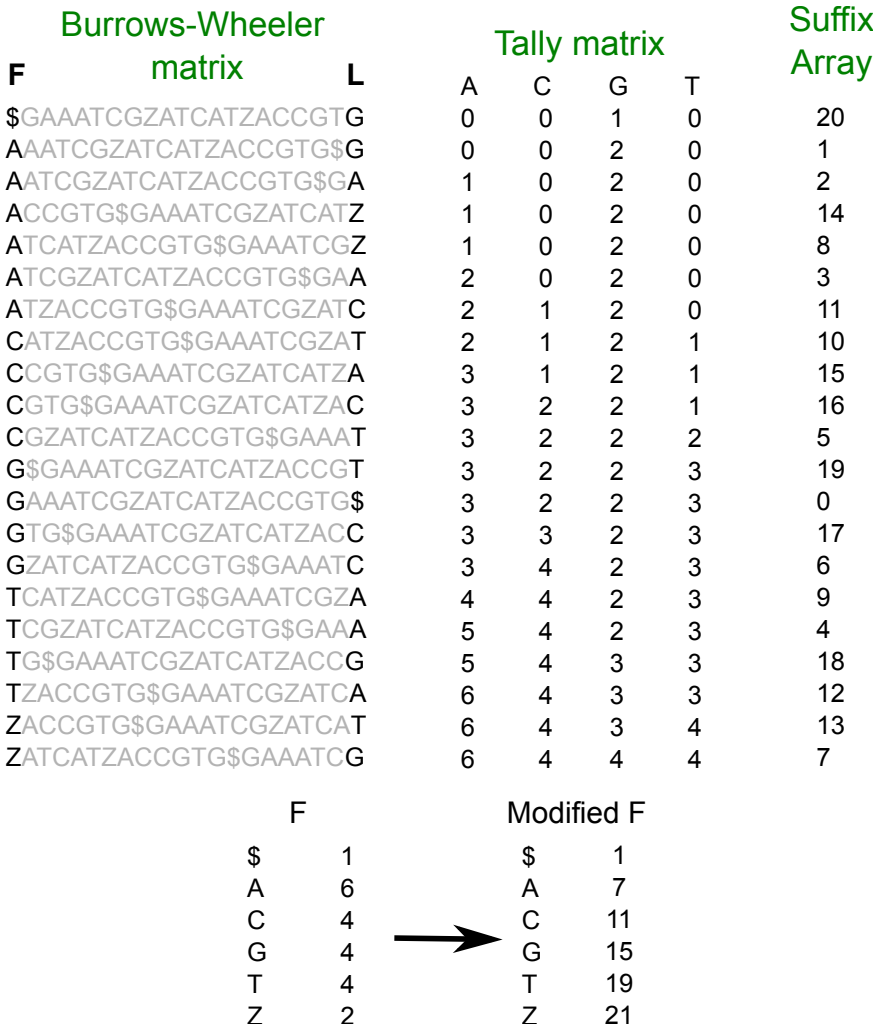

Figure 3.3: Visualization of preprocessing methodology of CORAL for a small sequence: `GAAATCGZATCATZACCGTG$` using FM-Index and suffix arrays. We store the tally matrix, suffix array and the modified **F** array to be used for querying *k-mers* in the filtration stage.

3.3.2 Preprocessing

We use FM-Index and suffix array data structures to store the reference genome. FM-Index uses the first and last arrays, denoted as **F** and **L**, of a matrix obtained by applying Burrows-Wheeler transform (BWT) [88] on a string. BWT lexicographically sorts the list of all reversible permutation of characters of a string. Fig. 3.2 visually presents the BWT steps and how it can lead to compression. The string is appended with an end-

of-file character, such as **$**, which is required to reconstruct the original string from the compressed form. The final step produces the BWT matrix with first(**F**)-last(**L**) arrays that can be stored using run-length coding, if possible, thus compressing the string. We demonstrate compression using the examples shown in Fig. 3.2 and demonstrate, how, it may or may not compress the string. For string `ABCABCABC$`, the resulting **F** and **L** arrays can be stored using run-length encoding as **F**=`$3A3B3C` and **L**=`3C$3A3B`. However, for string `ABCDDCBADBCA$`, BWT does not produce any compression as **L**=`AC$BCDABDBADC`.

Fig. 3.3 presents the preprocessing strategy of CORAL using an example sequence: `GAAATCGZATCATZACCGTG$`. The **L** array from BWT is used to construct the FM-Index auxiliary data structure called tally matrix, where each element in a row stores the number of occurrences of a particular base starting from the first row to the row where the element is present. It, basically, keeps a track of the numbers of each type of base encountered. Alongside, a suffix array is obtained that stores the position at which a particular base in **L** occurs in the original string. **F** array can be compressed using run-length encoding to represent the number of occurrences of all the alphabets in the string. We further modify **F** to provide cumulative numbers in the increasing order rather than the exact number of occurrences, as shown in bottom right of Fig. 3.3. The undetermined bases of reads due to sequencing errors are represented as `N`. In our proposed preprocessing methodology, we replace `N` with `Z` as it occurs in the end of the lexicographical order of alphabets, thus, appearing in the end of **F** array, after bases `A`, `C`, `G` and `T`. For further details on FM-Index and suffix arrays, interested readers may refer to [86, 89, 135].

3.3.3 Verification-Aware Filtration

We demonstrate our pattern matching methodology using an example where pattern `ATC` is searched in the text `GAAATCGZATCATZACCGTG$` using FM-Index backward search, as shown in Fig. 3.4. We demonstrate how search results can be extended if the pattern changes to `AATC` by prepending a character in the beginning, thus, increasing its size by one. Following that, we explain how the pigeonhole principle along with verification-aware FM-Index backward search can reduce the total number of candidate locations without affecting sensitivity.

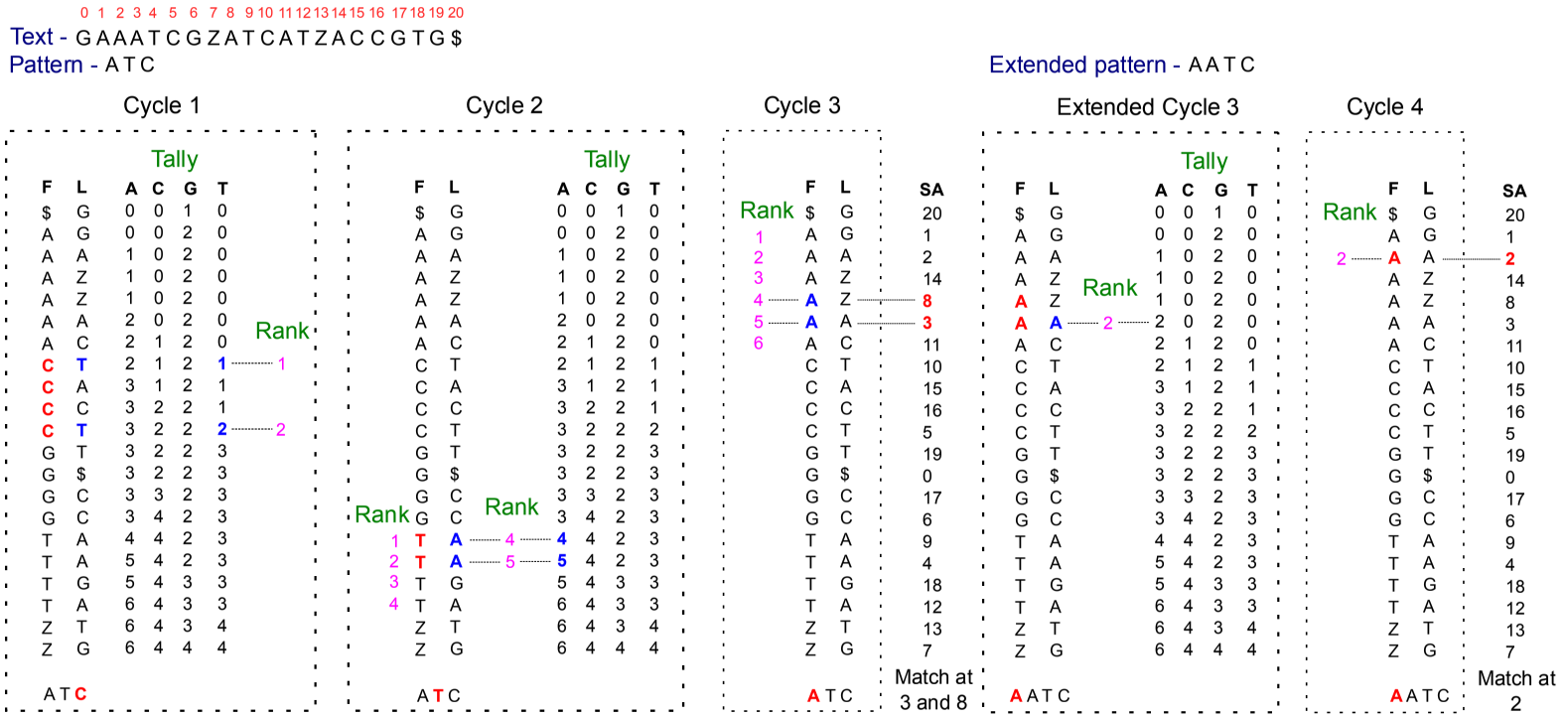

Figure 3.4: Visualization of the searching method using FM-Index and suffix arrays. We search for pattern: `ATC` in the text: `GAAATCGZATCATZACCGTG$` in three cycles and then show how the search can be extended if the pattern size increases on dynamically to `AATC`, using four cycles.

FM-Index Backward Search

Before we can proceed with FM-Index backward search, the concept of ranking of characters in the text is required to be understood. The rank of a character indicates the number of times that character has occurred in the text including the current instance. For example, the rank of the underlined highlighted base `A` in `GAAATCGZATCATZACCGTG$` is 4. The purpose of the tally matrix obtained in Section 3.3.2 is to store the ranks of all the desired characters in the text. We do not store ranks for `Z` or `N` as they are considered errors and the corresponding *k-mer*, where it occurs, need not be searched.

Searching starts from the last character and moves up to the first taking as many cycles as the length of the pattern. For example, to find pattern `ATC`, the first cycle identifies the number and positions of occurrence of `C` in the **F** array, as shown in Fig. 3.4. It then looks for the occurrence of next character i.e. `T` in the corresponding locations in **L** array and store the corresponding ranks from the tally matrix. We can see that `T` precedes `C` at two locations with ranks 1 and 2. Cycle 2 uses the, previously, stored ranks to locate the corresponding `T`s in the **F** array. We, again, look for the next character i.e. `A` in the corresponding locations in **L** array and store their ranks from the tally matrix. At the end of cycle 2, we find that `A` precedes `TC` at two locations with ranks 4 and 5. Cycle 3, then, locates `A` in **F** array and reports the corresponding locations of occurrence from the suffix array. Fig. 3.4 shows that pattern `ATC` matches the text at positions 3 and 8 (zero based numbering). As the pattern matching begins from the last base backwards, the method is known as backward search. We can further continue searching if the pattern size is increased in a similar fashion using extended cycle 3 and cycle 4. At the end of cycle 4, the pattern `AACT` can found at location 2 with help of the suffix array. FM-Index, thus, offers flexibility in the variation of length of *k-mers* by prepending more bases, each of which can searched in fixed time $\mathcal{O}(1)$. For read mapping, we do not need to store the **L** array but require the tally matrix, modified **F** array, suffix array and the reference genome.

Table 3.2: Charaterization of number of occurrences of *k-mers* for $k = 16, 17, 18, 19, 20, 21, 22$ in chromosome 2. Values given are in percentage approximated to the nearest decimal.

Occurrence count	k-mer lengths						
	16	17	18	19	20	21	22
	Proportion of total k-mers in %						
One	91.68	95.13	96.47	97.02	97.29	97.46	97.5
$\leq$100	8.30	4.85	3.51	2.96	2.69	2.52	2.40
$\leq$1000	0.022	0.020	0.019	0.018	0.017	0.016	0.015
$>$1000	.0015	.0013	.0012	.0011	.001	.0009	.0008

Verification-Aware filtration using Pigeonhole Principle

For mapping reads with an edit distance (or permissible error) of δ using the pigeonhole principle, a read is divided into $(\delta + 1)$ non-overlapping *k-mers*. CORAL measures the maximum possible k for equal length non-overlapping *k-mers*, as $k = \lfloor \frac{n}{\delta+1} \rfloor$, where n is the read length. For example, given $n = 100$ and $\delta = 5$, $k = \lfloor \frac{100}{6} \rfloor = 16$ and for $n = 150$ and $\delta = 7$, $k = \lfloor \frac{150}{8} \rfloor = 18$. We limit ourselves to $n = 100$ to 150 and $\delta = 0$ to 8, however, in theory the method can be generalised to any parametric value. More on this will be discussed in Section 3.5.

Upon obtaining k, we can calculate the number of extra or spare bases (*eb*) that remain after securing $(\delta + 1)$ non-overlapping *k-mers*. For example, $eb = n - k \times (\delta + 1) = 4$ for $n = 100$, $\delta = 5$, and $eb = 6$ for $n = 150$ and $\delta = 7$. These extra bases can be used to extend the length of any *k-mer*, as discussed in Fig. 3.4, to minimize the number of candidate locations depending on the given threshold. It should be noted that the number of non-overlapping *k-mers* remain intact despite extensions, thus, extensions do not affecting sensitivity.

Table 3.2 presents the characterization of chromosome 2 on the basis of the number of occurrences of *k-mers* for different k. We found that over 90% *k-mers* occur only once, however, a consistent number of *k-mers* can be encountered more than 1000 times. Thus, these *k-mers* will produce over 1000 candidate locations, if encountered. The maximum value ranged up to $1,644,958$ for a few k-mers. Intuitively, it can be understood that such high frequency *k-mers* should be avoided to reduce the average number of candidate

locations per read. In CORAL, the threshold is set as 1000 as *k-mers* that produce over 1000 occurrences are extended. This value is heuristic and can be changed depending on user requirements. The impact of verification-aware *k-mer* length adaptation is discussed in Section 3.5.

3.3.4 Implementation of Myers Bitvector Algorithm

The `C` code presented below represents the implementation of the Myer's bit-vector algorithm borrowed from the source code of CORAL kernel to discuss implementation detail and approximation used in CORAL. CORAL verifies a maximum of 1000 candidate locations per *k-mer*, as seen in line 2. It reports the *first-n* mapping locations after verification, where *n* in *first-n* is user defined before execution and is denoted by `CAND_LOC_PER_READ`. The parameter *n* in *first-n* should not be confused with the read length *n* used elsewhere in the thesis. The `no_of_locations` parameter keeps a count of number of locations that were mapped successfully, shown in line 56. It is used to skip the verification of all remaining candidate locations once the condition in line 6 is true, which indicates that the desired number of mappings for the read are found already. Although, banded Myer's bit-vector algorithm is a matured technology used in most of the recent read mappers, coding it can be challenging considering its correct implementation is critical to the accuracy of the read mapper. The implementation presented, here, is adapted from Hyyrö's [98] banded Myer's algorithm with the help of its implementation in RazerS3 [2]. For more details, interested readers can refer to these articles.

```
//EXCERPT FROM THE SOURCE CODE PRESENTING MYER'S BIT-VECTOR ALGORITHM
occurences = (occurences > 1000)?1000:occurences;
for(j = 0; j < occurences; j++)
{
    verif_start_pos_in_genome = SA[sa_start_pos + j] - x - ERROR;
    if(no_of_locations >= CAND_LOC_PER_READ)
    {       continue;       }
    score = c;   // Reseting of score
    edit_dist = ERROR+1;
    VP = ~0; VN = 0;
    B[0] = B_F[0]; B[1] = B_F[1];
    B[2] = B_F[2];B[3] = B_F[3];B[4] = B_F[4];
    for(k = 0; k < band_len; k++)
    { // verifying for n+2e length (READ_LENGTH + ERROR + ERROR)
        B[0] = B[0] >> 1;
        B[1] = B[1] >> 1;
        B[2] = B[2] >> 1;
        B[3] = B[3] >> 1;
        B[4] = B[4] >> 1;
        if(k + c < READ_LENGTH)
        {       B[RF[k+c]] = B[RF[k+c]] | MASK;     }

        X = B[genome[k + verif_start_pos_in_genome]] | VN;
        D0 = ((VP + (X & VP)) ^ VP) | X;
        HN = VP & D0;
        HP = VN | ~(VP | D0);
        X = D0 >> 1;
        VN = X & HP;
        VP = HN | ~(X | HP);

        if(k < (READ_LENGTH-c))
        {
            score = score + 1 - ((D0 >> (W-1)) & 1);
        }
        else
        {
            s = constant1 - k;
            //s = (W-2) - (k - (READ_LENGTH - c + 1));
            score = score + ((HP >> s) & 1);
            score = score - ((HN >> s) & 1);
        }
        if(score < edit_dist && (k >= (READ_LENGTH-c)))
        {
            edit_dist = score;
            temp_location = k + verif_start_pos_in_genome;
        }
    }
    if(edit_dist <= ERROR)
    {
        endpos_for_mapped_reads[gid + no_of_locations]
        = temp_location + 1;

        genomic_strand_and_ED_for_mapped_reads[gid + no_of_locations]
        = 128 + edit_dist;

        no_of_locations = no_of_locations + 1;
    }
}
```

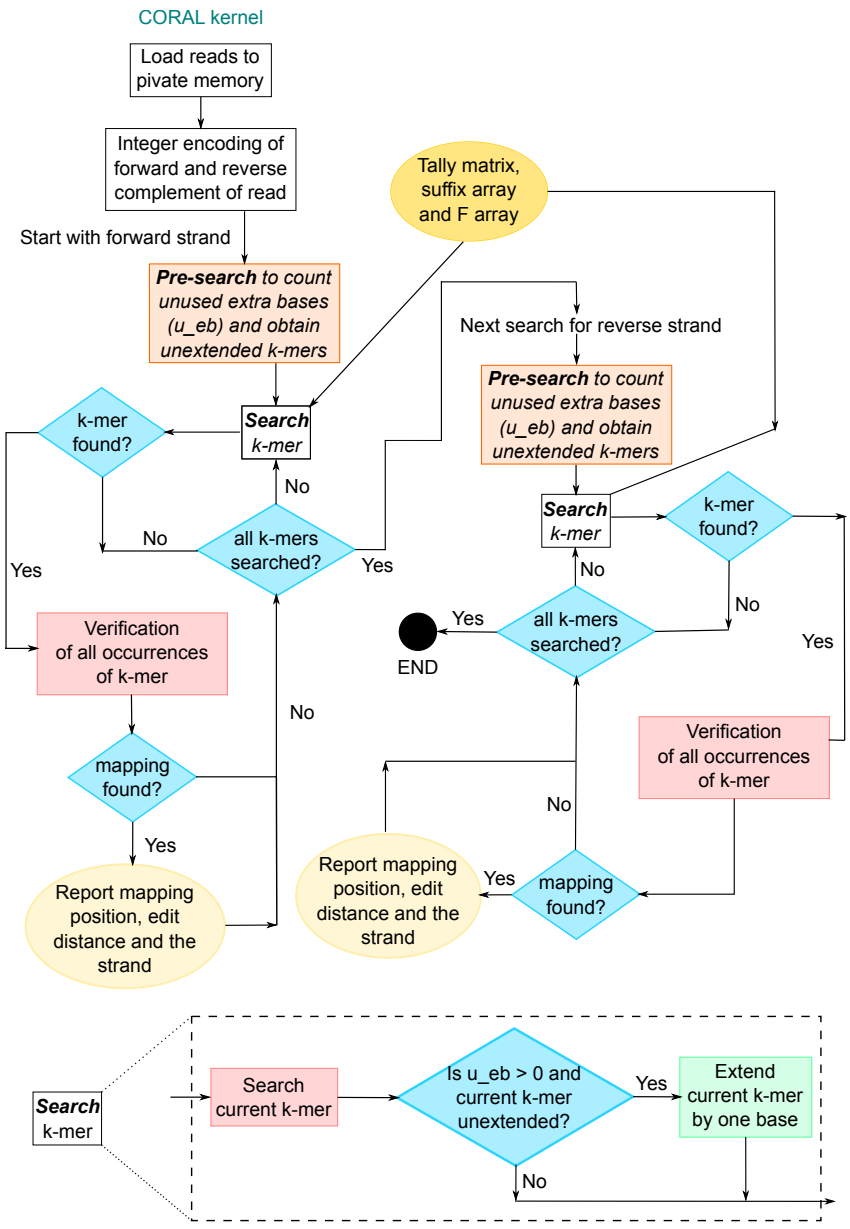

Figure 3.5: Algorithm for the CORAL kernel.

3.3.5 Kernel Algorithm: Search and Verification

Input to the kernel are: Integer encoded reference genome, reads, suffix array, tally matrix, modified F array and other constants. The integer reference encoded genome, suffix array, tally matrix and modified F array are generated by the preprocessing step.

The constants include read length, minimum *k-mer* length, number of *k-mers*, δ, *eb* and mapping locations to be reported. Except for number of *k-mers* and *eb* the rest constants are user-given parameters during execution. Fig. 3.5 visualizes the algorithmic procedure followed by the kernel. It starts with loading modified F array and read to the private memory followed by integer coding and storage of forward and reverse strand of the read. The tally matrix, reference genome and suffix array are kept in global memory as they are too huge for private memory. Integer encoding of reads is performed so that the encoded value can be directly used as index to access the elements of a particular column and row of tally matrix. It is performed using the following scheme: $\{(A : 0), (C : 1), (G : 2), (T : 3), (Z : 4)\}$. After preprocessing of reads, we perform filtration and verification of forward strand followed by reverse strand of the read.

Filtration is divided into two stages: *Pre-search* and *Search*, as highlighted in Fig. 3.5. *Pre-search* is a preliminary search of all the *k-mers* using our proposed dynamic extension approach to identify the unused extra bases, *u_eb*, if any, and record the corresponding unextended *k-mer*. In the context of *Pre-search* $u\_eb = eb > 0$, implying that the number of candidate locations reported by most *k-mers* were within the threshold of 1000, thus, the unused bases remain which could be used otherwise to extend the *k-mers* even though they meet threshold. *Pre-search*, thus, gives an estimate of unused bases that can be used to further extend some of the *k-mers* to further reduce the overall number of candidate locations reported by all the *k-mers*. For example, given $n = 100$, $\delta = 5$ and $eb = 4$, if each of 6 *k-mers* result in < 1000 candidate locations, then, $u\_eb = 4$. The goal, here, is to consume the entire read in filtration step irrespective of the *k-mer* lengths. Therefore, to utilize all the *u_eb*, we extend 4 out of 6 *k-mers* by one base each, increasing *k* from 16 to 17. The information on *u_eb* is used during the *Search* stage, each unextended *k-mer* is elongated by one base until $u\_eb = 0$. The choice of *k-mers* is serial starting from the first. This ensures that all the extra bases are utilized to minimize the number of candidate locations. Both *Pre-search* and *Search* start from the end of the read and depending on the number of *eb*, *u_eb* and candidate locations, it extends the *k-mer* till the number of candidate locations are < 1000 or all the spare bases are exhausted. As *Search* proceeds, all the candidate locations of each *k-mer* are verified in-situ and the mappings found are

reported. The same procedure is followed for the reverse strand.

3.4 Experimental Results

The host program of CORAL is written in `Python` and the kernel is in `C`. We use `PyOpenCL` rather than conventional `C`-based OpenCL because scripting languages, such as `Python`, enables fast modifications and prototyping and is more programmer friendly. We use OpenCL 1.2 standard to compile the kernel. We choose `Python` because it considerably eases string operations and manipulation, especially, the outlier operations which do not affect the mapping directly.

3.4.1 Experimental setup

We use both real and simulated reads to compare CORAL with RazerS3, Yara, Hobbes3 and FEM from the *all-mapper* category and BWA-MEM and GEM from the *best-mapper* category. We use a total of **6 million simulated reads** and **2 million real reads**. Wherever possible, only the mapping times and accuracy have been compared. We have mapped both simulated and real single-end reads to chromosome (chr) 2 and 21 of the human genome. The latest version of Mason [29] (`mason2-2.0.9`) is used to produce simulated reads. We use 12 sets of 500,000 reads each, 6 of them are derived from chr 2 and other 6 from chr 21. Out of the 6 sets, three of them have reads of length 100 and the other three have reads of length 150. The three sets, with read length of 100, are segregated based on edit distances with which they are sequenced from the chromosomes viz. 3 or less, 4 or less and 5 or less. Similarly, the reads of length 150 are segregated based on edit distance viz. 5 or less, 6 or less and 7 or less. Thus, resulting in a total of **6 million simulated reads** to be mapped by all the mappers. The chromosomes used in this chapter are from the human genome version GRCh38/hg38, dated Dec. 2013, and were downloaded from the UCSC genome browser [136]. We used **1 million (M) real reads** each from NCBI ERR012100_1 and SRR826460_1 of length 100 and 150, respectively. We run all the mapping tools including CORAL on two separate systems with the following configurations:

System 1: Intel Core i5-6600 CPU @ 3.30GHz, 64GB RAM

System 2: Intel Core i7-2600 CPU @ 3.40GHz, 16GB RAM + 2 $\times$ GeForce GTX 590, 1.5 GB RAM

OpenCL computing framework imposes the following two restrictions:

a) OpenCL 1.2 standard does not permit dynamic memory allocation. Because of this, CORAL requires the number of mapping locations per read to be mentioned beforehand, in order to allocate sufficient memory for each read to store the mapping locations and strands. Thus, it reports a maximum of *first-n* mapping locations per read as informed in Table 3.1.

b) It does not permit allocation of more than $(1/4)^{th}$ of the RAM capacity to a single variable. Example, with 16 GB RAM no variable can have more than 4GB of memory allocated. It limits both the size of the data structure to be stored and the number of mapping locations desired per read.

To elaborate further on (a), if a read matches only at few locations, it will still require to be allotted sufficient space for the given number of mapping locations desired per read. As we have limited RAM on system 2, especially, in the GPUs, the number of mapping locations per read must be assigned accordingly to ensure we do not run out of memory resource. As system 1 has large RAM capacity, we allot 3500 mapping locations per read to show the accuracy of CORAL and allot 100 mapping locations per read on system 2 to demonstrate speedups obtained by using multiple devices. The preprocessed tally matrix size depends on the length of the chromosome i.e. the size for chr 2 is 3.9GB which is much larger than that of 747.4 MB for chr 21. As GPUs have limited RAM size of 1.5 GB, the data structure for chr2 cannot be loaded on them. Hence, to demonstrate the implementation on multiple devices simultaneously, we use smaller chr21 to map the real and simulated reads. In summary, we present the results for the following combinations:

CORAL on System 1: Both real and simulated reads are mapped to chr2 and chr21 with 3500 mapping locations per read for different number of errors, using only the CPU.

CORAL-cpu on System 2: Both real and simulated reads are mapped to chr2 and

chr21 with 100 mapping locations per read for different number of errors, using only the CPU.

CORAL-all on System 2: Both real and simulated reads are mapped to chr21, only, with 100 mapping locations per read for different number of errors, using CPU along with the GPUs.

Estimating accuracy with respect to simulated reads

For the simulated reads, the SAM file obtained from Mason is used as the gold standard for measuring mapping accuracy. On system 1, CORAL, RazerS3, Hobbes3 and GEM report up to 3500 mapping locations per read while BWA-MEM, FEM and Yara report all the mapping locations, since they do not provide the facility to report fixed number of mappings. On system 2, all the mappers are configured to either report up to 100 mapping locations per read, wherever possible, or all the mapping locations. Simulated reads originate from a known position reported in the SAM file obtained from Mason. Hence, to determine the mapping accuracy, the output files from the mappers are parsed and searched for original mapping location, strand and edit distance. If any of mapping locations reported by the mappers match to that of the gold standard for a particular read within the given edit distance $\leq \delta$, we record an accurate mapping. This procedure is followed for all the simulated reads and all the mappers under consideration. While running comparison with the gold standard, we allow for a threshold, $\tau = \pm 10$ bases with respect to the original location. Irrespective of the chosen τ, the criteria for measuring a match remains uniform for all the mappers.

Estimating accuracy with respect to real reads

A similar approach is followed for real reads, however, the SAM file obtained from RazerS3 is used as the gold standard. We use RazerS3 as it has been used in Hobbes3, FEM and Yara to build the gold standard due to its high accuracy and *all-mapper* capability. On System 1, we use RazerS3 to produce SAM file with up to 1000 mapping locations per read and all the other mappers (including CORAL) are configured to report 3500 mapping locations per read, if possible, or all the mapping locations. Following

that, we identify if all the, up to 1000, mapping locations per read reported in the gold standard are present in the output of other mappers. In comparison with section 3.4.1, where it is sufficient to find a single known location of origin of a simulated read, here, for the real reads all the locations reported (up to 1000) by the gold standard is compared with 3500 mapping locations of other mappers, thus, making the evaluation criteria relatively stringent.

On System 2, we do the opposite. We configure RazerS3 to produce up to 1000 mapping locations per read and configure the mappers to map up to 100 mapping locations per read. Here, we measure accuracy by identifying if all the reads mapped by the gold standard i.e. RazerS3, have been reported by other mappers with at least one matching mapping location, strand and edit distance. We limit the number of mapping locations to 100 because of the limitations on RAM capacity of System 2. Using the aforementioned configurations for evaluation of accuracy, we present results similar to the benchmarking method used in Rabema [137] i.e. the *all* and *all-best* scenario on System 1, and *any-best* scenario on System 2.

Configurations of read mappers

RazerS3: We used the latest version available viz. `razers3-3.5.8`. Pigeonhole filter was used with thread count of 16 for different percentage identity or error rates and number of mapping locations. The following provides an example of the command line parameters used:

```
razers -fl pigeonhole -tc 16 -i 95 -rr 100 -m 3500 -v
-o OUTPUT.sam chr2.fa INPUT.fq
```

Yara: The latest version `1.0.2` was used with 16 thread in `full` sensitivity mode.

```
yara_mapper chr2.index INPUT.fq -v -e 4 -y full -t 16 -o OUTPUT.sam
```

Hobbes3: We used latest version `3.0` with 16 threads and varying number of maximum number of mapping locations and errors per read.

```
hobbes -sref chr2.fa -i chr2_hobbes3_index.hix -k 3500 --indel -q INPUT.fq
-v 5 -p 16 --mapout OUTPUT.sam
```

FEM: We used latest FEM version available dated 03/13/2018. For index construction, we used window size of 12 and step size of 4 (e.g. `FEM index 12 4 chr2.fa`) and for

mapping we used 16 threads with edit distance configuration. FEM, by default, reports all the mapping positions. The following provides an example of the command line parameters used:

```
FEM align -t 16 -f "vl" --ref chr2.fa --read INPUT.fq -o OUTPUT.sam -e 5
```

GEM: We used the latest version `3`. For simulated reads, we run GEM in `sensitive` mapping mode, however, for real reads we used `fast` mapping mode, as `sensitive` took hours to produce results. We used 16 threads with varying number of mapping locations i.e. 100 or 3500, and error rates viz. 3 to 7. The following provides an example of the command line parameters used:

```
gem-mapper --index chr2.gem -v -t 16 -M 3500 --mapping-mode sensitive
-i INPUT.fq -o OUTPUT5.sam -e 0.05
```

BWA-MEM: We used latest BWA version `0.7.17`. We configured BWA-MEM to find all mapping locations with a thread count of 16. BWA-MEM is configured to skip *k-mers*(or seeds) with more than 500 occurrences by default. We increased it to 1000 similar to the threshold value used for CORAL. BWA-MEM does not permit specifying edit distance for read mapping unlike BWA-aln, hence, for real reads we could not obtain mapping times for different edit distance values. The following provides an example of the command line parameters used:

```
bwa mem -t 16 -c 1000 -v 3 -a chr2.fa INPUT.fq > OUTPUT.sam
```

3.4.2 Results

Simulated reads mapped to chr 2

Table 3.3 presents the results of mapping three sets of 500,000 simulated reads to chr 2 on the CPU of System 1 and 2. The three sets of reads have different maximum error or edit distances viz. 3, 4 and 5, respectively. We can observe that CORAL is $5-16\times$ faster than RazerS3 and maps over 99% of reads, showing comparable accuracy. The runtime of CORAL is better than Hobbes3 for low error rates and comparable for higher error rates, for example, $n = 100, \delta = 5$ and $n = 150, \delta = 7$. On System 2, as the number of mapping locations is small, Hobbes3 performs better than CORAL for high error rates. The accuracy of both Hobbes3 and CORAL are comparable and are over 99%. CORAL

outperforms Yara and BWA-MEM in all the cases, with up to 2.27× and 4.84× speed-up, respectively. CORAL beats GEM in runtime for $n = 100$ and accuracy, however, it lags for $n = 150$. FEM runtimes are faster than that of CORAL but it maps less than 40% of reads. GEM performs better as it is a *best-mapper* and designed to produce fewer accurate solutions. From the accuracy point of view, CORAL performs comparable to RazerS3, Hobbes3 and BWA-MEM and outperforms Yara, FEM and GEM in all cases. With regards to mapping time CORAL is better than Yara, BWA-MEM and RazerS3, and comparable to Hobbes3 and GEM. It, however, underperforms with respect to FEM, which is fast but the accuracy is low.

Simulated reads mapped to chr 21

Table 3.4 presents the results of mapping three sets of 500,000 simulated reads to chr 21 on the CPU of system 1 and 2 and CPU+GPU combination of system 2. As mentioned in previous section, the three sets of simulated reads have maximum error of 3, 4 and 5. Compared to RazerS3, CORAL is $2 - 8\times$ faster and maps over 99% of the reads accurately. CORAL outperforms Hobbes3 in all cases except for $n = 100$ and $e = 5$. It, considerably, outperforms FEM on mapping accuracy. From Table 3.4, we can observe that CORAL-all, which distributes a portion of the workload on Nvidia GPUs, results in up to 2× speedup with the same accuracy. CORAL outperforms Yara and BWA-MEM in all the cases on System 1, with up to 2.08× and 7.96× speed-up, respectively. For CORAL-all, we equally distributed 256,000 out of 500,000 reads on two Nvidia devices and the remaining 244,000 on the CPU to obtain speedups. From the experiments, we conclude that CORAL-cpu outperforms RazerS3, Hobbes3, GEM, Yara and BWA-MEM in most of the cases on either mapping time or accuracy or both and if not produce comparable results. Similar to Section 3.4.2, FEM is faster than CORAL but lags considerably in accuracy. CORAL-all which uses multiple devices provide an additional speedup of up to 2×, unlike other mappers who are optimized to operate only on the CPU.

Table 3.3: The results of mapping three sets of 500,000 simulated reads, with different maximum edit distances viz. 3,4 and 5, to chromosome (chr) 2 on the CPU. RazerS3, Hobbes3, GEM and CORAL-cpu reported up to 3500 mapping per read on System 1 and 100 mapping locations per read on System 2. FEM, Yara, however, by default report all the mapping positions and BWA-MEM was configured to report all mapping positions. T and A are abbreviations for time and accuracy and have been reported in seconds and proportional percentage of accurate mapping against the total number of reads.

Chromosome 2	Read length	100						150					
	Error	3		4		5		5		6		7	
	Time/Accuracy	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)
System 1: Intel Core i5-6600 CPU@3.30GHz, 64GB RAM	RazerS3	169.2	99.99	215.8	99.99	265.9	100	190.7	99.99	224.5	100	294.3	99.99
	Hobber3	16.18	99.99	19.52	99.99	28.14	100	35.47	99.99	36.55	100	41.24	99.99
	FEM	3.30	41.18	4.58	41.10	6.51	41.17	5.71	37.32	7.49	37.47	9.67	37.48
	Yara	13.35	97.92	33.85	97.91	41.42	97.85	38.73	98.60	47.46	98.59	102.44	98.57
	BWA-MEM	53.30	99.71	61.53	99.32	66.96	98.65	74.35	99.61	80.46	99.36	85.39	99.03
	GEM	14	97.90	21	97.89	48	97.81	13	98.57	18	98.57	25	98.54
	CORAL-cpu	**11.89**	**99.77**	**19.35**	**99.71**	**30.99**	**99.75**	**23.24**	**99.91**	**34.50**	**99.91**	**45.04**	**99.92**
System 2: Intel Core i7-2600 CPU@3.40GHz, 16GB RAM + 2 × GeForce GTX 590, 1.5 GB	RazerS3	125.5	99.78	165.2	99.77	193.7	99.77	131.8	99.93	158.0	99.93	218.4	99.93
	Hobber3	11.70	99.77	10.58	99.76	10	99.73	32.54	99.90	28.80	99.88	24.86	99.86
	FEM	1.97	41.18	2.63	41.10	5.02	41.17	2.87	37.32	3.49	37.47	5.37	37.48
	Yara	8.87	97.92	25.04	97.91	28.92	97.85	28.34	98.60	33.07	98.59	72.1	98.57
	BWA-MEM	39.43	99.71	45.88	99.31	50.79	98.65	56.06	99.61	61.17	99.36	65.97	99.03
	GEM	7	97.90	12	97.89	27	97.81	7	98.57	11	98.57	14	98.54
	CORAL-cpu	**8.14**	**99.48**	**15.67**	**99.31**	**27.71**	**99.24**	**16.89**	**99.67**	**29.8**	**99.60**	**40.13**	**99.50**

Table 3.4: The results of mapping three sets of 500,000 simulated reads, with different maximum edit distances viz. 3,4 and 5, to chromosome (chr) 21. RazerS3, Hobbes3, GEM and CORAL-cpu reported up to 3500 mapping per read on System 1 and 100 mapping locations per read on System 2. CORAL-all, also, produce 100 mapping locations per read but executes on CPU and both the GPUs, simultaneously. FEM, Yara, however, by default report all the mapping positions and BWA-MEM was configured to report all mapping positions. T and A are abbreviations for time and accuracy and have been reported in seconds and proportional percentage of accurate mapping against the total number of reads.

Chromosome 21	Read length	100						150					
	Error	3		4		5		5		6		7	
	Time/Accuracy	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)
System 1: Intel Core i5-6600 CPU@3.30GHz, 64GB RAM	RazerS3	29.95	99.99	48.30	99.99	49.13	99.99	35.12	99.99	42.43	99.99	56.31	99.99
	Hobber3	13.42	99.99	14	99.99	17.19	99.99	31.75	99.99	30.73	99.99	32.13	99.99
	FEM	1.535	39.45	1.79	39.45	2.31	39.51	2.14	36.09	2.30	36.14	2.82	36.25
	Yara	10.58	92.71	18.57	92.64	22.62	92.58	27.75	94.25	32.07	94.17	41.09	94.13
	BWA-MEM	60.14	99.72	74.35	99.37	72.08	98.78	80.69	99.64	86.64	99.37	92.37	99.04
	GEM	10	92.69	13	92.59	20	92.54	11	94.12	15	94.10	17	94.04
	CORAL-cpu	**7.55**	**99.99**	**11.87**	**99.98**	**19.31**	**99.98**	**13.31**	**99.99**	**20.78**	**99.99**	**27.35**	**99.99**
System 2: Intel Core i7-2600 CPU@3.40GHz, 16GB RAM + 2 × GeForce GTX 590, 1.5 GB	RazerS3	23.39	99.74	31.42	99.72	39.21	99.70	26.52	99.98	33.39	99.98	46.53	99.98
	Hobber3	11.12	99.60	9.37	99.54	7.92	99.44	31.91	99.89	27.55	99.86	23.15	99.81
	FEM	1.16	39.45	1.29	39.45	1.59	39.51	1.59	36.09	1.73	36.14	1.89	36.25
	Yara	7.26	92.71	14.41	92.64	16.67	92.58	20.67	94.25	23.19	94.17	29.19	94.13
	BWA-MEM	43.27	99.71	51.16	99.37	57.55	98.78	73.98	99.64	78.05	99.37	85.05	99.04
	GEM	5	92.69	7	92.59	12	92.54	7	94.12	9	94.10	11	94.04
	CORAL-cpu	**4.42**	**99.43**	**8.12**	**99.25**	**14.85**	**99.10**	**9.13**	**99.70**	**15.01**	**99.53**	**21.88**	**99.30**
	CORAL-all	**3.09**	**99.43**	**5.40**	**99.25**	**8.55**	**99.10**	**6.15**	**99.70**	**9.20**	**99.53**	**13.04**	**99.30**

Table 3.5: The results of mapping 1M real reads to chromosome (chr) 2 on the CPU. RazerS3, Hobbes3, GEM and CORAL-cpu reported up to 3500 mapping per read on System 1 and 100 mapping locations per read on System 2. FEM, Yara, however, by default report all the mapping positions and BWA-MEM was configured to report all mapping positions. T and A are abbreviations for time and accuracy and have been reported in seconds and proportional percentage of accurate mapping against the total number of reads.

Chromosome 2	Read length	100						150					
	Error	3		4		5		5		6		7	
	Time/Accuracy	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)
System 1: Intel i5-6600 CPU@3.3GHz, 64GB RAM	RazerS3	229.7	100	334.4	100	443.5	100	268	100	388.6	100	631.6	100
	Hobber3	38.24	100	38.67	100	52.12	99.99	61.63	100	56.89	100	58.81	100
	FEM	4.80	0.65	6.87	0.44	12.14	0.32	5.23	0.20	6.73	0.83	9.33	0.11
	Yara	22.52	1.88	96.70	1.32	118.34	1.01	178	1.18	873.84	0.96	1971.8	0.83
	BWA-MEM	T(s) - 120.5		A(%) - 14.24				T(s) - 234.9		A(%) - 11.45			
	GEM	25	1.78	25	1.22	29	0.92	58	1.03	57	0.15	55	0.70
	CORAL-cpu	**31.19**	**96.05**	**52.75**	**95.77**	**89.09**	**94.04**	**68.57**	**99.89**	**125.6**	**99.5**	**187.88**	**97.72**
System 2: Intel i7-2600 CPU@3.4GHz, 16GB RAM + 2 × GeForce GTX590, 1.5GB	RazerS3	160.1	100	247.5	100	363	100	185.9	100	307.2	100	560.1	100
	Hobber3	24.92	100	24.37	100	27.85	100	63.49	100	56.36	100	52.60	100
	FEM	3.03	31.8	4.21	29.61	7.61	27.41	3.36	16.13	4.23	14.41	5.86	12.73
	Yara	13.44	100	74.17	100	87.15	99.99	136.5	100	824.9	100	1877	100
	BWA-MEM	T(s) - 85.44		A(%) - 97.49				T(s) - 161.6		A(%) - 93.51			
	GEM	22	94.54	22	92.97	21	91.33	55	86.80	52	86.93	52	84.34
	CORAL-cpu	**22.18**	**99.91**	**44.45**	**99.75**	**81.53**	**99.66**	**69.3**	**99.89**	**129.6**	**99.79**	**198.6**	**99.71**

Table 3.6: The results of mapping 1M real reads to chromosome (chr) 21. RazerS3, Hobbes3, GEM and CORAL-cpu reported up to 3500 mapping per read on System 1 and 100 mapping locations per read on System 2. CORAL-all, also, produce 100 mapping locations per read but executes on CPU and both the GPUs, simultaneously. FEM, Yara, however, by default report all the mapping positions and BWA-MEM was configured to report all mapping positions. T and A are abbreviations for time and accuracy and have been reported in seconds and proportional percentage of accurate mapping against the total number of reads.

Chromosome 21	Read length	100						150					
	Error	3		4		5		5		6		7	
	Time/Accuracy	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)
System 1: Intel Core i5-6600 CPU@3.30GHz, 64GB RAM	RazerS3	39.33	100	58.56	100	77.85	100	43.86	100	62.69	100	100.5	100
	Hobber3	24.57	100	24.1	100	22.22	100	53.31	100	45.75	100	38.31	100
	FEM	2.34	0.459	2.58	0.373	3.50	0.313	2.06	0.074	2.33	0.077	2.81	0.061
	Yara	12.58	100	26.65	100	36.18	100	52.48	100	135.9	100	357.58	100
	BWA-MEM	T(s) - 127.39		A(%) - 22.96				T(s) - 219.15		A(%) - 28.64			
	GEM	24	2.03	25	1.81	24	1.66	61	4.65	60	3.97	59	3.41
	CORAL-cpu	**11.68**	**99.93**	**21.39**	**99.87**	**42.14**	**99.91**	**21.52**	**100**	**45.33**	**100**	**75.80**	**100**
System 2: Intel Core i7-2600 CPU@3.0GHz, 16GB RAM + 2 × GeForce GTX 590, 1.5GB	RazerS3	26.79	100	42.37	100	64.52	100	30	100	49.10	100	88.92	100
	Hobber3	20.31	100	16.87	100	14.44	100	58.36	100	49.88	100	40.7	100
	FEM	2.50	16.45	2.15	14.52	2.27	12.72	2.27	1.44	2.12	1.75	3.04	1.59
	Yara	6.09	100	18.59	100	24.34	100	35.77	100	110.7	100	309.7	100
	BWA-MEM	T(s) - 184.38		A(%) - 97.17				T(s) - 359.56		A(%) - 95.09			
	GEM	23	93.66	22	92.04	22	89.97	56	90.20	54	91.35	54	89.07
	CORAL-cpu	**7.72**	**100**	**18.33**	**99.98**	**39.79**	**99.99**	**20.18**	**100**	**45.38**	**100**	**81.42**	**100**
	CORAL-all	**5.36**	**100**	**12.41**	**99.98**	**27.95**	**99.99**	**12.5**	**100**	**27.41**	**100**	**49.6**	**100**

Real reads mapped to chr 2

System 1: Table 3.5 presents the results of mapping 1 M real reads on chr2, from two different databases with different read lengths. RazerS3, Hobbes3, GEM and CORAL-cpu reported up to 3500 mapping per read. We can observe that CORAL is $3-7\times$ faster than RazerS3 and accurately maps over 94% of reads. It is also evident that it is considerably better than FEM and GEM in accuracy of mapping reads. We could not run GEM in the sensitive mode for real reads as it was taking very long runs. For $\delta = 3$, CORAL outperforms Hobbes3, however, it lags behind for higher error rates. On the contrary, CORAL outperforms Yara on all cases, especially, for higher error rates, leaving single case where $\delta = 3$ and $n = 100$, and beats it on accuracy in all cases. CORAL beats BWA-MEM on all parameters and cases.

System 2: On System 2, all the mappers are configured to map up to 100 positions per read. Yet again it can be seen that CORAL outperforms Yara and BWA-MEM in all the cases on mapping times. On accuracy, leaving for a few cases with respect to Yara, with a marginal $< 0.4\%$ difference, CORAL outperforms both Yara and BWA-MEM. The accuracy for Yara, BWA-MEM, FEM and GEM are higher on System 2 due to different comparison criteria used, as discussed in Section 3.4.1. Here, we measure *any-best* accuracy of Rabema. For all cases, CORAL, considerably, outperforms GEM and FEM in accuracy.

Real reads mapped to chr 21

System 1: Table 3.6 presents the results of mapping 1 M real reads, from two different databases with different read lengths, on chr 21. On System 1, all mappers produce 3500 mapping locations per read, except, the RazerS3 which serves as the gold standard. We can observe that CORAL is $2-4\times$ faster than RazerS3 in all cases. Leaving for $n = 100, \delta = 5$, CORAL is up to $5\times$ faster than Yara with similar accuracy. It outperforms BWA-MEM on all accounts. CORAL mapping times are better than Hobbes3 and GEM for lower error rates except for $n = 100, \delta = 5$ and $n = 150, \delta = 7$. FEM and GEM reportedly mapped only a small number of reads, hence, CORAL mapping accuracy

supersedes them.

System 2: On System 2, all mappers report up to 100 mapping locations per read, except, the RazerS3 which serves as the gold standard. The results, here, follow similar trend as explained above. However, we can see that mapping times can be halved if all the available resources are used with judicious workload distribution. CORAL-all executes on CPU and both the GPUs producing up to 2× speedup. For $n = 150$, we mapped 368,000 reads on GPUs and remaining 632,000 reads on the CPU. For $n = 100$, we mapped 340,000 reads on the GPU and remaining 660,000 reads on the CPU. These numbers were chosen depending on the memory capacity of GPUs and kernel requirements. CORAL-cpu and CORAL-all outperform RazerS3 and BWA-MEM in all the cases. Except for $n = 100, \delta = 5$ and $n = 150, \delta = 7$, it outperforms Hobbes3, GEM and Yara in all other cases. In case of FEM, the mapping accuracy were found to be very low despite successive experiments.

Evaluation of Verification-Aware Filtration

Fig. 3.6 shows the average number of verifications performed per read using three filtration schemes: non verification-aware (NVA), verification-aware (VA) and verification-aware with approximations (VA+A). In NVA scheme, we fix the lengths of *k-mer* and calculate the total number of verifications required for all the reads. For example, given $n = 100, \delta = 5$ and $n = 150, \delta = 7$, the *k-mer* lengths are $(17, 17, 17, 17, 16, 16)$ and $(19, 19, 19, 19, 19, 19, 18, 18)$, respectively. In VA scheme, CORAL dynamically determines the lengths of *k-mer* by extending them depending on the number of verifications encountered. VA+A scheme is similar to VA with an additional condition that limits the maximum allowed candidate locations per *k-mer* to 1000. Experiments performed on both the real data sets show that the number of verifications reduces significantly (up to 3.67×) across the filtration schemes.

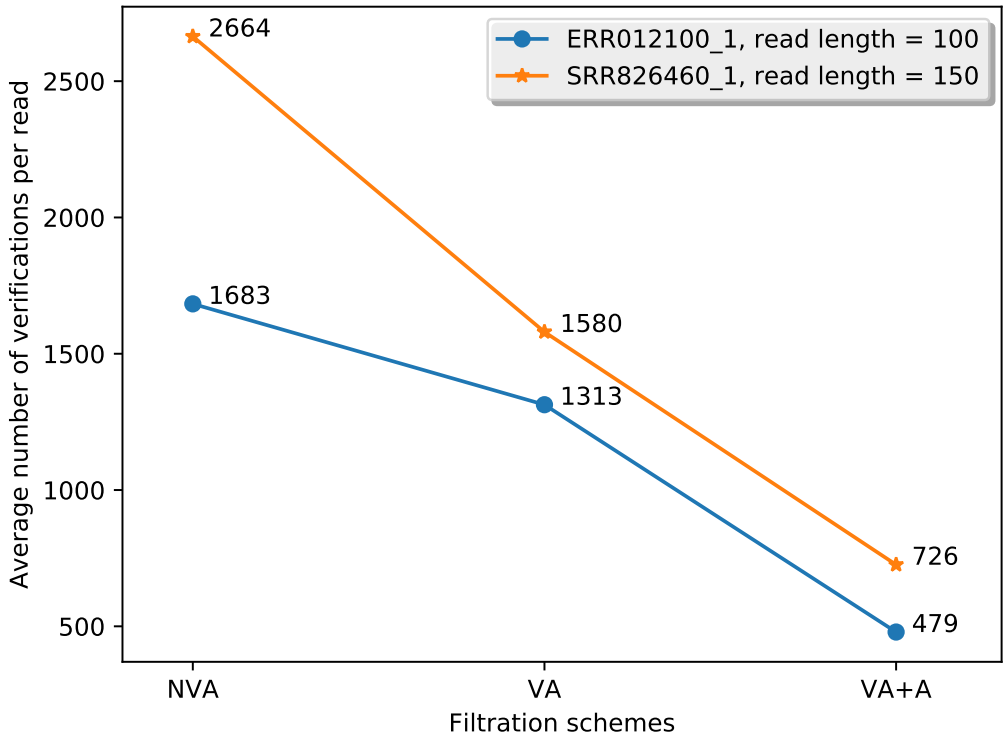

Figure 3.6: Average number of verifications per read using different filtration schemes for real data sets, viz. ERR012100_1 ($n = 100, \delta = 5$) and SRR826460_1 ($n = 150, \delta = 7$), on chr2. NVA - non verification-aware, VA - verification-aware and VA+A - verification-aware along with approximation. The approximation used, here, limits the maximum number of verifications per *k-mer* to 1000.

3.5 Discussion

From Section 3.4.2, we see that, even though GEM uses a similar approach of *k-mer* length variation on FM-Index, CORAL outperforms GEM on either mapping times, accuracy or both for different read lengths, errors and datasets. For real reads, GEM could not produce any output in the sensitive mode, even after allowing longer runtimes. Compared to GEM, CORAL is an *all-mapper* with flexibility to work on heterogeneous systems. We observe that Hobbes3 outperforms CORAL in few cases, like, with longer chromosome, chr2, and high error rates, $\delta = 5, 7$. One of the major reasons is that the

latest mappers use Streaming SIMD Extensions (SSE) instruction set. SSE instruction set utilizes 128-bit registers to accelerate computations. It enables loading of multiple bit vectors into a machine word, therefore, accelerating the banded Myers bit-vector algorithm, which is a major bottleneck in read mappers. OpenCL abstracts different hardware manifestations of parallel architecture including SIMD, but does not support the SSE instructions, yet, for portability on wide-spectrum of devices. Difference in performance between CORAL and Hobbes3 is, also, due to *k-mer* selection criteria. CORAL selects the maximum possible length for each *k-mer* with an objective to reduce the number of candidate location, of a particular *k-mer*, by increasing its length using excess bases, if available. While Hobbes3 uses a dynamic programming based filtration scheme.

The state-of-the-art mappers have focused on algorithmic innovations and software optimizations targeting only the CPU. To use them on different hardware, such as GPU or FPGAs, will require to be either rewritten or tailored. K. Reinert et al [36] present a review of the existing methods and algorithms, and predict in their concluding remarks that further improvements in the assembling time will result from accelerators and co-processors. S. Aluru and N. Jammula [30] present a review on hardware accelerators for genome assembly on FPGAs and GPUs. Darwin [138] is a FPGA based co-processor for whole genome alignment aiming at aligning genomes of two or more species. The authors have reported significant improvements in performance/$ and sensitivity by employing ungapped seeds. Darwin differs from CORAL as it aligns two genomes while we are mapping reads to assemble genome. A similarity between the two is the use of approximate string search algorithms. GateKeeper [139] implements the filtration stage on the FPGA and reports speedups over existing filtration schemes. The FPGA implementation, however, suffers from flexibility in mapping parameters such as read length and permissible edit distance. FPGAs, also, lack in on-board memory and communication bandwidth for faster data transfer between host processors, RAM and the FPGA chip, thereby, lagging behind the CPU in performance unless a large or multiple FPGA chips are used. Additionally, any change in the parameters may require extensive recoding and verification cycles. Jeremie S. Kim et al [140] present processing-in-memory (PIM) approach towards acceleration of filtration stage by implementing

their filter on a 3D-stacked DRAM. It aims to optimize the filtration algorithm for 3D-stacked memory with high memory bandwidth and PIM capabilities. This, however, limits its portability to other hardware architectures.

In the case of GPU acceleration, the kernels are, often, designed targeting specific GPU architecture, often, using vendor specified programming framework such as CUDA, as discussed in Section 2.4.4. GPU architecture is optimized for floating-point operations while genome assembly involves integer based operations; hence, GPU may or may not serve as the best possible choice for accelerating genome assembly. Thus, mappers optimized for just one platform, be it CPU, GPU or FPGA, are unable to use the advantage of all the available resources. GPUs, for example, accompany CPU in most of the modern platforms ranging from workstations to servers. To the best of our knowledge, CORAL, for the first time, demonstrates simultaneous usage of all available resources on a system without any additional programming effort and achieving up to 2$\times$ speedups. We showcase this by simultaneously deploying kernels on a quad-core CPU and two Nvidia GPUs. CORAL determines the maximum number of workitems in a workgroup in multiples of 2, as recommended by the Khronos group for better performance. OpenCL, then, automatically determines the total number of workgroups.

CORAL employs verification-aware filtration scheme which significantly reduces the average number of verifications performed per read. From Fig. 3.6, we can see that VA+A filtration scheme outperforms the others by reducing the average number of candidate locations that need to be verified. The VA+A scheme uses FM-Index backward search with pigeonhole principle in fully sensitive mode. It imposes approximation by limiting the maximum number of candidate locations per *k-mer* to 1000, as discussed in Section 3.4.1 and 3.4.2, thus, limiting the maximum number of verification 12000 per read for both forward and backward strand combined. We observed that only about 3.22% and 4.77% of reads in ERR012100_1 and SRR826460_1, respectively, produce large numbers of candidate locations in the VA case, as can observed in Fig. 3.6. Although, the proportion of reads is small, however, number of the candidate locations produced per read is huge enough to skew the average number of verifications from 479 to 1313 and 726 to 1580, respectively. Therefore, we limit the maximum possible verification cycles to 1000.

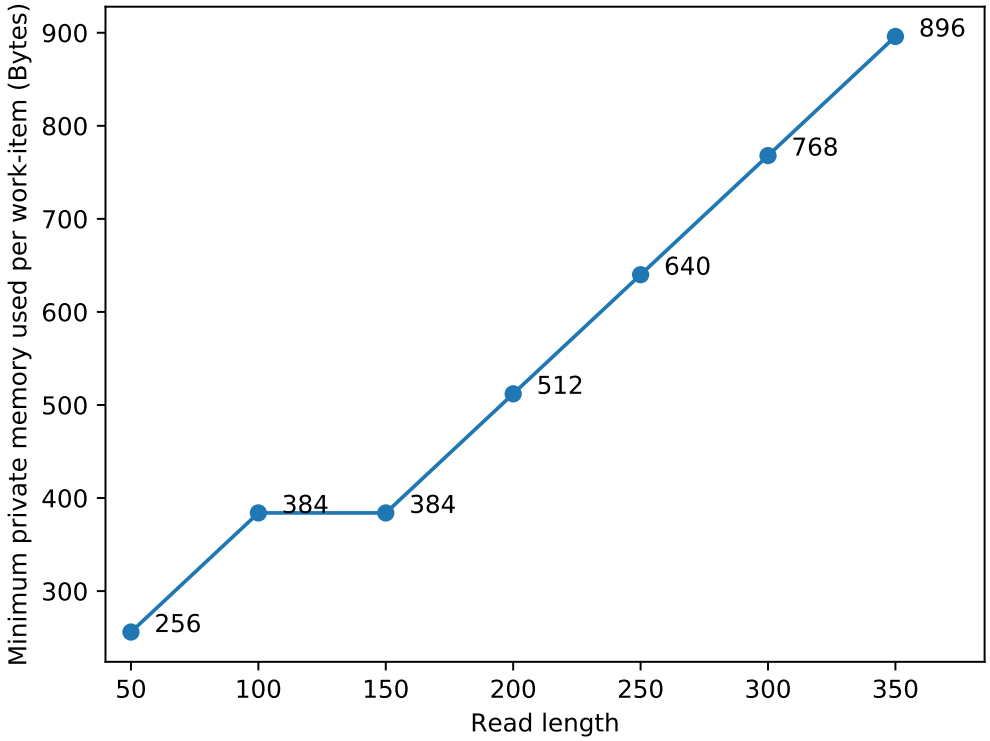

Figure 3.7: The minimum amount of private memory, in bytes, used by each workitem in the CORAL kernel for different read lengths.

CORAL, algorithmically, doesn't impose restrictions on the read lengths *per se*, however, in the current implementation it is practical to use it with short reads. This is because CORAL kernel loads read from the global memory to the private memory, as shown in Fig. 3.5, to reduce frequent memory accesses and from Fig. 3.7, we can see that the minimum private memory size used by each workitem in the kernel is proportional to the real length. These values were reported using inbuilt OpenCL function `cl.kernel_work_group_info.PRIVATE_MEM_SIZE`, which returns the minimum amount of private memory, in bytes, used by each workitem in the kernel [141]. In the current CORAL implementation, thus, the practicality of using longer reads depends on the availability of the private memory. CORAL does not produce the CIGAR string and SAM output format, yet. However, it reports the edit distance upon alignment, mapping

location, and the strand which is similar to the PAF format of the Minimap2 [124] and can be used in many genomic analysis pipelines such as metagenomics [142–145] except for variant calling. The memory footprint of CORAL is large due to tally and suffix array matrices, as mentioned in Section 3.4.1. These data structures, however, have the capability to significantly reduce their memory footprint as demonstrated in [26]. With OpenCL based framework, CORAL can be run on credit-card sized single board computers (SBCs), designed for embedded scenarios. Such board have multicore architectures along with GPU, however, limited memory. All the compute units available in the form of CPU and GPU can be simultaneously used using CORAL unlike other mappers proposed till date. We present aforementioned improvements in the following chapters of the thesis.

3.6 Summary

In this chapter, we address the first hypothesis stated in Section 1.2 by presenting a **C**ross-platf**O**rm **R**ead m**A**pper using openc**L** (CORAL) targeting heterogeneous systems. Such systems have different kinds of devices in various combinations on a single platform. Using CORAL all OpenCL conformant devices can be used concurrently to map reads in task-parallel fashion. For example, this chapter presents a case where a quad-core Intel CPU is accompanied by two Nvdia GTX 590 GPUs. To efficiently use all the compute resources on a system, CORAL employs OpenCL programming framework to launch kernels on the chosen devices and distributes the workload with the maximum possible workgroup size. Without any additional programming effort, CORAL can be use devices such as CPUs and GPUs, from different vendors, stitched together on laptops, workstations and servers. Additionally, it uses a number of algorithmic optimisations, including verification-aware filtration, to significantly reduce the computational costs. Both simulated and real reads are used to compare the runtimes and accuracy of CORAL with the state-of-the-art read mappers and showing competitive tradeoffs besides portability.

Chapter 4

Dynamic Programming based Filtration

4.1 Introduction

Chapter 2 discusses in detail the expansive nature of genomic data, especially, due to the NGS technology, which produces massive amounts of reads because of oversampling and amplification of sample genome during the sequencing process. Current trends indicate that our computation performance will scale poorly under the future demands generated from a large population [13]. To bridge the widening gap, Chapter 3 insists on hardware-agnostic, software based solution targeting heterogeneous architecture of most modern computers for maximum utilisation of available hardware resources to gain speedups. It showcases a scenario where computational tools are portable and flexible in using a variety of devices from different vendors across various platforms. This is achieved using OpenCL computing framework with impetus on mitigating additional implementation effort.

Besides performance, the energy required to process a large volume of data poses challenges from cost and environmental aspects. The growth in electricity consumption due to computing platforms such as laptops, workstations and servers has been higher than the growth in worldwide electricity consumption [146]. Genomics finds application in many domains including medicine, agriculture and forensics which has made it a top contributor to Big Data. The energy requirements posit a significant challenge to processing genomic data and needs to be addressed.

MESGA [147] originally explored read mapping on a multiprocessor system on a chip (MPSoC) based embedded system. They implemented an existing mapping tool, BWA-aln [25], on 16 processors with 2 GB memory each and showed 7× speed-up compared to linear pipeline on an Intel server. They achieve this by partitioning the genome so that the associated data structures can fit in the, respective, memory of each core. However, their results were demonstrated on a cycle-accurate simulator rather than a real hardware platform. The results presented were, only, for 1 million simulated reads without any energy measurement. Besides, an existing state-of-the-art tool was modified to adapt to embedded platform rather than designing it as per the requirements of the underlying hardware. As genome analysis involves integer based operations, more investigation is required to see if simpler cores found on the embedded System-on-Chip (SoC) platforms may be better suited than complex general purpose CPU cores, optimised for floating-point operations.

This chapter addresses hypotheses 1 and 2 stated in Section 1.2 by proposing an OpenCL based REad maPper for heterogeneoUs sysTEms (REPUTE). REPUTE is a cross-platform tool, similar to CORAL presented in Chapter 3, capable of maximizing parallelization on multiple OpenCL conformant devices. REPUTE uses a memory optimized dynamic programming based filtration method inspired by the Optimum Seed Solver (OSS) [85]. In contrast to OSS which uses hash-based indexing, REPUTE adopts the underlying theory and presents a novel optimised implementation using FM-Index and suffix array data structure. Compared to CORAL, which examines *k-mers* serially and uses a heuristic based variable length *k-mer* selection criteria, the DP based filtration in REPUTE improves selectivity as it examines the entire read before determining lengths and positions of *k-mer*. The lengths and position are optimally selected to minimise the total number of candidate locations per read. REPUTE gets rid of approximations used in CORAL filtration kernel and provides an efficient, theory backed, method for variable-length seed selection. Unlike the state-of-the-art mappers, our OpenCL based implementation enables us to demonstrate performance gains by launching parallel kernel executions on multiple devices. Most read mappers that have focused on reducing energy consumption have used an existing tool and implemented them with necessary modification. REPUTE, on the other hand, has adopted an

algorithm-hardware co-design approach to design a hardware-aware kernel with low memory footprint. It articulately integrates DP-based filtration algorithm with FM-Index backward search while keeping the number of operations to the minimum. In addition, the REPUTE kernel flow is optimised to mitigate the increase in memory footprint due to DP based filtration. We compare REPUTE with CORAL, GEM, Hobbes3, RazerS3, BWA-MEM and Yara [2, 16, 24, 25, 28, 148] by mapping 2 million real reads to chromosome 21 on two separate systems: 1) Intel CPU + 2×Nvidia GPUs; 2) HiKey970 embedded SoC with ARM Cortex-A73/A53 cores. We demonstrate that REPUTE is up to 13× faster than existing mappers with similar accuracy on System 1 and consumes up to 27× less energy on embedded SoC, with comparable performance and similar accuracy. To the best of our knowledge, this is the first work which demonstrates the possibilities and potential of Embedded Genomics. The source code can be found at: https://github.com/nclaes/REPUTE

4.2 Methodology

Sequencing process fragments genome randomly into smaller sections and identifies them as small strings called reads. To re-assemble the genome, these reads are mapped to a reference genome with an aim to find the possible candidate locations from where it may have originated in the actual genome. Approximate string matching is used to account for errors and variations between the actual and the reference genome. There are three stages in read mapping: Preprocessing, Filtration and Verification. We discuss Preprocessing and Verification, briefly, as they employ methods similar to CORAL. We concentrate our efforts, mainly, on the DP based Filtration method, which is the bottleneck to performance gains.

4.2.1 Preprocessing and Verification

The aim of the Preprocessing stage is to facilitate quick retrieval of information in the reference genome to speed-up the Filtration stage. REPUTE uses FM-Index [89] and suffix array [86] data structures to store the reference genome. These data structures have been, previously, used in many mappers including GEM, Yara, CORAL and BWA-

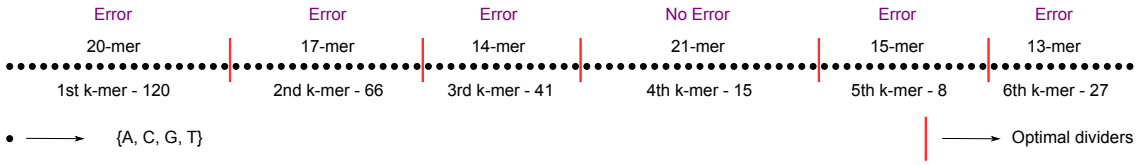

Figure 4.1: Fig. 2.13 is reused here to demonstrate the pigeonhole principle for ($n = 100, \delta = 5$), where n is read length and δ is error. *K-mers* with their respective number of candidate locations are displayed. The vertical lines are the optimal dividers, identified in filtration stage, to minimize the total number of candidate locations. The dots represent one of the four bases: {`A, C, G, T`}.

MEM. A detailed discussion on these data structures was presented in Chapter 2 and the algorithms and methodology followed to construct them was presented in Section 3.3.2 of Chapter 3. The objective of approximate string matching algorithm is to identify the candidate locations for reads in the reference genome where a read is likely to match. To find candidate locations, a k long section of the read, known as a *k-mer* as shown in Fig. 4.1, is searched using FM-Index backward search method. Upon a successful match, the location is obtained from the Suffix Array. This location points to a section in the reference genome where the entire read may align within the specified edit distance. Similar to CORAL, the alignment is performed in the verification stage using the Myer's bit vector algorithm. The implementation details of verification for REPUTE remains similar to that of CORAL, as discussed in Section 3.3.4.

4.2.2 Dynamic Programming based Filtration

Pigeonhole principle [2] states that δ errors cannot occur in more than δ sections of the read. Therefore, dividing a read in $\delta + 1$ sections will leave at least a section error free, which should, ideally, match exactly in the reference genome at locations where it is suppose to have originated. Fig. 4.1 demonstrates pigeonhole principle for read length $n = 100$ and $\delta = 5$, divided into *k-mers* with different lengths (k). As the error-free *k-mer* is not pre-known, all the $\delta + 1$ *k-mers* are scanned in the reference genome and all candidate locations are verified. The error free 4^{th} *21-mer* shown in Fig. 4.1 is randomly chosen, without any loss of generality, as it is not known beforehand. Each *k-mer* produces a different number of candidate locations depending on the starting, ending positions and their lengths. The vertical lines, in Fig. 4.1, are called optimal dividers.

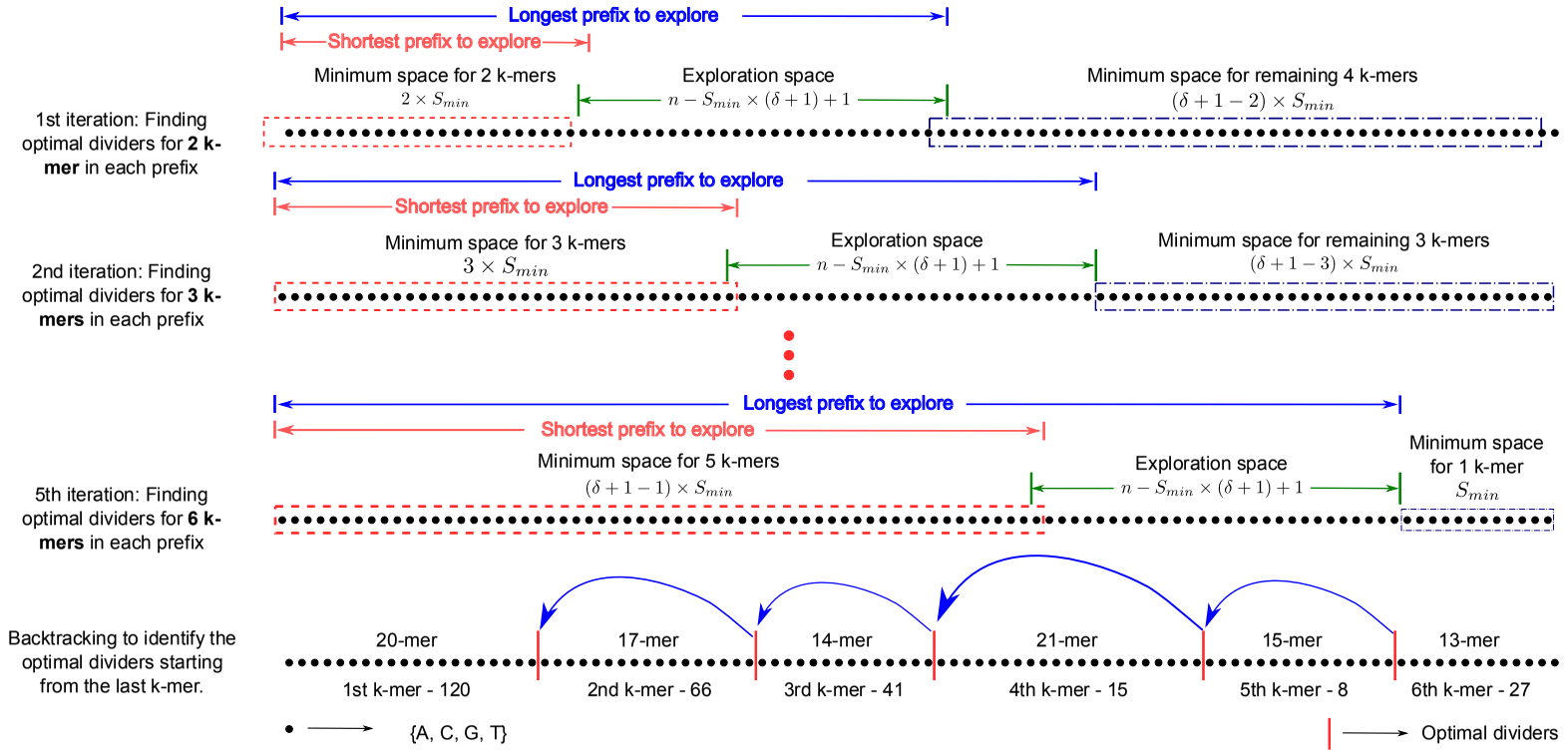

Figure 4.2: Demonstration of memory optimised dynamic programming based filtration algorithm for parameters ($n = 100, \delta = 5$). δ iterations are required to obtain optimal dividers for $\delta + 1$ *k-mers*. In the end, the optimal dividers are found starting from the last one using backtracking.

The optimal dividers partition the read into optimum length *k-mers* to minimize candidate locations, for better performance. The objective of a filtration approach, proposed in any read mapper, is to find an optimum set of *k-mers* with the aim to minimize the number of candidate locations. RazerS3 and Hobbes3 use hash-based data structures to store and retrieve reference genome while CORAL, Yara, BWA-MEM and GEM use FM-Index based methods.

Fig. 4.2 demonstrates our memory optimized DP based filtration method which is a variable-length seeding technique. To begin with, the read parameters of (n, δ) and minimum *k-mer* length S_{min} are specified. These parameters are user specified before execution. The algorithm requires δ iterations for $\delta + 1$ *k-mers*. In each iteration, a fixed number of prefixes are explored, called the exploration space ($n - S_{min} \times (\delta + 1)$). The exploration space ensures that the minimum length of *k-mers* is not violated. Within an iteration, each prefix is divided into two sections: 1^{st} and 2^{nd}, with the objective of identifying optimal divider for the two sections, of each prefix, in the exploration space. The first iteration aims to find optimal divider for first two *k-mers*. Therefore, in this case, the 1^{st} section is the 1^{st} *k-mer* and 2^{nd} section is the 2^{nd} *k-mer*. The algorithm, then, finds optimal divider for each prefix starting from the longest to the shortest, as shown in Fig. 4.2. The second iteration carries forward the solution of first iteration to identify the optimal dividers for first three *k-mers*. However, the difference, here, is that the 1^{st} section, now, consists of first two *k-mers* combined while the 2^{nd} section is the 3^{rd} *k-mer*. Likewise, in the last iteration, the 1^{st} section will consist of first δ *k-mers* and the 2^{nd} section will be $(\delta + 1)^{th}$ *k-mer*. At the end of all iterations, the backtracking process results in optimal divider for all the *k-mers*, as shown in Fig. 4.2.

For the DP based approach, the optimal dividers for all the prefixes in each iteration is required to be stored for backtracking in the end. This along with several other additional variables required by the algorithm, considerably, increases the memory footprint of the kernel. To avoid this, contrary to OSS, we have limited the exploration space from the entire read to the minimum space required. Among others, we optimized the bitwidths of variables to reduced memory footprint used FM-Index backward search in an efficient way to reduce memory accesses. However, it should be noted that the memory footprint varies with the size of the exploration space and, hence, depends on the n, δ and S_{min}. For

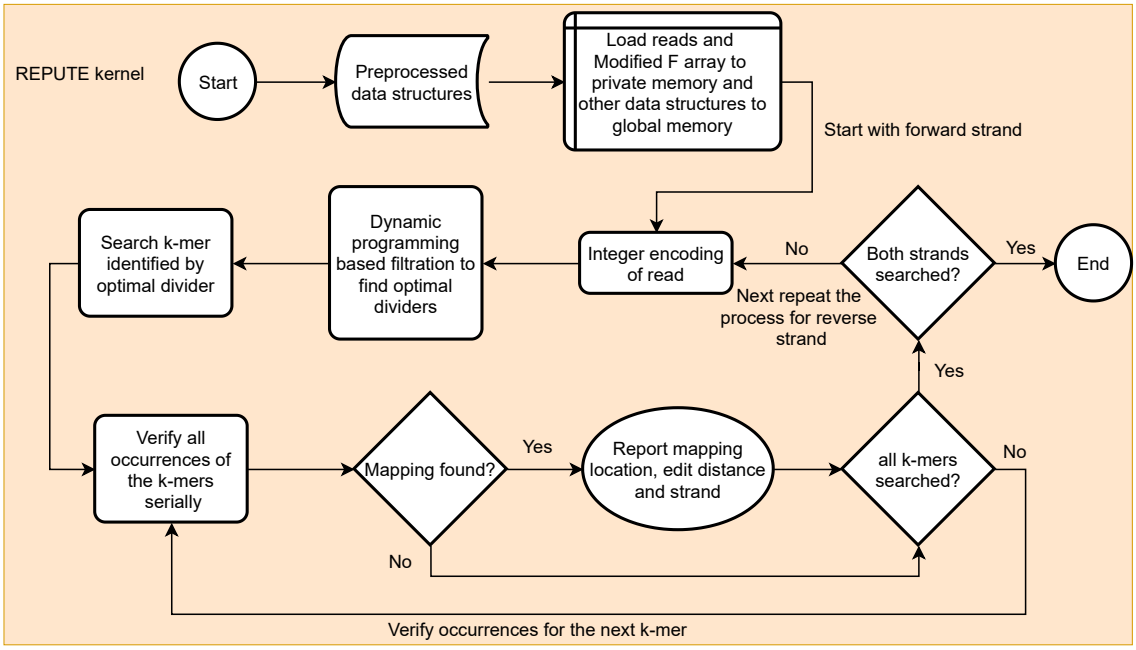

Figure 4.3: Flowchart of the REPUTE kernel.

smaller S_{min}, a highly optimized solution can be obtained at the cost of greater memory footprint and filtration time, while a larger S_{min} results in lower footprint but, relatively, longer mapping time due to a greater number of candidate locations. Inspired by OSS, we have retained all the optimizations proposed in [85]. To understand the approach in greater detail, interested readers can refer to [85].

4.3 Algorithm flowchart

Fig 4.3 presents the flowchart of the REPUTE kernel. When the kernel execution starts the data structures and inputs are loaded to the global memory. Each workitem loads a read and modified F array into the private memory for fast access. The read is integer encoded while loading so that it can be directly used as index to access the relevant element of the tally matrix and auxiliary data structures. This is followed by filtration where the optimal dividers that produce minimum candidate locations are identified. All the occurrences for each *k-mer* are verified using Myer's bit-vector algorithm and successful mapping locations are stored. The entire process is repeated for the reverse

strand of the reference genome.

4.4 Experimental Setup

The host program of REPUTE is written in `Python` and the kernel is in `C`. We use `PyOpenCL` because `Python` enables fast modifications and prototyping, yet, not affecting the mapping process. We compare REPUTE with CORAL, RazerS3 (`3.5.8`), Hobbes3 (`3.0`), Yara(`1.0.2`), BWA-MEM (`0.7.17`) and GEM (`3`). All mappers map 1 million (M) real reads each from NCBI databases: ERR012100_1 and SRR826460_1, to chromosome 21 of the Human Genome (version GRCh38/hg38) [136]. These databases consists of reads with lengths 100 and 150, respectively, and are mapped for error range (or edit distance) of 3-7. We use the following two platforms to map reads:

System 1: Intel Core i7-2600 CPU @ 3.40GHz, 16GB RAM and 2 $\times$ GeForce GTX 590, 1.5 GB RAM.

System 2: HiKey970 embedded SoC with ARM Cortex-A73 MPCore4 @up to 2.36GHz, ARM Cortex-A53 MPCore4 @up to 1.8GHz and 6 GB RAM.

We have used OpenCL 1.2 standard for portability across variety of platforms. This standard, however, imposes the following two restrictions:

a) It does not permit dynamic memory allocation. Hence, the number of outputs per read requires to be specified beforehand, in order to allocate sufficient memory.

b) The maximum amount of memory that can allocated to a variable is $(1/4)^{th}$ of the RAM capacity.

The aforementioned restrictions limit the maximum number of mapping locations per read due to the memory available to a variable. Thus, REPUTE reports the *first-n* mapping locations. It should be noted that we have compared, only, the mapping times wherever possible and all mappers used for comparison were configured to their recommended settings, unless specified. To ensure a comprehensive comparison, we have designed the following experiments.

4.4.1 Homogeneous Scenario

In this experiment, we run the mappers on the CPU of System 1. SAM file from RazerS3 is used as the gold standard, as it is an *all-mapper* with high accuracy and has been used in Hobbes3 and Yara, previously. We configure RazerS3 to report a maximum of 100 mapping locations per read while other mappers produce up to 1000 locations per read. As the number of mapping locations reported by any mapper for any read cannot be pre-determined, hence, we set a limit on the maximum number of locations. Yara and BWA-MEM were configured to report all locations as they are *best-mappers* and can either produce the best mapping location or all the locations. To determine the mapping accuracy, all the mapping locations reported by the gold standard per read is searched in the output of other mappers. Along with the mapping locations the genome strand that the reads were mapped to are, also, matched.

4.4.2 Heterogeneous Scenario

In this experiment, we execute REPUTE on both CPU and GPU. Due to limited RAM of 1.5 GB on the GPUs and fairness of comparison, all mappers report 100 locations per read excluding Yara and BWA-MEM, which report all the mapping locations. Unlike state-of-the-art mappers, REPUTE distributes the workload on CPU and GPU, as per user specification, executing the work-items in task-parallel fashion using OpenCL framework. To obtain accuracy measurements, we employ a method similar to *any-best* scenario of the Rabema benchmark [137]. In contrast to Section 4.4.1, we identify if all the reads mapped by the gold standard have been reported by other mappers with at least one matching mapping location and strand.

4.4.3 Embedded Scenario

HiKey970 SoC boots with a Linux distribution, `Lebuntu`, provided by the manufacturer. It needs to be flash booted instead of a bootable SD card. With limited onboard flash memory, we could install only a limited number updates and libraries. Among other mapping tools, we could successfully run, only, RazerS3, Hobbes3, CORAL and REPUTE on HiKey970. We adopt the same measurement methodology as stated in Section 4.4.2.

RazerS3 and Hobbes3 are `C++` based tools capable of multithreading for concurrent execution using all the 8 cores available on the embedded board.

4.4.4 Power and Energy Consumption

We compare Hobbes3, RazerS3, CORAL and REPUTE for energy efficiency using a power meter at the power source of the two systems. We measure the average power consumption during the mapping process and subtract it with the idle power to measure the power consumption during mapping process. The power is visually observed over many runs and is noted once it stabilises during the run. We multiply the power consumption with mapping time to measure energy consumption. For a fair comparison between homogeneous and heterogeneous scenarios, we need to distribute, approximately, equal amounts of reads between the CPU and GPUs. For this purpose, we chose following cases to take the measurements: $n = 100, \delta = 3$ and $n = 150, \delta = 5$. The former case maps 480,000 reads and the latter maps 500,000 reads on the GPU using REPUTE.

4.5 Results and Discussion

Table 4.1 presents the results of the Homogeneous scenario mentioned in Section 4.4.1. It is evident that REPUTE outperforms RazerS3, Yara, BWA-MEM on both runtimes and accuracy for all error profiles. REPUTE is up to 13$\times$ faster than Yara. RazerS3 is configured to produce 100 outputs per read compared to 1000 outputs for other mappers, hence, reducing its mapping time significantly. Except for ($n = 100$, $\delta = 5$), REPUTE performs better than Hobbes3 and GEM. The mapping accuracy of REPUTE is considerably better than GEM and is equal or comparable to Hobbes3. We can see that REPUTE provides up to 4$\times$ speedup over Hobbes3 for lower error profiles and longer read lengths. Compared to CORAL, DP based filtration has reduced the mapping time, especially, for longer read lengths and high error profiles.

Table 4.2 presents the results of the Heterogeneous scenario discussed in Section 4.4.2. The performance of REPUTE compared to other mappers follow similar trends as discussed in the previous paragraph. The contrast between the mapping times in

REPUTE-cpu and REPUTE-all demonstrate that performance can be enhanced by using multiple devices in parallel. Using GPUs, we obtained an additional speedup of up to $\approx 2\times$ making REPUTE up to $7\times$ faster than Hobbes3 for longer reads and smaller error profiles.

REPUTE-all uses CPU along with two Nvidia GPUs to distribute the workload and map reads in task parallel fashion. It launches the kernels simultaneously and upon completion it combines the results, thus, making one of the devices the performance bottleneck. The distribution of workload among various devices, hence, should be performed judiciously to obtain optimum performance. The maximum number of reads that can be mapped on a device is limited by the private memory available to the compute cores and the global memory or RAM available to the device. Fig. 4.4 presents a scenario of performance gains by offloading more workloads to GPU for given ($n = 150, \delta = 5$) and fixed minimum *k-mer* length of 22. Large *k-mer* lengths reduces the memory footprint of the kernel allowing more workgroups to be processed by the GPU without running out of resources. The extreme point on the left in Fig. 4.4, indicates the mapping time when GPU is not used while the rightmost point gives the mapping time when all reads are mapped on the GPUs. We can see that utilizing GPUs along with CPUs can provide additional performance gains, however, using only the GPUs deteriorates the performance as shown by the rightmost point in Fig. 4.4. This is because GPUs are not suitable for task-parallel executions compared to the performance delivered by CPU.

Table 4.1: The results of mapping 2M real reads to chromosome (chr) 21 on the CPU. T represents mapping time in seconds and A represents accuracy, measured in accordance with Section 4.4.1. Hobbes3, GEM and REPUTE-cpu reported up to 1000 mapping per read. Yara, however, by default reports all the mapping positions and BWA-MEM, being a *best-mapper*, is configured to report all mapping positions. RazerS3 is used as the gold standard with 100 outputs per read.

Chromosome 21 System 1	Read length	100						150					
	Error	3		4		5		5		6		7	
	Time/Accuracy	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)
Intel Core i7-2600 CPU@3.4GHz, 16GB RAM + 2 × GeForce GTX590, 1.5GB RAM	RazerS3	26.7	100	42.6	100	65.7	100	30.7	100	50.6	100	91.3	100
	Hobber3	21.6	100	18.6	100	16.6	100	58.4	100	50	100	40.7	100
	Yara	10	5.22	21	4.51	25.5	4.00	38.2	5.27	116.5	4.54	321.4	4.14
	BWA-MEM	T(s) - 82.2			A(%) - 39.94			T(s) - 159.1			A(%) - 30.82		
	GEM	22	4.88	22	4.14	21	3.59	56	4.74	54	4.15	53	3.68
	CORAL-cpu	7.03	99.96	16.34	99.91	32.29	99.87	17.31	100	37.36	100	66.35	100
	REPUTE-cpu	**7.49**	**99.99**	**14.88**	**99.98**	**24.92**	**99.94**	**13.75**	**100**	**21.1**	**100**	**33.4**	**99.99**

Table 4.2: The results of mapping 2M real reads to chromosome (chr) 21 on the CPU + GPU. T represents mapping time in seconds and A represents accuracy, measured in accordance with Section 4.4.2. Hobbes3, GEM and REPUTE-all report up to 100 mapping per read. Here, REPUTE-all, however, distributes the workload on both CPU and GPUs. Yara, by default, reports all the mapping positions and BWA-MEM, being a *best-mapper*, is configured to report all mapping positions. RazerS3 is used as the gold standard and reports 100 outputs per read.

Chromosome 21 System 1	Read length	100						150					
	Error	3		4		5		5		6		7	
	Time/Accuracy	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)
Intel Core i7-2600 CPU@3.4GHz, 16GB RAM + 2 × GeForce GTX590, 1.5GB RAM	RazerS3	26.7	100	42.6	100	65.7	100	30.7	100	50.6	100	91.3	100
	Hobber3	20.4	100	16.9	100	14.6	100	58.2	100	49.5	100	40.5	100
	Yara	10	99.2	21	99.4	25.5	99.5	38.2	100	116.5	100	321.4	100
	BWA-MEM	T(s) - 82.2		A(%) - 97.16				T(s) - 159.1		A(%) - 95.09			
	GEM	22	92.9	22	91.4	22	89.4	54	90.2	54	91.3	53	89.1
	CORAL-all	5.24	99.98	9.74	99.97	24.73	99.98	12.2	100	29.47	100	56.05	100
	REPUTE-all	**5.27**	**99.99**	**12.65**	**99.99**	**19.8**	**99.9**	**7.87**	**100**	**12.9**	**100**	**23.9**	**100**

Table 4.3: Read mapping on the HiKey970 SoC. T - time in seconds and A - accuracy, measured in accordance with Section 4.4.3.

Chromosome 21 System 2	Read length	100						150					
	Error	3		4		5		5		6		7	
	Time/Accuracy	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)	T(s)	A(%)
ARM Cortex-A73, A53, 8 cores, 6 GB RAM	RazerS3	89.1	100	127.5	100	222.3	100	96.8	100	168.1	100	328.1	100
	Hobber3	54.06	100	47.37	100	46.68	100	89.95	100	78.21	100	69.34	100
	CORAL-HiKey	16.41	100	38.39	100	67.48	100	38.65	100	78.50	100	134.1	100
	REPUTE-Hikey	**17.47**	**99.99**	**35.35**	**99.99**	**60.61**	**99.99**	**49.44**	**100**	**56.3**	**100**	**84.72**	**100**

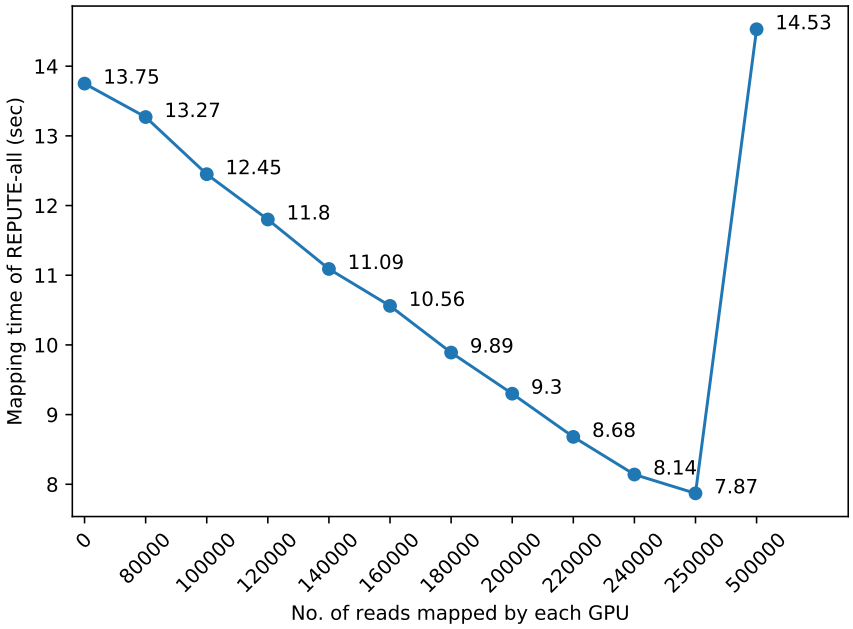

Figure 4.4: Mapping time for different distributions of workloads on CPU and GPU for ($n = 150, \delta = 5$) and minimum *k-mer* length of 22. X-axis shows the number of reads, out of 1 million, mapped by each GPU and the remaining reads are mapped on the CPU.

Fig. 4.5 shows the mapping times for different minimum *k-mer* lengths with a constant workload distribution between CPU (820,000 reads) and GPU (90,000). For smaller *k-mer* lengths, the mapping time is higher due to exploration of larger number of possibilities in the DP based filtration method. As the *k-mer* lengths increase, the time taken in filtration decreases, still producing similar number of candidate locations per read. However, for larger *k-mer* length of 20, the exploration space to select the optimum divisions of read is fewer, thus, resulting in higher candidate locations per read and, therefore, longer mapping time. The results presented in Table 4.1 and 4.2 are the best performances of REPUTE taking into consideration the *k-mer* lengths and workload distribution.

Table 4.3 presents the results for the embedded Scenario mentioned in Section 4.4.3. We can see that for ($n = 100, \delta = 5$) and ($n = 150, \delta = 7$), REPUTE performs comparable to Hobbes3 with similar accuracy while for other cases it outperforms Hobbes3. REPUTE is up to 4× times faster than RazerS3. Table 4.4 presents the power and energy measurements mentioned in Section 4.4.4. We observe that REPUTE-all,

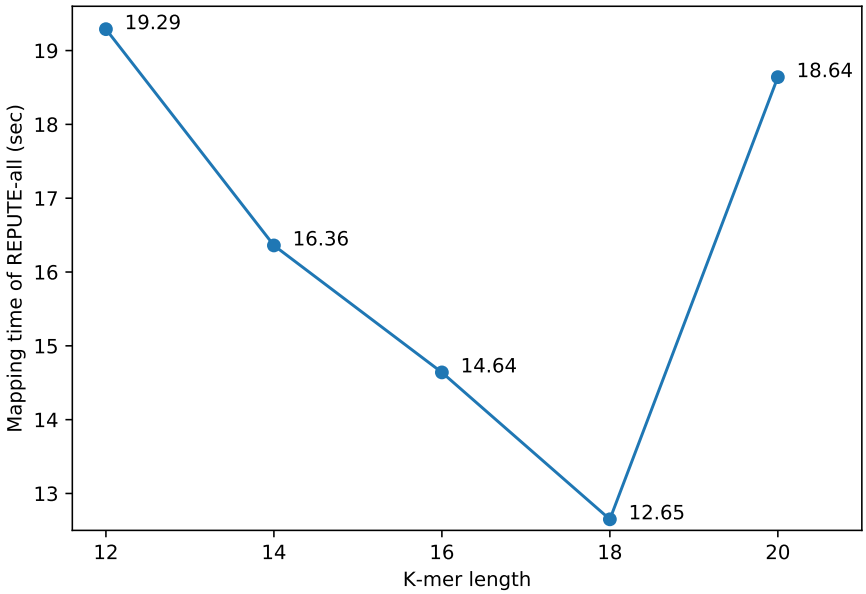

Figure 4.5: Mapping time for different minimum *k-mer* lengths with same distributions of workloads on CPU and GPU. CPU mapped 820,000 reads and GPU mapped 90,000 reads each with the read configuration of ($n = 100, \delta = 4$).

which distributes workload on CPU and GPU, uses more power but less energy and is faster than other mappers including REPUTE-cpu. On Hikey970, REPUTE outperforms other mappers, significantly, on the energy consumption. However, the most important takeaway, in our opinion, is that exploration of genomic computations on embedded SoCs can produce significant energy savings which can lower down the overall cost of whole genome sequencing. We demonstrate energy savings of upto 20$\times$ on HiKey970 embedded SoC compared to general purpose workstations, as shown in Table 4.4.

Currently, REPUTE is tailored to map short reads of length 100-150, even though the algorithm does not impose any such restrictions *per se*. REPUTE reports the mapping positions, edit distance and strand for each read in the output file. One of the reasons for large memory footprint of REPUTE is due to limitations on dynamic memory allocation posed by the OpenCL standard. OpenCL does not allow dynamic memory allocation and hence, each read should be assigned the same amount of memory to store mapping locations and associated information, even though it may match to, say, just one location.

Table 4.4: Energy consumption in accordance with Section 4.4.4.

	$n = 100, \delta = 3$		$n = 150, \delta = 5$	
	P(W)	E(J)	P(W)	E(J)
	System 1 - 160 W (Idle power)			
RazerS3	241	2162.7	243	2548.1
Hobbes3	254	1917.6	258	5703.6
CORAL-CPU	365	1440.1	371	3652.3
CORAL-all	454	1540.7	461	3673.1
REPUTE-CPU	354	1691.5	358	2859.1
REPUTE-all	455	1554.7	490	2597.1
	System 2 - 3.5 W (Idle power)			
RazerS3	7.5	356.3	8.6	493.5
Hobbes3	7.5	216.2	8.4	440.8
CORAL-HiKey	8.5	82.06	9.1	216.5
REPUTE-HiKey	8	78.6	7.8	212.6

Hence, depending on the RAM available, we may have to limit the number of mappings per read or run the kernel multiple times after grouping reads in sets. Another reason for large memory footprint is the size of the FM-Index data structure and suffix array, which becomes prohibitive for large chromosomes, especially, for embedded platforms. This, however, can be significantly reduced by storing elements after fixed intervals as used in [26] at the cost additional computational burden.

4.6 Summary

In this chapter, we attempt to prove the hypotheses stated in Section 1.2. We propose a cross-platform OpenCL based REad maPper for heterogeneoUs sysTEms (REPUTE)

capable of parallel kernel executions on multiple devices. REPUTE uses a memory optimized dynamic programming based algorithm for performance driven pruning of reference genome to map short reads. REPUTE kernel was designed using algorithm-hardware co-design approach targeting embedded platform with limited memory and simpler cores. The kernel uses filtration approach inspired by the OSS [85], which has been shown to select optimum seeds and produce minimum number of candidate locations for a read. We have compared REPUTE with state-of-the-art read mappers using real human reads on two different platforms. The low-memory footprint kernel of REPUTE with optimised implementation for performance, outperforms other mappers on mapping time and accuracy parameters in most of the cases. REPUTE demonstrates, for the first time, the potential of embedded genomics for energy efficiency without loss of accuracy and competitive performance. Compared to other mappers, REPUTE provides better energy savings. Our results show that moving genomics from high-performance servers and workstations to embedded systems can potentially unleash new opportunities for low-cost genomics.

Chapter 5

Embedded Whole Genome Read Mapping

5.1 Introduction

Prerequisite to genomics is the availability of genome which is obtained from the sequencing and assembly pipelines of the whole genome sequencing (WGS) [149]. Sequencing process produces fixed-length small subsections of genome, called reads, which are then reassembled to obtain the original genome. To reassemble the genome, reads obtained from sequencing cycles are mapped to an existing reference genome using read mapping tools. The mapping process engages approximate string matching and dynamic programming (DP) algorithms in tandem with the reference genome, stored in the form of data structures following a tool-specific preprocessing strategy. Conventionally, most of the state-of-the-art read mappers, such as [2, 16, 24, 25], have been optimized for CPU and are oblivious to other hardware resources available in modern heterogeneous systems such as the GPU. Several platform-specific tools have been proposed targeting FPGA and GPU; however, they are not flexible to changes in parameters and, often, require platform-specific programming skills [30].

CORAL [148], proposed in Chapter 3, demonstrates an OpenCL based heterogeneous read mapping scenario where workloads are distributed on available CPU+GPU to accelerate read mapping. It, however, uses a heuristic based filtration methodology. Also, the preprocessed data structures have large memory footprint making it unfit to be used with longer chromosomes (e.g. chr 1, chr 2) in memory-restricted environments such as the embedded platforms. REPUTE [150], presented in Chapter 4, proposes a DP

based filtration methodology using OpenCL to improve performance and demonstrates an embedded implementation of read mapping on HiKey970 platform with energy savings of 27× compared to a workstation. Though, REPUTE outperforms state-of-the-art read mappers for chromosome 21 but the size of data structure renders it infeasible for longer chromosomes on embedded platforms. Hobbes3 [16] uses a DP based filtration methodology along with heuristic schemes to optimize performance on *q-gram* inverted index for high-performance. RazerS3 [2] is accuracy focused, commonly used as gold standard for comparison but it does not employ any data structures to accelerate read mapping. As such, it is slower than other state-of-the-art read mappers. As discussed in Section 4.1, MESGA [147] is among the few contributions that aim to provide an embedded genomic solution to the computational pipelines, however, it is a simulation work rather than an actual implementation. SWARAM [22] is a recent contribution to embedded genomics, where state-of-the-art read mapper BWA-MEM is modified and implemented on embedded platform to map reads. It partitions the HRG into sufficiently small sections to reduce the memory footprint and map a portion of the short read files in each thread. This process is repeated in a task-parallel fashion covering the entire HRG to map all the reads distributing the workload on available processing cores. It, also, integrates downstream analysis tools such as Platypus [151] and/or GATK HaplotypeCaller [152], to process the assembled genome and accelerate it using an embedded system cluster. It was demonstrated in Chapters 3 and 4 that CORAL and REPUTE are faster and accurate than BWA-MEM, especially, when all mapping locations are needed. This chapter demonstrates similar results and uses algorithm-hardware co-design approach to design dedicated read mapping tool targeting embedded platforms, thus proving the second hypothesis stated in Section 1.2.

This chapter proposes a Pyopencl based tooL for gEnomic workloaDs tarGeting Embedded platforms (PLEDGER). PLEDGER aims to optimise the read mapping algorithm for the target memory-restricted hardware platform to enable translational genomics. It is an OpenCL based tool offering, in principle, identical portability to that of CORAL and REPUTE. However, PLEDGER outperforms CORAL and REPUTE as it is implementable on a memory restricted embedded platforms for all chromosomes: 1-22, X and Y. PLEDGER employs algorithm-hardware co-design approach to propose a novel

preprocessing scheme capable of generating memory-aware data structures on platforms with available RAM capacity of 3.6 GB. It can complete entire read mapping process with small memory footprint, making it a stand-alone tool tailored for embedded platforms. It uses DP based filtration and verification kernel akin to REPUTE with modifications to use memory-aware data structures, which affects the performance as a trade-off. To improve performance, it optimises the algorithm for the target memory-restricted platform using bit-vector operations and localized variable optimizations to minimize the memory footprint of the kernel. In CORAL and REPUTE, the mapping process needs to be repeated for each chromosome while PLEDGER is capable of mapping to all or user selected chromosomes, automatically, making it first of its kind implementation to map the whole genome on an embedded platform. In addition to parallel kernel executions on heterogeneous systems offered by CORAL and REPUTE, the memory-aware data structures used in PLEDGER enables mapping the whole genome on any GPU with over 3.6 GB available RAM.

This chapter compares PLEDGER with RazerS3, Hobbes3, CORAL and REPUTE by mapping 1 million real human reads of lengths 100 and 150 each, to chromosomes 1-22, X and Y of the human genome. We execute read mappers on two systems 1) Intel i7-8750H CPU, 16GB RAM + Nvidia GTX 1050 Ti, 4GB RAM; 2) Odroid N2 with quad-core ARM Cortex-A73 + dual core Cortex-A53, 4GB RAM. Among other state-of-the-art read mappers, only Hobbes3 and RazerS3, although oriented towards CPU, were successfully executed on an embedded platform for comparison by REPUTE on HiKey970 with 6GB RAM for chr21. However, with just 4GB RAM on Odroid N2 platform, only Hobbes3 and PLEDGER could be successfully benchmarked. We demonstrate up to 11$\times$ speedup compared to state-of-the-art read mappers. Our embedded implementation consumes 5.9$\times$ less energy than state-of-the-art computing resources. The PLEDGER source code can be found at: https://github.com/chitlangia/pledger

5.2 Methodology

Fig. 5.1 reuses Fig. 2.11 to visualise an overview of the read mapping process. It starts with preprocessing the reference genome and storing in the form of data structures

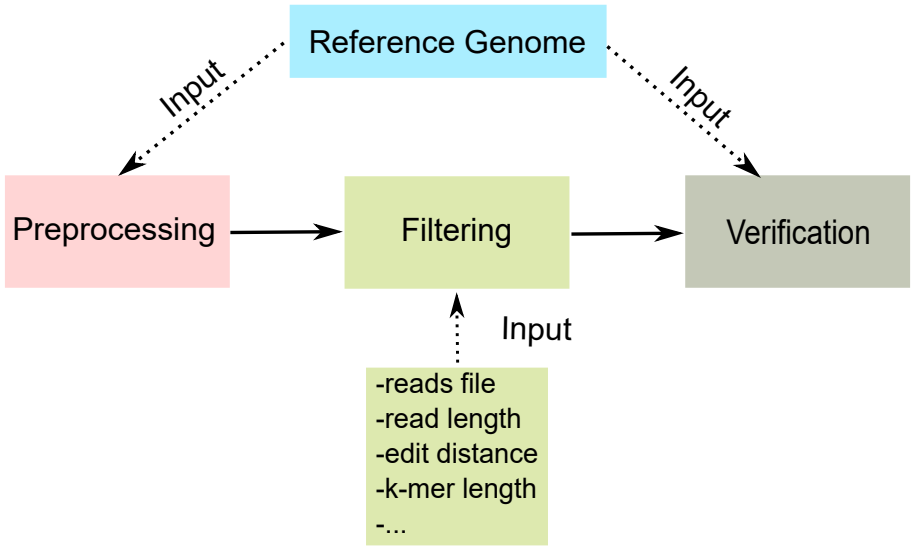

Figure 5.1: Overview of the read mapping process explained by reusing Fig. 2.11. There are three stages: Preprocessing, filtration and verification. Reference genome is the input to the preprocessing and verification stages. The are other inputs to the filtration stage such as reads file, read length, edit distance, k-mer length, constants, etc.

suitable to filtration scheme of the mapper. The objective is to assist the filtration stage with rapid pruning of reference genome while searching for possible candidate locations for a read. These candidate locations are then verified against the reference genome to find if the read originated from this location in the original genome during sequencing. Verification is performed in the ambit of δ mismatches and indels, known as error or edit distance, which originate during the sequencing process or are natural variations between the genomes of different individuals. Verification leverages a variant of the semi-global DP algorithm, known as Myer's bit vector algorithm. It is one of the fastest and widely used method whose details can be found in Chapter 2 and 3. The algorithmic flowchart for PLEDGER kernel is similar to that of REPUTE discussed in Section 4.3 using Fig 4.3.

In the following subsections, we focus on the proposed preprocessing scheme and associated filtration modifications to reduce the memory footprint and improve performance of the read mapper.

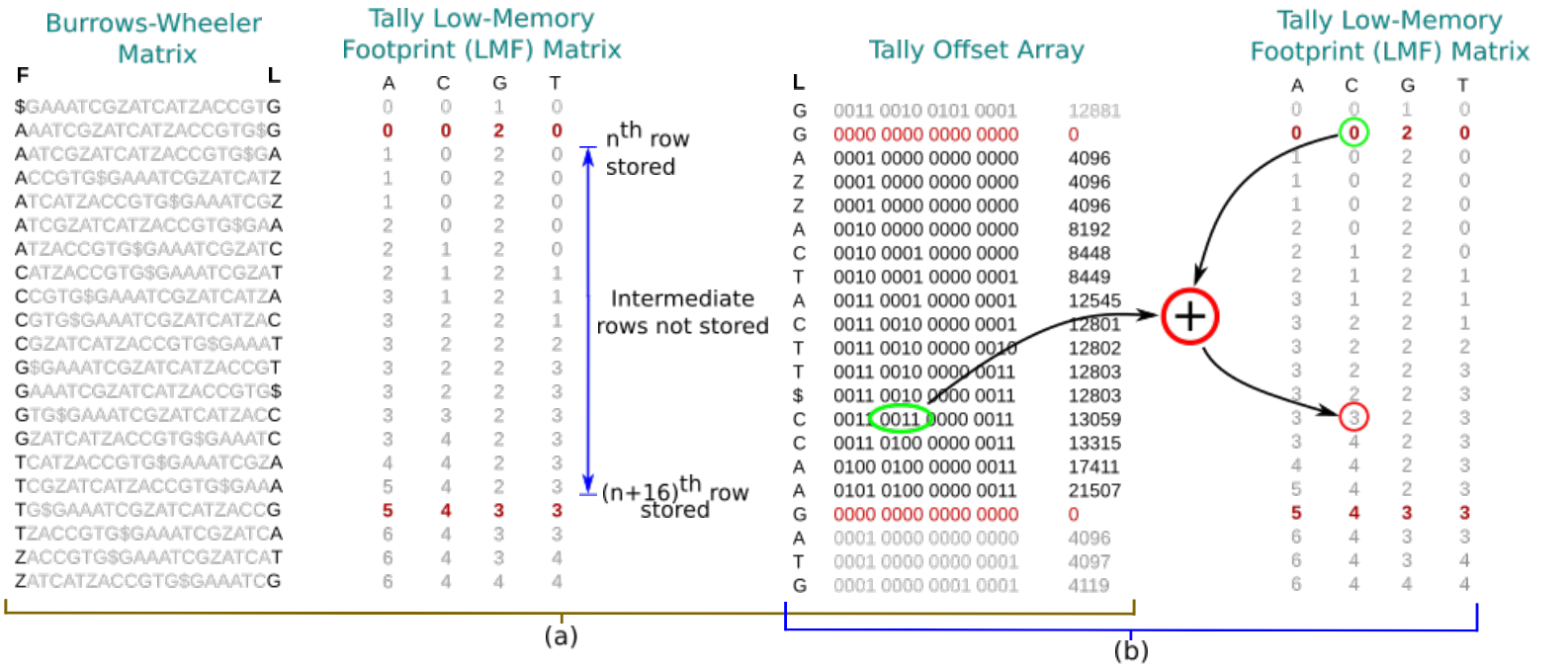

Figure 5.2: Demonstration of proposed low memory footprint (LMF) tally matrix along with the tally offset array to help obtain the missing information in the tally LMF matrix during filtration in $O(1)$ time.

5.2.1 Memory-aware preprocessing

The reference genome is preprocessed and stores as the auxiliary data structures of FM-Index and suffix array. FM-Index backward search offers $O(n)$ time complexity to search a string of length n, making it one of the fastest approximate string matching algorithm. These data structures have been, previously, used in many mappers including Bowtie2, Yara and BWA-MEM and were discussed in detail in Chapters 2 and 3. Fig. 5.2 visualizes the construction of FM-Index data structure for the string `GAAATCGZATCATZACCGTG$`. It involves the formation of tally matrix using **F** and **L** arrays obtained by applying Burrows-Wheeler transform on the string. The length of tally matrix depends on the length of the chromosome/genome and each row stores four integers (16 Bytes) making it a major bottleneck to low-memory implementations. For example, for the longest chromosome, chr1, the size of tally matrix is 4GB.

To reduce the size of tally matrix, only a limited number of rows at fixed intervals can be stored, provided the **L** array is available to reconstruct the missing rows during run-time. During run-time, the **L** array will have to be looped over the rows missing in the tally matrix to count the number of occurrences of bases (`A C G T`). This count will be added to the nearest available row, in the tally matrix, to obtain the values of the desired row. The DP based filtration methodology used in this chapter would require repetitive looping over the **L** array numerous times which will significantly increase the filtration time. To reduce the complexity from $O(n)$ to $O(1)$, we eliminate looping and use bit-vector operations with the help of an additional tally offset array as shown in Fig. 5.2. We store every 16^{th} row in the tally matrix and store the number of occurrences of `A C G T` for the missing 15 rows using 4-bits each, enabling a 16-bit unsigned integer to store the information of each row of the **L** array. This step compresses the data structure and as we will see in the next subsection it is decompressed in run-time during filtration. As preprocessing is a one-off task, it does not affect the runtime and prevents looping. The proposed preprocessing scheme reduces the size of tally matrix by $\approx 5.5\times$, bringing 4 GB down to 746.9 MB for chr1. PLEDGER requires five types of preprocessed data structure during runtime for read mapping: integer encoded reference genome (`A C G T` to `0 1 2 3`), suffix array, tally low-memory footprint (LMF) matrix, tally offset array and

run-length encoded version of **F** array. The total size of these data structures varies from 373.6 MB to 746.9 MB for chr 1-22, X and Y.

5.2.2 Filtration for memory aware data structures

Approximate string matching with error δ will require the use of pigeonhole principle [2]. Pigeonhole principle states that if a read is divided into non-overlapping $\delta + 1$ sections, then at least one section would be left error-free and match exactly at its place of origination. The non-overlapping sections of length k are called *k-mer*. Each *k-mer* from the read is pruned through the reference genome using FM-Index backward search to find the candidate locations. Fewer number of candidate locations will lead to less DP-based verification cycles and will greatly reduce the overall mapping time. The objective of filtration is to identify suitable *k-mers* to minimize the total number of candidate locations. We use DP-based filtration algorithm proposed in [150]; however, with our proposed memory aware data structure we modify the backward search of the algorithm using bit-vector operations in conjunction with method used to build tally offset array. Through this method, the desired value of tally matrix is known in $O(1)$ time by adding the tally offset value to the immediate tally matrix value before it, as shown in Fig. 5.3. At the end of filtration, the candidate locations are obtained from the Suffix Array and verified, *in situ* for alignment within error δ in the same work-item (or thread).

5.3 Experimental setup

PLEDGER is `PyOpenCL` based with the host program written in `Python` and the kernel in `C`. `Python` enables easy handling of strings and fast prototyping. We compare PLEDGER with REPUTE, CORAL, RazerS3 (`3.5.8`) and Hobbes3 (`3.0`). As mentioned in Section 5.1, we use RazerS3 as the gold standard and use a method similar to *any-best* scenario of the Rabema benchmark [137] for accuracy comparison. All mappers map 1 million (M) real reads each from NCBI databases: ERR012100_1 and SRR826460_1, with lengths 100 and 150, respectively, to chromosome 1-22, X and Y of the Human Genome (version GRCh38/hg38) [136]. We perform verification with 5% error rate i.e. $\delta = 5$ and 7 for read length $n = 100$ and 150, respectively. We use the following two platforms to

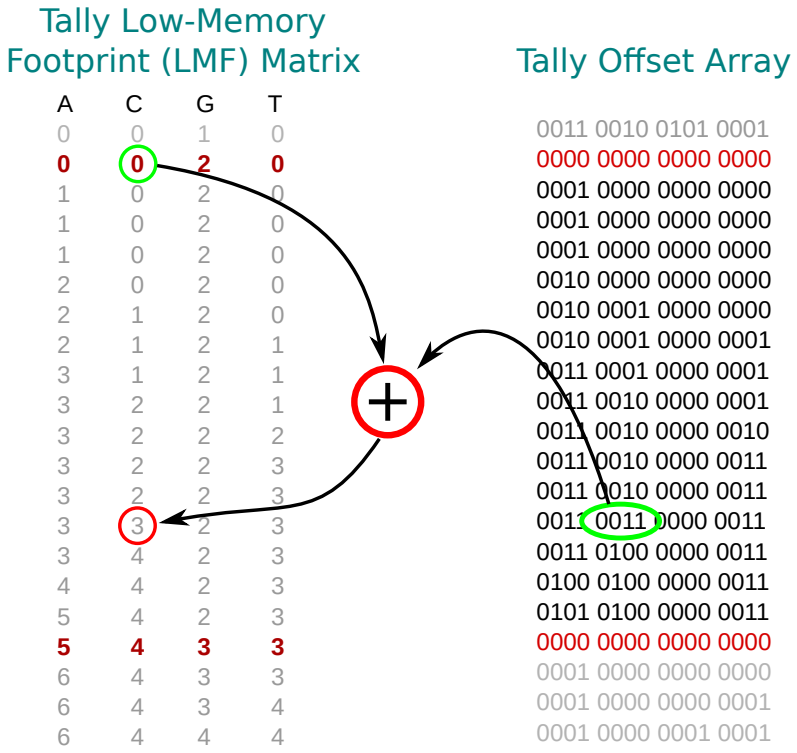

Figure 5.3: Visualization of bit-vector operation to obtain the desired element of the tally matrix using tally LMF matrix and tally offset array.

map reads:

System 1: Intel i7-8750H CPU, 16GB RAM + Nvidia GTX 1050 Ti, 4GB RAM.

System 2: Odroid N2 with quad-core ARM Cortex-A73 + dual core Cortex-A53, 4GB RAM.

We use OpenCL 1.2 standard to support portability across different devices. OpenCL 1.2, however, does not allow dynamic memory allocation and the maximum memory that can be allocated to one variable cannot exceed $(1/4)^{th}$ of the RAM capacity. This had earlier restricted floating of large tally matrices in CORAL and REPUTE on memory restricted platforms. Similar to REPUTE and CORAL, PLEDGER reports the *first-n* mapping locations. We compare the mapping times of different mappers, with their recommended settings, configured to report 100 mapping locations per read.

5.4 Results and discussion

All mappers in the following experiments have been configured to report 100 mapping locations per read. In the following subsections, we present mapping times for different read lengths and chromosomes on system 1 and 2.

5.4.1 System 1 - CPU+GPU

In this experiment, we run RazerS3, Hobbes3, CORAL and REPUTE on the CPU of System 1. Although, CORAL and REPUTE can execute on heterogeneous platforms, however, the memory requirements prohibit them to do so for longer chromosomes. Table 5.1 compares different mappers for reads of length $n = 100$, and error $\delta = 5$. PLEDGER was run twice: once only on the CPU, denoted as **PLEDGER-cpu**, and second time both on the CPU and GPU by distributing workload in the ratio 4:1, denoted as **PLEDGER-all**. As per our observation, using this ratio produce the best mapping times. We can see that PLEDGER-cpu and PLEDGER-all outperforms RazerS3 and CORAL for all chromosomes, producing 1.6-11$\times$ speedups. CORAL is slower than PLEDGER as it uses heuristic based filtration methodology producing more candidate locations that lead to as many expensive DP based verification cycles. PLEDGER performance is comparable to REPUTE while PLEDGER-all outperforms as it can use GPU due to low-memory footprint data structures. Hobbes3 has outperformed PLEDGER in performance. The performance gap, however, narrows when PLEDGER distributes workload on available GPU using its parallel heterogeneous execution capabilities. We can observe similar trends for $n = 150$ and $\delta = 7$ as shown in Table 5.2. For chromosome Y in Table 5.2, we see that PLEDGER-all outperforms Hobbes3. Chromosome Y is the smallest of all chromosomes and we observe that PLEDGER's performance improves for smaller chromosomes. In Section 5.4.5, we discuss the reasons for high performance of Hobbes3 compared to PLEDGER.

5.4.2 System 2 - Odroid N2

Table 5.3, 5.4 present the mapping times for Hobbes3 and PLEDGER on Odroid N2 embedded platform. Among existing read mappers, only Hobbes3 is capable of executing in a memory restricted environment. From the tables, we can see that Hobbes3 outperforms PLEDGER in all cases. Even though PLEDGER is capable of using the on-board Mali GPU but we found that distributing workload to GPU, like in system 1 (Section 5.4.1), does not yield any additional performance gains. This is because on-board Mali GPU doesn't have a dedicated RAM and shares the RAM with ARM processors. The architecture of Mali is, relatively, simple compared to Nvidia GPU and has low operational frequency of 950 MHz compared to 1392 MHz for the Nvidia GPU. Although, PLEDGER is slower compared to Hobbes3 it provides an opportunity for portability to OpenCL conformant devices and scalability for implementation on an embedded cluster to accelerate data intensive genomic workloads.

Table 5.1: Mapping time(in seconds) for 1M real reads, of length $n = 100$ and error $\delta = 5$, to chromosome (chr) 1-22, X and Y on system - 1 (CPU+GPU). PLEDGER-all distributes workload over CPU and GPU in 4:1 ratio while others execute only on the CPU.

	chr1	chr2	chr3	chr4	chr5	chr6	chr7	chr8	chr9	chr10	chr11	chr12
RazerS3	335.3	338	269.1	246.6	245.9	234.2	233.5	194.5	181.9	199.2	186.1	190.6
Hobber3	20.3	18.7	17	16.3	16.5	15.7	16.4	14.7	14.3	15.2	14.2	15.3
CORAL	48	47.1	47.3	47.5	45.6	38.8	45.8	40	40.1	44.2	35.3	37.3
REPUTE	38.7	36.5	33.5	32.2	32.2	31.4	35.3	30.9	30.7	32.7	30.5	32.5
PLEDGER-cpu	35.6	35.4	35.6	34.5	34.5	33.4	34.6	31.5	31.1	32.2	30.4	32
PLEDGER-all	**30.1**	**29.3**	**27.5**	**26.2**	**26.3**	**25.5**	**26.3**	**24.1**	**23.6**	**24.7**	**24**	**24.7**

	chr13	chr14	chr15	chr16	chr17	chr18	chr19	chr20	chr21	chr22	chrX	chrY
RazerS3	120.5	127.1	124.8	136.2	152.7	104	128	97.2	54	68.7	197.6	36.4
Hobber3	12.3	13	12.6	13	13.8	11.7	12.7	11.5	9.6	10.2	15.1	9
CORAL	29.9	34.6	35.9	38.2	40.5	30.4	37.4	33.9	23.5	30.1	41.7	17.5
REPUTE	25.1	26.7	28.6	30	22.1	27.9	27.9	24.1	17.9	21	32.1	15.5
PLEDGER-cpu	26.4	27.4	27.3	28.9	29.6	24.5	27.5	25.3	20.1	22.3	30.7	16.7
PLEDGER-all	**20.4**	**21.4**	**21.1**	**21.9**	**22.5**	**19**	**21.1**	**19.5**	**15.6**	**17.6**	**24.7**	**13.5**

Table 5.2: Mapping time(in seconds) for 1M real reads,, of length $n = 150$ and error $\delta = 7$,, to chromosome (chr) 1-22, X and Y on system - 1 (CPU+GPU). PLEDGER-all distributes workload over CPU and GPU in 4:1 ratio while others execute only on the CPU.

	chr1	chr2	chr3	chr4	chr5	chr6	chr7	chr8	chr9	chr10	chr11	chr12
RazerS3	487	434.2	359.5	321	318.5	319	253	237.9	259.3	244.8	272.4	162.6
Hobber3	40.6	34.4	32.4	30.8	31.3	31.4	32.4	30.6	30.5	31.1	31.4	31.7
CORAL	123.8	115.1	106.8	96.7	93	100.9	113.8	100.3	100.7	104.4	93.7	100
REPUTE	79	70.8	62.9	56.5	62.6	58.8	65.9	57.2	56.2	58.3	55.6	60.2
PLEDGER-cpu	72.4	71.4	66.9	61.8	63.4	63.8	65.6	58.8	58.5	60.5	58.7	61.8
PLEDGER-all	**61.5**	**56.4**	**51.9**	**47.9**	**49.1**	**48.9**	**50.7**	**45.3**	**44.8**	**46.5**	**45.7**	**48.1**

	chr13	chr14	chr15	chr16	chr17	chr18	chr19	chr20	chr21	chr22	chrX	chrY
RazerS3	162.6	175.4	177.8	175.9	231.5	129	210.4	137.9	71.8	105.7	280.9	43.5
Hobber3	28.5	29.4	29.6	29.8	31	28	30.1	28.1	26.4	27.4	31.4	25.5
CORAL	75.4	89.8	91	98.6	105.3	74.4	102.6	80.4	51.3	75.7	102.6	31.7
REPUTE	40.3	48.1	48.8	53	60.3	39.6	56.8	41.9	29.9	75.7	102.6	31.7
PLEDGER-cpu	48.4	51.8	52.4	54.1	58	41.4	55.1	44.9	34.9	41.7	57.4	28.6
PLEDGER-all	**37.7**	**39.6**	**40.7**	**43.3**	**46.3**	**34.6**	**44.8**	**36.1**	**28.3**	**33.6**	**46.3**	**23.6**

Table 5.3: Mapping time(in seconds) for 1M real reads, of length $n = 100$ and error $\delta = 5$, to chromosome (chr) 1-22, X and Y on system - 2, Odroid N2 platform.

	chr1	chr2	chr3	chr4	chr5	chr6	chr7	chr8	chr9	chr10	chr11	chr12
Hobber3	88.5	82.1	73.63	69.8	75.5	62.8	65.1	57.5	56.5	59.3	56.8	59.8
PLEDGER	168.8	162.8	151.2	148.2	146.7	144	147.4	137.4	134.7	140.7	135.7	138.7
	chr13	chr14	chr15	chr16	chr17	chr18	chr19	chr20	chr21	chr22	chrX	chrY
Hobber3	48.1	63.6	50	52	58	48.4	50.9	45.9	37.3	40.8	59.8	35.4
PLEDGER	119.6	123.2	122.7	125.9	128.5	111.1	119.8	113.8	91.8	99.7	140.4	79.2

Table 5.4: Mapping time(in seconds) for 1M real reads,, of length $n = 150$ and error $\delta = 7$,, to chromosome (chr) 1-22, X and Y on system - 2, Odroid N2 platform.

	chr1	chr2	chr3	chr4	chr5	chr6	chr7	chr8	chr9	chr10	chr11	chr12
Hobber3	134.6	129	101.8	97.2	98	97.5	99.7	92.7	92.5	94.7	93.1	96.2
PLEDGER	339.2	313	290	273.6	278.1	279.3	286.4	258.6	256.3	264.9	257.4	269.5
	chr13	chr14	chr15	chr16	chr17	chr18	chr19	chr20	chr21	chr22	chrX	chrY
Hobber3	84.4	86.4	86.8	88.8	96.5	84.8	90.2	82.1	75.1	81	94.9	71.3
PLEDGER	216.8	229	230.6	242.6	258.1	201.3	242.7	207.4	159.6	186.4	262.9	133.3

Table 5.5: Energy consumption in accordance with Section 5.4.4.

	$n = 100, \delta = 5$		$n = 150, \delta = 7$	
	P(W)	E(J)	P(W)	E(J)
	System 1 - 20 W (Idle power)			
Hobbes3	79	20006.9	80	44028
PLEDGER-cpu	78	41035	79	78605.7
PLEDGER-all	113	51205.8	114	98859.8
	System 2 - 3 W (Idle power)			
Hobbes3	6.6	4611.8	6.6	7422.69
PLEDGER	6	9396	6.1	18404.7

5.4.3 Accuracy

Due to limited RAM capacity, all mappers are configured to report 100 locations per read. As mentioned in Section 5.3, to determine the accuracy of mapped reads we adopt a method similar to *any-best* scenario of the Rabema benchmark [137], where each reported location for a read is compared to those reported by RazerS3, the gold standard, for the same read. If any of the location and strand matches to the gold standard, we report it as an accurate match. In our experiments, we have found that all mappers produced over 99% accuracy in reporting locations in comparison to the gold standard.

5.4.4 Power and energy consumption

We compare the power and energy utilization of Hobbes3 and PLEDGER on system 1 and system 2. We measure the average power consumption during the mapping process and deduct the idle power to measure the power consumption during mapping process. To measure energy consumption, we multiply the power consumption with the total mapping time for 24 chromosomes. Table 5.5 presents the energy measurements on system 1 and system 2. We observe that using embedded platform for read mapping can lead to 4.34-5.93$\times$ energy savings compared to general purpose computers. It is, also,

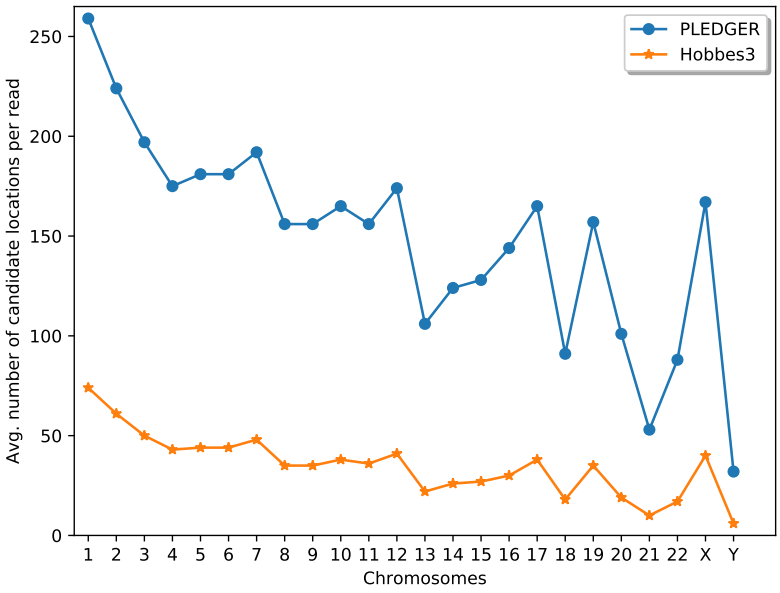

Figure 5.4: Comparison between average number of candidate locations per read verified by PLEDGER and Hobbes3. Values presented for $n = 150$ and $\delta = 7$

.

evident that high performance can directly yield huge energy savings in the embedded scenario.

5.4.5 Performance gap and future work

Hobbes3 and PLEDGER both use DP based filtration methodology to minimize the number of candidate locations. In principle, both DP based approaches evaluate the read to select similar optimum *k-mers*, however, Hobbes3 applies additional heuristic optimization to exclude multiple verification of similar candidate locations obtained from multiple *k-mers*. One such optimization is to divide read in $\delta + 2$ non-overlapping *k-mers* rather than, traditionally used, $\delta + 1$ *k-mers*. In this scenario, only those candidate locations which are found in at least two *k-mers* are verified. This significantly improves specificity of identifying candidate locations. PLEDGER, presently, does not use any post-filtration optimizations which leads to many more verification cycles than needed. Fig. 5.4 visualizes the average number of candidate locations, per read per chromosome,

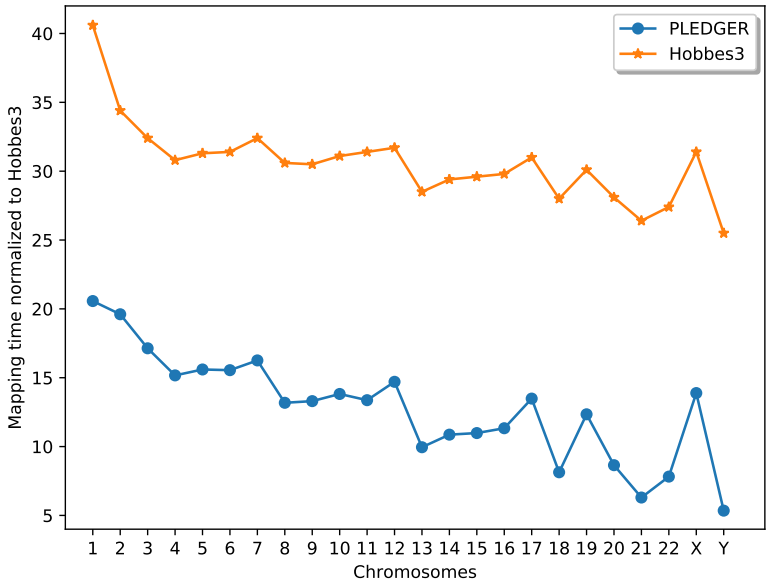

Figure 5.5: Linear normalization of mapping times for PLEDGER if it reports same number of candidate locations as Hobbes3. Values presented for $n = 150$ and $\delta = 7$.

verified by PLEDGER compared to Hobbes3. We can see that PLEDGER verifies 3-5× more locations per read which leads to longer mapping times. Fig. 5.5 presents a linear estimation of mapping time taken by PLEDGER if it had to verify the same number of candidate locations as Hobbes3. Although, computation times do not necessarily scale linearly, it demonstrates significant scope of improving performance. In our future work, we intend to append our filtration scheme with post-filtration optimizations to increase the specificity of selection of candidate locations.

5.4.6 Challenges with OpenCL

OpenCL imposes restrictions on the memory such that no variable can be allocated more than $1/4^{th}$ of the total RAM on the device. Thus, the data structures should fit within the limited window of available memory which inspired us to compress the FM-Index auxiliary data structures . It does not allow dynamic memory allocation which leads to inefficient use of memory while reporting mapping locations per read. For example, all reads will be allocated the same amount of memory to store the mapping locations even

if the locations reported for one read is much smaller than the other. Additionally, it is sometimes difficult to find and install the OpenCL driver of a device as some vendors may not make it open source.

5.5 Summary

We present a Pyopencl based tooL for gEnomic workloaDs tarGeting Embedded platforms (PLEDGER). PLEDGER is a stand-alone tool capable of generating data structures and mapping reads to the whole human genome in an embedded environment with limited available RAM capacity, as low as 3.6 GB. Its an automated tool that provides option of mapping reads to a selected or all the chromosomes of an organism. The underlying OpenCL framework supports portability across different devices enabling parallel kernel executions on multiple devices, such as CPU, GPU, simultaneously. It uses memory-aware data structures and algorithm-hardware co-design to target embedded scenarios for energy efficiency. It uses bit-vector operations and memory optimized dynamic programming based algorithm to accelerate the mapping process.We compare PLEDGER with state-of-the-art read mappers and demonstrate significant performance gains and energy savings. PLEDGER is first of its kind implementation that maps real reads to the entire human genome. With continuous growth in genomic data due to high-throughput sequencing, embedded genomics supported by PyopenCL framework, as presented by PLEDGER, can play a key role in translation genomics and next-generation medicine.

Chapter 6

Conclusion

6.1 Contributions

In this research, we aim to provide an energy-efficient and performance-driven novel hardware solution to the read mapping problem. Our approach towards solving the problem is as follows:

- We study the existing mappers and their target devices to understand the bottlenecks to efficient utilisation of available hardware resources in modern heterogeneous architectures.
- To target heterogeneous architectures, which often includes CPU and GPU together in most modern workstations and laptops, a method with cross-platform flexibility and low-programming effort is required. The aim is to distribute workload, without any additional effort, concurrently on different devices for speedups
- The advent of Internet-of-Things (IoT) has led to emergence of a variety of embedded platforms for low-power and low-energy requirements. They have limited memory capacity but are capable of battery powered operations and have sufficient compute capacity to run heavy smartphone workloads, making them worthy of exploration for genomic pipelines.

Most modern workstations, laptops and even smartphones are equipped with GPUs alongside CPU to accelerate parallel operations. To mitigate the extra programming effort required to program GPUs and, yet, use its availability, we develop a novel cross-platform read mapper, called CORAL, to target heterogeneous systems which can utilise all available hardware resources in a system. CORAL is based on OpenCL and is capable of parallel kernel executions on CPU and GPU, concurrently, distributing the workload judiciously in task-parallel fashion to achieve speedups. It can execute on any number of OpenCL conformant CPUs and GPUs on the same platform, simultaneously, as per user-specified workload distribution and collects the mapping result to a single file. CORAL used FM-Index and suffix array based data structures to preprocess and store the reference genome. It uses FM-Index backward search to implement a verification-aware filtration methodology which aims to reduce the total number of candidate locations that are verified using expensive DP-based banded Myers bit-vector algorithm. CORAL is validated by mapping both real and simulated reads to chr2 and chr21 of the human genome and is compared with state-of-the-art read mappers such as RazerS3, Hobbes3, FEM, BWA-MEM, GEM and Yara. CORAL is an *all-mapper* and outperforms *best-mappers* such as BWA-MEM both in performance and accuracy and GEM in accuracy. It, also, outperforms RazerS3 and Yara in performance and FEM in accuracy, while performs competitive against Hobbes3. The kernel was written in `C` with host code in `Python` and executions using `PyOpenCL` libraries for rapid modifications and prototyping. CORAL demonstrates that read mapping is more suitable for CPUs than GPUs unless GPU-specific implementation is adopted, however, outsourcing smaller workload to GPU in task parallel fashion can provide 2$\times$ or more speedups depending on the GPU architecture.

With the advent of IoT and its, direct and indirect, widespread adoption in most applications related to human society worldwide, tremendous effort has been put in developing low-power and energy-efficient embedded systems. Embedded systems, usually, have many-core architecture with low-complexity processing cores, often, accompanied with smaller and simpler GPUs. Genomic computational pipelines, generally, includes string searching and matching algorithms, which use integer based operations and, hence, may underutilise complex processors optimised for handling

floating-point operations. With this hypothesis, we propose an OpenCL-based read mapper for heterogeneous systems, called REPUTE, with focus on mapping reads on embedded platforms. REPUTE posits two major contributions: first, a dynamic programming (DP) based filtration method is proposed using FM-Index backward search for improved performance, and second, using algorithm-hardware co-design REPUTE is tailored for embedded platforms with limited memory. DP based filtration methodology implemented in REPUTE is among the most efficient methods available that selects seeds with lowest cumulative frequency. This reduces the number of candidate locations to be verified, therefor, reducing mapping times. Limited memory on embedded boards are the major bottleneck in using them for memory intensive genomic pipelines. REPUTE uses algorithm-hardware co-design to optimise the kernel for low-memory footprint so that more work-items (or threads) can run, concurrently, on ARM cores with smaller register capacity compared to complex processors. REPUTE inherits cross-platform flexibility from CORAL and can execute on multiple devices, simultaneously, on the same platform. REPUTE shows improved performance and accuracy when using CPU, Nvidia GPUs and HiKey 970 platform. It not only outperforms CORAL but, also, other state-of-the-art mappers including Hobbes3 using real reads demonstrating up to 13$\times$ speedups. On Hikey 970 embedded platform, REPUTE demonstrated up to 27$\times$ energy savings compared to workstation with Intel Core i7-2600 CPU and 2$\times$ Nvidia GeForce GTX 590.

REPUTE demonstrates that genomic computational pipelines, tagged as memory intensive applications, can be done on embedded platforms obtaining significant energy savings with competitive performance. However, the reads were mapped to a small chromosome, chr21, because the data structures to store larger chromosomes require higher memory capacity. This is major bottleneck to bringing whole genome mapping to embedded systems. To address this, we propose a `PyOpenCL` based tool for genomic workloads targeting embedded platforms (PLEDGER). PLEDGER is a memory-aware whole genome read mapping implementation which proposes generation of low memory footprint (LMF) data structures at the preprocessing stage. Using algorithm-hardware co-design, it re-constructs the REPUTE kernel to efficiently use the LMF data structures and mitigate the additional computational burden using bit-vector operations.

It implements variable level optimisations to minimise the memory footprint of the kernel. Additionally, PLEDGER is completely automated tool which maps the given reads to user selected chromosomes with options of mapping it all or one chromosome segregates the output files in separate folders. PLEDGER demonstrates, for the first time, that entire genome can be mapped on a single Odroid N2 board with $<$3.6 GB available RAM. It demonstrates up to 11$\times$ speedups compared to state-of-the-art mappers and energy savings of 5.9$\times$ compared to computing system with Intel i7-8750H CPU and Nvidia GTX 1050 Ti.

In summary, this thesis presents three whole genome read mapping tools viz. CORAL, REPUTE and PLEDGER. These three tools are build progressively with the aim of providing an energy-efficient and performance-driven novel hardware solution to the reassembly pipeline of the whole genome sequencing. PLEDGER is the latest version of the tool set that target embedded platforms and attempts to mitigate limitations of other hardware platforms such as FPGA and GPU, yet, retaining the advantages they provide such as speedups and energy savings. A major advantage of using an embedded platform is its low cost and negligible maintenance requirements. They do not require any cooling system which can result in saving huge electricity bills. The tools have been validated against state-of-the-art tools used by geneticists and bioinformaticians demonstrating speedups and high accuracy.

6.2 Future Work

In the following paragraphs we discuss possible directions of future work to enable translational genomics by providing an energy-efficient and performance-driven hardware solution to computational pipelines of genomics.

Chapter 3 presents an OpenCL-based read mapper implementing efficient heuristic for filtration to target heterogeneous architecture of modern computing machines with cross-platform capabilities. It presents a case where both CPU and GPU can be used together with minimal programming effort to obtain speedups. However, there is a scope of significant improvements in the filtration approach which is implemented in REPUTE, presented in Chapter 4. REPUTE proposes a dynamic programming based

filtration approach with theoretical background of providing best possible selection of seeds resulting in fewer candidate locations, overall. This results in fewer verification cycles leading to reduction in mapping time. It, also, presents an optimized kernel with low-memory footprint suited for embedded platforms. However, there remains two major shortcomings in the approach with scope of further improvements in mapping time. First, the memory footprint of the data structures for bigger chromosomes are prohibitive and they cannot be mapped on an embedded platform with limited memory. Second, there are many candidate locations which are verified more than once for the same read. Multiple verification for same candidate locations happen because multiple *k-mers* adjacently generate same candidate locations corresponding to their respective positions in the read. It happens when two or more *k-mers* are free of error.

PLEDGER, proposed in Chapter 5, mitigates the first shortcoming of REPUTE. It proposes a novel proprocessing methodology that, significantly, reduces the memory footprint of the data structures and is able to map to all the chromosomes in human genome. This, however, increases the filtration time because of the additional computational burden of obtaining the missing information in new data structure. To mitigate performance degradation, PLEDGER uses an additional auxiliary array and employs bit-vector operations for fast retrieval of information missing in the data structure. It produces competitive timings and, in most cases, outperforms REPUTE. The second shortcoming remains unresolved in both REPUTE and PLEDGER, where multiple verification is performed for the same candidate location generated by adjacent *k-mers*. Hobbes3 [16] prevents extra verification by obtaining candidate locations for $(\delta + 2)$ *k-mers* and verifying those locations which appear at least twice by pruning through the entry list in the hash table. FEM [27], also, practices a similar method and employ binary search method to find a copy of candidate location in the list and verifies, only, if it appears at least twice. This method prevents multiple verification of the same candidate location and increases the selectivity of these mappers by reducing fast positives. A similar approach can be integrated with PLEDGER to obtain further speedups.

This thesis presents a working prototype of mapping entire genome on a single embedded platform. However, genome reassembly is the first computational pipeline

of WGS. It is followed by downstream analysis of genomic data. Downstream analysis is increasingly dominating the computational requirements as large number of genomes are being sequenced. Geneticists and bioinformaticians are working towards making sense of massive amounts of data to translate it into improvements in diagnosis and therapy of genetic disorders. Among others, analysis includes variant calling such as SNP and indel or deletions and duplications [82, 153], genotype calling and annotations [154]. Assembling genome without follow-up analysis renders the data useless and, therefore, there is a need to integrate reassembly pipeline with analysis to deliver meaningful insights. SWARAM [22] is a recent attempt, along the same lines, where state-of-the-art existing mapping and analysis algorithms, such as BWA-MEM, Platypus [151] and/or GATK HaplotypeCaller [152], are stitched together and accelerated on an embedded system cluster. SWARAM divides the human genome depending on the number of devices available and then distributes the workload keeping a track of the memory available on board. Designing an optimised embedded genomics solution for the entire computational pipeline of WGS targeting an application requires attention. This thesis does not report mapping output in the form of SAM output format which would require backtracking after application of semi-global dynamic programming to produce the CIGAR string [155]. CIGAR string shows the sequence and position of each match, substitution, insertion, and deletion for the read with respect to the selected mapping location of the reference. The output files produced by CORAL, REPUTE and PLEDGER report the edit distance upon alignment, mapping location, and the strand which is similar to the PAF format of the Minimap2 [124] and can be used in many genomic analysis pipelines such as metagenomics [142–145] except for variant calling. The embedded genomic scenario discussed earlier may mitigate the need of SAM format as mapped reads can be analysed directly without intermediary format of storage.

The advent of third generation of sequencing technology, which employs single-molecule technology and produces long reads with different error profiles, will require novel computational approaches. These reads are produced at very low-cost compared to NGS short reads, however, they have higher error rates. Long reads have an important advantage of spanning large portions of repeats which dominate genomes of larger and complex species. Third generation sequencing technology holds great potential

in providing affordable healthcare with opportunities for application of embedded genomics.

Bibliography

[1] M. J. Chaisson, R. K. Wilson, and E. E. Eichler, "Genetic variation and the de novo assembly of human genomes," *Nature Reviews Genetics*, vol. 16, no. 11, pp. 627–640, 2015.

[2] D. Weese, M. Holtgrewe, and K. Reinert, "Razers 3: Faster, fully sensitive read mapping," *Bioinformatics*, vol. 28, no. 20, pp. 2592–2599, 2012.

[3] L. Hood and D. Galas, "P4 medicine: Personalized, predictive, preventive, participatory a change of view that changes everything," *Computing community consortium*, 2008.

[4] L. Hood, "Systems biology and p4 medicine: past, present, and future," *Rambam Maimonides Med J*, vol. 4, no. 2, Apr 2013.

[5] M. Flores, G. Glusman, K. Brogaard, N. D. Price, and L. Hood, "P4 medicine: how systems medicine will transform the healthcare sector and society," *Personalized medicine*, vol. 10, no. 6, p. 565—576, 2013.

[6] "Genomics for everyone: Prepairing you for the revolution in healthcare," 2021. [Online]. Available: http://genomicsforeveryone.org/genomics-and-p4-medicine-ethics-and-policy/

[7] A. of Medical Sciences, "Stratified, personalised or p4 medicine: a new direction for placing the patient at the centre of healthcare and health education," 2015.

[8] E. R. B. Mccabe, "Translational genomics in medical genetics," *Genet Med*, vol. 4, pp. 468–471, 2002.

[9] E. Zeggini, A. L. Gloyn, A. C. Barton, and L. V. Wain, "Translational genomics and precision medicine: Moving from the lab to the clinic," *Science*, vol. 365, no. 6460, pp. 1409–1413, 2019. [Online]. Available: https://science.sciencemag.org/content/365/6460/1409

[10] "Dna sequencing costs: Data," 2021. [Online]. Available: https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data

[11] S. D. Kahn, "On the future of genomic data," *Science*, vol. 331, no. 6018, pp. 728–729, Feb 2011.

[12] R. M. Ward, R. Schmieder, G. Highnam, and D. Mittelman, "Big data challenges and opportunities in high-throughput sequencing," *Systems Biomedicine*, vol. 1, no. 1, pp. 29–34, 2013. [Online]. Available: https://doi.org/10.4161/sysb.24470

[13] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: Astronomical or genomical?" *PLOS Biology*, vol. 13, no. 7, pp. 1–11, 07 2015.

[14] "The genomic data challenges of the future," 2018. [Online]. Available: https://medicalfuturist.com/the-genomic-data-challenges-of-the-future/

[15] R. M. Ward, R. Schmieder, G. Highnam, and D. Mittelman, "Big data challenges and opportunities in high-throughput sequencing," *Systems Biomedicine*, vol. 1, no. 1, pp. 29–34, 2013.

[16] J. Kim, C. Li, and X. Xie, "Hobbes3: Dynamic generation of variable-length signatures for efficient approximate subsequence mappings," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, 2016, pp. 169–180.

[17] B. Langmead and A. Nellore, "Cloud computing for genomic data analysis and collaboration," *Nature Reviews Genetics*, vol. 19, no. 4, pp. 208–219, 2018.

[18] W. Van Heddeghem, S. Lambert, B. Lannoo, D. Colle, M. Pickavet, and P. Demeester, "Trends in worldwide ict electricity consumption from 2007 to 2012," *Comput. Commun.*, vol. 50, p. 64–76, Sep. 2014. [Online]. Available: https://doi.org/10.1016/j.comcom.2014.02.008

[19] "How much energy do data centers really use?" 2020. [Online]. Available: https://energyinnovation.org/2020/03/17/how-much-energy-do-data-centers-really-use/

[20] A. Shehabi, S. J. Smith, D. A. Sartor, R. E. Brown, M. Herrlin, J. G. Koomey, E. R. Masanet, N. Horner, I. L. Azevedo, and W. Lintner, "United states data center energy usage report. no. lbnl-1005775. lawrence berkeley national lab.(lbnl), berkeley, ca (united states)," Tech. Rep., 06/2016 2016. [Online]. Available: https://go.nature.com/ejg7sr

[21] N. Jones, "How to stop data centres from gobbling up the world's electricity," *Nature*, vol. 561, pp. 163–166, September 2018.

[22] R. P. Mohanty, H. Gamaarachchi, A. Lambert, and S. Parameswaran, "Swaram: Portable energy and cost efficient embedded system for genomic processing," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3358211

[23] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 69:1–69:35, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2788396

[24] E. Siragusa, "Approximate string matching for high-throughput sequencing," *Free University of Berlin*, 2015.

[25] H. Li and R. Durbin, "Fast and accurate long-read alignment with burrows–wheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, 2010.

[26] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature methods*, vol. 9, no. 4, pp. 357–359, 2012.

[27] H. Zhang, Y. Chan, K. Fan, B. Schmidt, and W. Liu, "Fast and efficient short read mapping based on a succinct hash index," *BMC bioinformatics*, vol. 19, no. 1, p. 92, 2018.

[28] S. Marco-Sola, M. Sammeth, R. Guigo, and P. Ribeca, "The gem mapper: fast, accurate and versatile alignment by filtration," *Nature Methods*, vol. 9, pp. 1185–1188, 2012.

[29] M. Holtgrewe, "Mason–a read simulator for second generation sequencing data," *Technical Report FU Berlin*, 2010.

[30] S. Aluru and N. Jammula, "A review of hardware acceleration for computational genomics," *IEEE Design Test*, vol. 31, no. 1, pp. 19–30, Feb 2014.

[31] E. Kuzemchak, "You don't have to give up performance to have energy-efficient embedded systems," 2019. [Online]. Available: https://softwaredesignsolutions.com/blog/you-dont-have-to-give-up-performance-to-have-energy-efficient-embedded-systems/

[32] G. Khronos. (2018) List of opencl conformant products. https://www.khronos.org/conformance/adopters/conformant-products/opencl. Accessed: 2018-08-12.

[33] J. F. Degner, J. C. Marioni, A. A. Pai, J. K. Pickrell, E. Nkadori, Y. Gilad, and J. K. Pritchard, "Effect of read-mapping biases on detecting allele-specific expression from rna-sequencing data," *Bioinformatics*, vol. 25, no. 24, pp. 3207–3212, 2009.

[34] Y. Gilad, J. K. Pritchard, and K. Thornton, "Characterizing natural variation using next-generation sequencing technologies," *Trends in Genetics*, vol. 25, no. 10, pp. 463–471, 2009.

[35] R. Vijaya Satya, N. Zavaljevski, and J. Reifman, "A new strategy to reduce allelic bias in rna-seq readmapping," *Nucleic acids research*, vol. 40, no. 16, pp. e127–e127, 2012.

[36] K. Reinert, B. Langmead, D. Weese, and D. J. Evers, "Alignment of next-generation sequencing reads," *Annual Review of Genomics and Human Genetics*, vol. 16, no. 1, pp. 133–151, 2015, pMID: 25939052.

[37] E. L. van Dijk, Y. Jaszczyszyn, D. Naquin, and C. Thermes, "The third revolution in sequencing technology," *Trends in Genetics*, vol. 34, no. 9, pp. 666–681, 2018.

[38] A. Philippidis, "Top 10 sequencing companies," 2018. [Online]. Available: https://www.genengnews.com/a-lists/top-10-sequencing-companies-2/

[39] M. L. Metzker, "Sequencing technologies—the next generation," *Nature reviews genetics*, vol. 11, no. 1, pp. 31–46, 2010.

[40] "What is pcr (polymerase chain reaction)?" 2016. [Online]. Available: https://www.yourgenome.org/facts/what-is-pcr-polymerase-chain-reaction

[41] "Polymerase chain reaction (pcr) fact sheet." [Online]. Available: https://www.genome.gov/about-genomics/fact-sheets/Polymerase-Chain-Reaction-Fact-Sheet

[42] D. Branton, D. W. Deamer, A. Marziali, H. Bayley, S. A. Benner, T. Butler, M. Di Ventra, S. Garaj, A. Hibbs, X. Huang *et al.*, "The potential and challenges of nanopore sequencing," *Nanoscience and technology: A collection of reviews from Nature Journals*, pp. 261–268, 2010.

[43] D. Deamer, M. Akeson, and D. Branton, "Three decades of nanopore sequencing," *Nature biotechnology*, vol. 34, no. 5, pp. 518–524, 2016.

[44] N. Kono and K. Arakawa, "Nanopore sequencing: review of potential applications in functional genomics," *Development, growth & differentiation*, vol. 61, no. 5, pp. 316–326, 2019.

[45] X. Liao, M. Li, Y. Zou, F.-X. Wu, J. Wang *et al.*, "Current challenges and solutions of de novo assembly," *Quantitative Biology*, vol. 7, no. 2, pp. 90–109, 2019.

[46] J. K. Kulski, "Next-generation sequencing—an overview of the history, tools, and "omic" applications," *Next generation sequencing-advances, applications and challenges*, pp. 3–60, 2016.

[47] H. H. Kazazian, "Mobile elements: drivers of genome evolution," *science*, vol. 303, no. 5664, pp. 1626–1632, 2004.

[48] R. Cordaux and M. A. Batzer, "The impact of retrotransposons on human genome evolution," *Nature Reviews Genetics*, vol. 10, no. 10, pp. 691–703, 2009.

[49] N. V. Fedoroff, "Transposable elements, epigenetics, and genome evolution," *Science*, vol. 338, no. 6108, pp. 758–767, 2012.

[50] L. Schrader and J. Schmitz, "The impact of transposable elements in adaptive evolution," *Molecular Ecology*, vol. 28, no. 6, pp. 1537–1549, 2019.

[51] R. G. Hunter, "Stress, Adaptation, and the Deep Genome: Why Transposons Matter," *Integrative and Comparative Biology*, vol. 60, no. 6, pp. 1495–1505, 06 2020.

[52] J.-i. Sohn and J.-W. Nam, "The present and future of de novo whole-genome assembly," *Briefings in Bioinformatics*, vol. 19, no. 1, pp. 23–40, 10 2016.

[53] N. Nagarajan and M. Pop, "Sequence assembly demystified," *Nature Reviews Genetics*, vol. 14, no. 3, pp. 157–167, 2013.

[54] P. Green, "Against a whole-genome shotgun," *Genome Research*, vol. 7, no. 5, pp. 410–417, 1997.

[55] E. W. Myers and J. L. Weber, "Is whole human genome sequencing feasible?" in *Theoretical and Computational Methods in Genome Research*. Springer, 1997, pp. 73–89.

[56] M. Jain, S. Koren, K. H. Miga, J. Quick, A. C. Rand, T. A. Sasani, J. R. Tyson, A. D. Beggs, A. T. Dilthey, I. T. Fiddes *et al.*, "Nanopore sequencing and assembly of a human genome with ultra-long reads," *Nature biotechnology*, vol. 36, no. 4, pp. 338–345, 2018.

[57] J. T. Simpson and M. Pop, "The theory and practice of genome sequence assembly," *Annual review of genomics and human genetics*, vol. 16, pp. 153–172, 2015.

[58] S. Gladman, "De novo genome assembly for illumina data," 2021. [Online]. Available: https://www.melbournebioinformatics.org.au/tutorials/tutorials/assembly/assembly-protocol/

[59] J. D. Kececioglu and E. W. Myers, "Combinatorial algorithms for dna sequence assembly," *Algorithmica*, vol. 13, no. 1, pp. 7–51, 1995.

[60] P. A. Pevzner, H. Tang, and M. S. Waterman, "An eulerian path approach to dna fragment assembly," *Proceedings of the national academy of sciences*, vol. 98, no. 17, pp. 9748–9753, 2001.

[61] E. W. Myers, "The fragment assembly string graph," *Bioinformatics*, vol. 21, no. suppl_2, pp. ii79–ii85, 2005.

[62] M. Boetzer and W. Pirovano, "Toward almost closed genomes with gapfiller," *Genome biology*, vol. 13, no. 6, pp. 1–9, 2012.

[63] D. Paulino, R. L. Warren, B. P. Vandervalk, A. Raymond, S. D. Jackman, and I. Birol, "Sealer: a scalable gap-closing application for finishing draft genomes," *BMC bioinformatics*, vol. 16, no. 1, pp. 1–8, 2015.

[64] X. Huang, J. Wang, S. Aluru, S.-P. Yang, and L. Hillier, "Pcap: a whole-genome assembly program," *Genome research*, vol. 13, no. 9, pp. 2164–2170, 2003.

[65] S. Batzoglou, D. B. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J. P. Mesirov, and E. S. Lander, "Arachne: a whole-genome shotgun assembler," *Genome research*, vol. 12, no. 1, pp. 177–189, 2002.

[66] E. W. Myers, G. G. Sutton, A. L. Delcher, I. M. Dew, D. P. Fasulo, M. J. Flanigan, S. A. Kravitz, C. M. Mobarry, K. H. Reinert, K. A. Remington *et al.*, "A whole-genome assembly of drosophila," *Science*, vol. 287, no. 5461, pp. 2196–2204, 2000.

[67] S. Koren, M. C. Schatz, B. P. Walenz, J. Martin, J. T. Howard, G. Ganapathy, Z. Wang, D. A. Rasko, W. R. McCombie, E. D. Jarvis *et al.*, "Hybrid error correction and de novo assembly of single-molecule sequencing reads," *Nature biotechnology*, vol. 30, no. 7, pp. 693–700, 2012.

[68] C.-S. Chin, D. H. Alexander, P. Marks, A. A. Klammer, J. Drake, C. Heiner, A. Clum, A. Copeland, J. Huddleston, E. E. Eichler *et al.*, "Nonhybrid, finished microbial genome assemblies from long-read smrt sequencing data," *Nature methods*, vol. 10, no. 6, p. 563, 2013.

[69] R. Luo, B. Liu, Y. Xie, Z. Li, W. Huang, J. Yuan, G. He, Y. Chen, Q. Pan, Y. Liu *et al.*, "Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler," *Gigascience*, vol. 1, no. 1, pp. 2047–217X, 2012.

[70] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol, "Abyss: a parallel assembler for short read sequence data," *Genome research*, vol. 19, no. 6, pp. 1117–1123, 2009.

[71] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de bruijn graphs," *Genome research*, vol. 18, no. 5, pp. 821–829, 2008.

[72] J. Butler, I. MacCallum, M. Kleber, I. A. Shlyakhter, M. K. Belmonte, E. S. Lander, C. Nusbaum, and D. B. Jaffe, "Allpaths: de novo assembly of whole-genome shotgun microreads," *Genome research*, vol. 18, no. 5, pp. 810–820, 2008.

[73] A. Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S. Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski *et al.*, "Spades: a new genome assembly algorithm and its applications to single-cell sequencing," *Journal of computational biology*, vol. 19, no. 5, pp. 455–477, 2012.

[74] Y. Peng, H. C. Leung, S.-M. Yiu, and F. Y. Chin, "Idba-ud: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth," *Bioinformatics*, vol. 28, no. 11, pp. 1420–1428, 2012.

[75] J. Luo, J. Wang, W. Li, Z. Zhang, F.-X. Wu, M. Li, and Y. Pan, "Epga2: memory-efficient de novo assembler," *Bioinformatics*, vol. 31, no. 24, pp. 3988–3990, 2015.

[76] J. T. Simpson and R. Durbin, "Efficient construction of an assembly string graph using the fm-index," *Bioinformatics*, vol. 26, no. 12, pp. i367–i373, 2010.

[77] I. Ben-Bassat and B. Chor, "String graph construction using incremental hashing," *Bioinformatics*, vol. 30, no. 24, pp. 3515–3523, 2014.

[78] G. Gonnella and S. Kurtz, "Readjoiner: a fast and memory efficient string graph-based sequence assembler," *BMC bioinformatics*, vol. 13, no. 1, pp. 1–19, 2012.

[79] H. Dinh and S. Rajasekaran, "A memory-efficient data structure representing exact-match overlap graphs with application for next-generation dna assembly," *Bioinformatics*, vol. 27, no. 14, pp. 1901–1907, 2011.

[80] A. M. Kaye and W. W. Wasserman, "The genome atlas: Navigating a new era of reference genomes," *Trends in Genetics*, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168952520303280

[81] 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.

[82] M. Mielczarek and J. Szyda, "Review of alignment and snp calling algorithms for next-generation sequencing data," *Journal of applied genetics*, vol. 57, no. 1, pp. 71–79, 2016.

[83] J. Kim, M. Ji, and G. Yi, "A review on sequence alignment algorithms for short reads based on next-generation sequencing," *IEEE Access*, vol. 8, pp. 189 811–189 822, 2020.

[84] A. Ahmadi, A. Behm, N. Honnalli, C. Li, L. Weng, and X. Xie, "Hobbes: optimized gram-based methods for efficient read alignment," *Nucleic acids research*, vol. 40, no. 6, pp. e41–e41, 2012.

[85] H. Xin, S. Nahar, R. Zhu, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu, "Optimal seed solver: optimizing seed selection in read mapping," *Bioinformatics*, vol. 32, no. 11, pp. 1632–1642, 2016.

[86] 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. [Online]. Available: http://dx.doi.org/10.1137/0222058

[87] 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.

[88] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, Palo Alto, CA, Tech. Rep. 124, 1994.

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

[90] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "The enhanced suffix array and its applications to genome analysis," in *International Workshop on Algorithms in Bioinformatics*. Springer, 2002, pp. 449–463.

[91] G. Kucherov, L. Noe, and M. Roytberg, "Multiseed lossless filtration," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 2, no. 1, pp. 51–61, 2005.

[92] S. Burkhardt and J. Kärkkäinen, "Better filtering with gapped q-grams," in *Annual Symposium on Combinatorial Pattern Matching*. Springer, 2001, pp. 73–85.

[93] K. R. Rasmussen, J. Stoye, and E. W. Myers, "Efficient q-gram filters for finding all ϵ-matches over a given length," in *Research in Computational Molecular Biology*, S. Miyano, J. Mesirov, S. Kasif, S. Istrail, P. A. Pevzner, and M. Waterman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 189–203.

[94] 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*, vol. 48, no. 3, pp. 443–453, 1970.

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

[96] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of molecular biology*, vol. 162, no. 3, pp. 705–708, 1982.

[97] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *J. ACM*, vol. 46, no. 3, pp. 395–415, May 1999.

[98] H. Hyyrö, "A bit-vector algorithm for computing levenshtein and damerau edit distances," *Nordic J. of Computing*, vol. 10, no. 1, pp. 29–39, Mar. 2003.

[99] J. Grealey, L. Lannelongue, W.-Y. Saw, J. Marten, G. Meric, S. Ruiz-Carmona, and M. Inouye, "The carbon footprint of bioinformatics," *bioRxiv*, 2021.

[100] G. Alonso, "Fpgas in data centers: Fpgas are slowly leaving the niche space they have occupied for decades." *Queue*, vol. 16, no. 2, pp. 52–57, 2018.

[101] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, "Hardware acceleration of short read mapping," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012, pp. 161–168.

[102] N. Homer, B. Merriman, and S. F. Nelson, "Bfast: an alignment tool for large scale genome resequencing," *PloS one*, vol. 4, no. 11, p. e7767, 2009.

[103] J. Arram, K. H. Tsoi, W. Luk, and P. Jiang, "Reconfigurable acceleration of short read mapping," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, 2013, pp. 210–217.

[104] F. Zokaee, H. R. Zarandi, and L. Jiang, "Aligner: A process-in-memory architecture for short read alignment in rerams," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 237–240, 2018.

[105] V. Y. Gudur, S. Maheshwari, R. Shafik, and A. Acharyya, "Accelerated filtering and in situ verification for energy-optimized genome read mapping," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5.

[106] T. Robinson, J. Harkin, and P. Shukla, "Hardware acceleration of genomics data analysis: challenges and opportunities," *Bioinformatics*, 05 2021.

[107] J. Arram, T. Kaplan, W. Luk, and P. Jiang, "Leveraging fpgas for accelerating short read alignment," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 14, no. 3, pp. 668–677, 2017.

[108] M. Alser, J. Rotman, D. Deshpande, K. Taraszka, H. Shi, P. I. Baykal, H. T. Yang, V. Xue, S. Knyazev, B. D. Singer *et al.*, "Technology dictates algorithms: recent developments in read alignment," *Genome biology*, vol. 22, no. 1, pp. 1–34, 2021.

[109] Y. Liu, B. Schmidt, and D. L. Maskell, "CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform," *Bioinformatics*, vol. 28, no. 14, pp. 1830–1837, 05 2012.

[110] Y. Liu and B. Schmidt, "Cushaw2-gpu: Empowering faster gapped short-read alignment using gpu computing," *IEEE Design Test*, vol. 31, no. 1, pp. 31–39, 2014.

[111] A. M. Aji, L. Zhang, and W.-c. Feng, "Gpu-rmap: Accelerating short-read mapping on graphics processors," in *2010 13th IEEE International Conference on Computational Science and Engineering*, 2010, pp. 168–175.

[112] A. D. Smith, W.-Y. Chung, E. Hodges, J. Kendall, G. Hannon, J. Hicks, Z. Xuan, and M. Q. Zhang, "Updates to the RMAP short-read mapping software," *Bioinformatics*, vol. 25, no. 21, pp. 2841–2842, 09 2009.

[113] C.-M. Liu, T. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, R. Li, and T.-W. Lam, "SOAP3: ultra-fast GPU-based parallel alignment tool for short reads," *Bioinformatics*, vol. 28, no. 6, pp. 878–879, 01 2012.

[114] S. D. Goenka, Y. Turakhia, B. Paten, and M. Horowitz, "Segalign: A scalable gpu-based whole genome aligner," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–13.

[115] R. S. Harris, "Improved pairwise alignmnet of genomic dna," 2007.

[116] R. Wilton, X. Li, A. P. Feinberg, and A. S. Szalay, "Arioc: Gpu-accelerated alignment of short bisulfite-treated reads," *Bioinformatics*, vol. 34, no. 15, pp. 2673–2675, 2018.

[117] D. Newkirk, J. Biesinger, A. Chon, K. Yokomori, and X. Xie, "Arem: aligning short reads from chip-sequencing by expectation maximization," *Journal of Computational Biology*, vol. 18, no. 11, pp. 1495–1505, 2011.

[118] A. Roberts and L. Pachter, "Streaming fragment assignment for real-time analysis of sequencing experiments," *Nature methods*, vol. 10, no. 1, pp. 71–73, 2013.

[119] J. Kim, C. Li, and X. Xie, "Improving read mapping using additional prefix grams," *BMC bioinformatics*, vol. 15, no. 1, pp. 1–11, 2014.

[120] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[121] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped blast and psi-blast: a new generation of protein database search programs," *Nucleic acids research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[122] D. Weese, A.-K. Emde, T. Rausch, A. Döring, and K. Reinert, "Razers—fast read mapping with sensitivity control," *Genome research*, vol. 19, no. 9, pp. 1646–1654, 2009.

[123] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating read mapping with fasthash," in *BMC genomics*, vol. 14, no. 1. Springer, 2013, pp. 1–13.

[124] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 05 2018. [Online]. Available: https://doi.org/10.1093/bioinformatics/bty191

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

[126] 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.

[127] S. Thankaswamy-Kosalai, P. Sen, and I. Nookaew, "Evaluation and assessment of read-mapping by multiple next-generation sequencing aligners based on genome-wide characteristics," *Genomics*, vol. 109, no. 3, pp. 186–191, 2017.

[128] "Cuda toolkit," 2021. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[129] M. Scarpino, *OpenCL in Action: How to accelerate graphics and computations*. Manning Publications, 2011.

[130] "Nhs genomic medicine centres," 2021. [Online]. Available: https://www.genomicsengland.co.uk/about-genomics-england/the-100000-genomes-project/genomic-medicine-centres/

[131] A. Prakash, S. Wang, A. E. Irimiea, and T. Mitra, "Energy-efficient execution of data-parallel applications on heterogeneous mobile platforms," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, Oct 2015, pp. 208–215.

[132] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–10.

[133] M. Damschen, F. Mueller, and J. Henkel, "Co-scheduling on fused cpu-gpu architectures with shared last level caches," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2337–2347, Nov 2018.

[134] A. K. Singh, A. Prakash, K. R. Basireddy, G. V. Merrett, and B. M. Al-Hashimi, "Energy-efficient run-time mapping and thread partitioning of concurrent opencl applications on cpu-gpu mpsocs," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 147:1–147:22, Sep. 2017. [Online]. Available: http://doi.acm.org/10.1145/3126548

[135] B. Langmead. (2014) Teaching materials: video lectures. [Online]. Available: http://www.langmead-lab.org/teaching-materials/

[136] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, Haussler, and David, "The human genome browser at ucsc," *Genome Research*, vol. 12, no. 6, pp. 996–1006, 2002.

[137] M. Holtgrewe, A.-K. Emde, D. Weese, and K. Reinert, "A novel and well-defined benchmarking method for second generation read mapping," *BMC Bioinformatics*, vol. 12, no. 1, p. 210, May 2011.

[138] Y. Turakhia, S. D. Goenka, G. Bejerano, and W. J. Dally, "Darwin-wga: A co-processor provides increased sensitivity in whole genome alignments with high speedup," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 359–372.

[139] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read

mapping," *Bioinformatics*, vol. 33, no. 21, pp. 3355–3363, 05 2017. [Online]. Available: https://doi.org/10.1093/bioinformatics/btx342

[140] J. S. Kim, D. Senol Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "Grim-filter: Fast seed location filtering in dna read mapping using processing-in-memory technologies," *BMC Genomics*, vol. 19, no. 2, p. 89, May 2018. [Online]. Available: https://doi.org/10.1186/s12864-018-4460-0

[141] "clgetkernelworkgroupinfo manual page," 2021. [Online]. Available: https://www.khronos.org/registry/OpenCL/sdk/2.2/docs/man/html/clGetKernelWorkGroupInfo.html

[142] N. LaPierre, M. Alser, E. Eskin, D. Koslicki, and S. Mangul, "Metalign: efficient alignment-based metagenomic profiling via containment min hash," *Genome biology*, vol. 21, no. 1, pp. 1–15, 2020.

[143] N. LaPierre, S. Mangul, M. Alser, I. Mandric, N. C. Wu, D. Koslicki, and E. Eskin, "Micop: microbial community profiling method for detecting viral and fungal organisms in metagenomic samples," *BMC genomics*, vol. 20, no. 5, pp. 1–10, 2019.

[144] F. Meyer, A. Fritz, Z.-L. Deng, D. Koslicki, A. Gurevich, G. Robertson, M. Alser, D. Antipov, F. Beghini, D. Bertrand, J. J. Brito, C. Brown, J. Buchmann, A. Buluç, B. Chen, R. Chikhi, P. T. Clausen, A. Cristian, P. W. Dabrowski, A. E. Darling, R. Egan, E. Eskin, E. Georganas, E. Goltsman, M. A. Gray, L. H. Hansen, S. Hofmeyr, P. Huang, L. Irber, H. Jia, T. S. Jørgensen, S. D. Kieser, T. Klemetsen, A. Kola, M. Kolmogorov, A. Korobeynikov, J. Kwan, N. LaPierre, C. Lemaitre, C. Li, A. Limasset, F. Malcher-Miranda, S. Mangul, V. R. Marcelino, C. Marchet, P. Marijon, D. Meleshko, D. R. Mende, A. Milanese, N. Nagarajan, J. Nissen, S. Nurk, L. Oliker, L. Paoli, P. Peterlongo, V. C. Piro, J. S. Porter, S. Rasmussen, E. R. Rees, K. Reinert, B. Renard, E. M. Robertsen, G. L. Rosen, H.-J. Ruscheweyh, V. Sarwal, N. Segata, E. Seiler, L. Shi, F. Sun, S. Sunagawa, S. J. Sørensen, A. Thomas, C. Tong, M. Trajkovski, J. Tremblay, G. Uritskiy, R. Vicedomini, Z. Wang, Z. Wang, Z. Wang, A. Warren, N. P. Willassen, K. Yelick, R. You, G. Zeller, Z. Zhao, S. Zhu,

J. Zhu, R. Garrido-Oter, P. Gastmeier, S. Hacquard, S. Häußler, A. Khaledi, F. Maechler, F. Mesny, S. Radutoiu, P. Schulze-Lefert, N. Smit, T. Strowig, A. Bremges, A. Sczyrba, and A. C. McHardy, "Critical assessment of metagenome interpretation - the second round of challenges," *bioRxiv*, 2021. [Online]. Available: https://www.biorxiv.org/content/early/2021/07/12/2021.07.12.451567

[145] H. Gamaarachchi, S. Parameswaran, and M. A. Smith, "Featherweight long read alignment using partitioned reference indexes," *Scientific reports*, vol. 9, no. 1, pp. 1–12, 2019.

[146] W. V. Heddeghem, S. Lambert, B. Lannoo, D. Colle, M. Pickavet, and P. Demeester, "Trends in worldwide ict electricity consumption from 2007 to 2012," *Computer Communications*, vol. 50, pp. 64 – 76, 2014, green Networking. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0140366414000619

[147] V. Gnanasambandapillai, A. Bayat, and S. Parameswaran, "Mesga: An mpsoc based embedded system solution for short read genome alignment," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2018, pp. 52–57.

[148] S. Maheshwari, V. Y. Gudur, R. Shafik, I. Wilson, A. Yakovlev, and A. Acharyya, "Coral: Verification-aware opencl based read mapper for heterogeneous systems," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pp. 1–1, 2019.

[149] H. Ye, J. Meehan, W. Tong, and H. Hong, "Alignment of short reads: A crucial step for application of next-generation sequencing data in precision medicine," *Pharmaceutics*, vol. 7, no. 4, pp. 523–541, 2015.

[150] S. Maheshwari, R. Shafik, I. Wilson, A. Yakovlev, and A. Acharyya, "Repute: An opencl based read mapping tool for embedded genomics," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 121–126.

[151] A. Rimmer, H. Phan, I. Mathieson, Z. Iqbal, S. R. Twigg, A. O. Wilkie, G. McVean, and G. Lunter, "Integrating mapping-, assembly-and haplotype-based approaches for calling variants in clinical sequencing applications," *Nature genetics*, vol. 46, no. 8, pp. 912–918, 2014.

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

[153] D. Muzzey, E. A. Evans, and C. Lieber, "Understanding the basics of ngs: from mechanism to variant calling," *Current genetic medicine reports*, vol. 3, no. 4, pp. 158–165, 2015.

[154] R. Ekblom and J. B. Wolf, "A field guide to whole-genome sequencing, assembly and annotation," *Evolutionary applications*, vol. 7, no. 9, pp. 1026–1042, 2014.

[155] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin, "The sequence alignment/map format and samtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.