IMPROVING BWA-EM WITH GPU PARALLEL COMPUTING

Connor Li

Department of Computer Science

Submitted in partial fulfillment
of the requirement for the degree of

Master of Science

Faculty of Mathematics and Science, Brock University
St. Catharines, Ontario

Dedication

This dissertation/thesis is dedicated to Doctor Qiu and Doctor Liang who take me in as a master student and support during COVID 19

This dissertation/thesis is also dedicated to my mother and father who provided both emotional and financial support

I also give special thanks
to my friends being there for me during my master years

ABSTRACT

Due to the many advances made in designing algorithms, especially the ones used in bioinformatics, it is becoming harder and harder to improve their efficiencies. Therefore, hardware acceleration using General-Purpose computing on Graphics Processing Unit has become a popular choice. BWA-MEM is an important part of the BWA software package for sequence mapping. Because of its high speed and accuracy, we choose to parallelize the popular short DNA sequence mapper. BWA has been a prevalent single node tool in genome alignment, and it has been widely studied for acceleration for a long time since the first version of the BWA package came out. This thesis presents the Big Data GPGPU distributed BWA-MEM, a tool that combines GPGPU acceleration and distributed computing. The four hardware parallelization techniques used are CPU multi-threading, GPU paralleled, CPU distributed, and GPU distributed. The GPGPU distributed software typically outperforms other parallelization versions. The alignment is performed on a distributed network, and each node in the network executes a separate GPGPU paralleled version of the software.

The process of BWA-MEM, a popular short DNA sequence mapper, has five major stages, including 1) loading index and read files, 2) finding super-maximal exact matches (SMEM), 3) chaining and chain filtering, 4) seed extension, and 5) generating the output file. After a hot spot analysis, function chain2aln from stage four is found to have higher usage.

We parallelize the chain2aln function in three levels. In Level 1, the function ksw_extend2, an algorithm based on Smith-Waterman, is parallelized to handle extension on one side of the seed. In Level 2, the function chain2aln is parallelized to handle chain extension, where all seeds within the same chain are extended. In Level 3, part of the function mem_align1_core is parallelized for extending multiple chains. Due to the program's complexity, the parallelization work was limited at the GPU version of ksw_extend2 parallelization Level 3.

However, we have successfully combined Spark with BWA-MEM and ksw_extend2 at parallelization Level 1, which has shown that the proposed framework is possible. The paralleled Level 3 GPU version of ksw_extend2 demonstrated noticeable speed improvement with the test data set.

TABLE OF CONTENTS

Page

ABSTRACT ii

LIST OF TABLES ix

LIST OF FIGURES xi

LIST OF ALGORITHMS xii

CHAPTERS

1 INTRODUCTION 1

1.0.1 Major Contributions 4

2 BACKGROUND 5

2.1 DNA Sequencing 5

2.1.1 Sanger Sequencing 5

2.1.1.1 Capillary Sanger Sequencing 6

2.1.1.2 Microfluidic Sanger Sequencing 6

2.1.2 *NGS platforms* 7

2.1.2.1 Pyrosequencing (Roche/454) 7

2.1.2.2 Reversible Dye Terminator Sequencing (Illumina) 7

2.1.2.3 Ion Semiconductor Sequencing (Ion Torrent/Proton) . . . 8

2.1.2.4 PacBio Sequencing 8

2.1.2.5 Nanopore Sequencing 9

2.1.3 De-novo Sequencing Versus Re-sequencing 9

2.2 String Matching 10

2.2.1 Brute Force Approach 10

2.2.2 Dynamic Programming Algorithms 11

2.2.2.1 Edit Distance Computation 11

2.2.2.2 Text Searching with Edit Distance 14

2.2.3 Finite Automata Approach 15

2.2.4 Hashing Based Approach 17

2.2.5 Bit Parallel Approach 17

2.2.6 String Indexing 18

2.2.6.1 The Word Neighbourhood 19

2.2.6.2 Exact Partitioning 19

2.2.6.3 Intermediate Partitioning 20

2.3 BWA Package 20

2.3.1 Seed-and-extend Strategy 21

2.3.1.1 Type of Seeds 21

2.3.2 BWA-ALN 22

2.3.2.1 Burrows-Wheeler Transform 22

2.3.2.2 Suffix Arrays 24

2.3.2.3 Indexing 26

2.3.2.4 Exact and Approximate Matching 26

2.3.2.5 Alignment Determination 28

2.3.3 BWA-MEM 28

2.3.3.1 Indexing and File Loading 29

2.3.3.2 Seeding and Re-seeding 29

2.3.3.3 Chaining and Chain Filtering 30

2.3.3.4 Seed Extension 30

2.3.3.5 Output 31

2.4 Parallel Computing 31

2.4.1 Definition of Parallel Computing 32

2.4.2 Classification of Parallel Computing 32

2.4.3 CPU multi-threaded 33

2.4.4 CPU Distributed 33

2.4.4.1 Message Passing Interface 33
2.4.4.2 Big Data 34
2.4.4.3 GPU multi-threaded 34

3 RESEARCH DESIGN 39
3.1 Technical Road-Map 40
3.2 Hardware Setup 47
3.2.1 SharcNET 47
3.2.2 Personal PCs 53
3.2.3 Cloud Services 55
3.2.4 Brock University Department of Computer Science 55
3.2.5 Summary 55
3.3 Software Setup 56
3.3.1 SparkBWA 56
3.4 BWA-MEM GPGPU Parallelization 57
3.4.1 Hot Spot Analysis 57
3.4.2 Smith-Waterman Algorithm 59
3.4.2.1 Algorithm 60
3.4.2.2 Substitution Matrix 61
3.4.2.3 Gap Penalty 62
3.4.2.4 Smith-Waterman Algorithm Parallelization 63
3.4.3 ksw_extend2 66
3.4.3.1 Pruning Optimization 73
3.4.4 Program Design ksw_extend2 Parallelization 74
3.4.4.1 *Level 1* Parallelization: Seed Extension 76
3.4.4.2 *Level 2* and *Level 3* Parallelization: Chaining and Chain Filtering 76
3.4.5 ksw_extend2 with Time-saving Version Implementation 79
3.4.5.1 Phase 1: Alignment Collection and Preparation 80
3.4.5.2 Phase 2: Calculation of Alignment Scores 82
3.4.5.3 Phase 3: Production of Profiles as Output of Results . . . 88

3.4.6 ksw_extend2 with Memory-saving Version Implementation 88

4 TEST RESULTS 90

4.1 The Generation of the Performance Data 91

4.2 Level 1 Parallelization 93

4.2.1 Test Data Set Generation with Random Number Generator for Parallelization *Level 1* 94

4.2.2 Test 1: Sequence Alignment Similarity and Time Cost without the pruning mechanism 96

4.2.3 Test 2: The Effect of the Pruning Mechanism Towards CPU and GPGPU versions Alignment Performance 98

4.2.4 Test 3: The Effect of the Pruning Mechanism towards CPU and GPGPU Versions Alignment Performance with Longer Sequence . 101

4.2.5 Summary 103

4.3 Parallelization Level 2 and Level 3 104

4.3.1 How the Sequence Data has been Generated and Used in Parallelization Level 2 and Level 3 105

4.3.2 Test 1: Performance Comparisons Among Different Implementations at Parallelization Level 2 and Level 3 of ksw_extend2 with Alignments length at 8 bp 107

4.3.3 Test 2: Performance Comparisons Among Different Implementations at Parallelization Level 2 and Level 3 of ksw_extend2 with Alignments length at 16 bp 109

4.3.4 Test 3: Performance Comparisons Among Different Implementations at Parallelization Level 2 and Level 3 of ksw_extend2 with Alignments length at 32 bp 111

4.3.5 Summary 113

4.4 GPGPU distributed BWA-MEM 114

4.4.1 GPGPU Distributed Framework 114

4.4.2 Big Data 114

4.4.2.1 Hadoop 115

4.4.2.2 Apache Spark 115
4.4.2.3 SparkBWA Analyzation 116
4.4.2.4 Summary 117
4.4.3 Implementation 117
4.5 Summary 120

5 DISCUSSION 121
5.1 Rationale and Objective 121
5.2 Result 123
5.3 Theoretical Running Time 125
5.4 Future Work 126

6 APPENDIX EXPLANATION 129

REFERENCES 135

APPENDIX

A Appendix Explanation 137

B List of Test Output 138
B.1 Parallelization Level 1 Test 1 138
B.2 Parallelization Level 1 Test 2 140
B.3 Parallelization Level 1 Test 3 142
B.4 Parallelization Level 2 and Level 3 Test 1 144
B.5 Parallelization Level 2 and Level 3 Test 2 146
B.6 Parallelization Level 2 and Level 3 Test 3 148

C ksw_extend2 GPU Version 149
C.1 definitions.h 149
C.2 typedefs.h 150
C.3 gpuAlign.h 152
C.4 gpuAlign.cu 155
C.5 smithwaterman.h 162
C.6 smithwaterman.cu 166
C.7 ksw_extend2CPU.c 182

LIST OF TABLES

2.1 All Rotations of $T = \$banada$ 23

2.2 A lexicographically Sorted BWT Matrix Given an Input String $T = \$banada$ 23

3.1 Types of Parallelization Techniques 40

3.2 Types of Parallelization Techniques 41

3.3 Description of the Simplified Version of big data *GPGPU distributed BWA-MEM* framework 46

3.4 Hardware Specification for Graham Cluster 48

3.5 The Hardware Specification for Mosaic Cluster. 49

3.6 Software Specification for Graham Cluster. 50

3.7 Software Setup Process on Mosaic Cluster 51

3.8 Output of Software Setup on Mosaic Cluster 52

3.9 SparkBWA Environment Setup on Mosaic Cluster. 53

3.10 Offline PC 1's Specification 54

3.11 Offline PC 2's Specification 54

3.12 SparkBWA Environment Setup on PC 1 56

3.13 An Example of the Subsection Matrix 61

3.14 Seed Extension in *ksw_ extend2* 67

3.15 Time Complexity of GPGPU and CPU version 74

4.1 CPU and GPGPU versions alignment performance comparisons without the pruning mechanism with different sequence similarity levels at parallelization *Level 1*. 96

4.2 CPU and GPGPU versions alignment performance comparisons with or without the pruning mechanism at parallelization *Level 1*. 99

4.3 CPU and GPGPU versions alignment performance comparisons with or without the pruning mechanism up to *7416 bp* length at parallelization *Level 1*. . 102

4.4 Performance comparisons among different implementations at parallelization *Level 2* and *Level 3* of *ksw_ extend2* with alignments length at *8 bp*. 107

4.5 Performance comparisons among different implementations at parallelization *Level 2* and *Level 3* of *ksw_ extend2* with alignments length at *16 bp*. 109

4.6 Performance comparisons among different implementations at parallelization *Level 2* and *Level 3* of *ksw_ extend2* with alignments length at *16 bp*. 111

B.1 Parallelization Level 1 Test 1 138

B.2 Parallelization Level 1 Test 2 140

B.3 Parallelization Level 1 Test 3 142

B.4 Parallelization Level 2 and Level 3 Test 1 144

B.5 Parallelization Level 2 and Level 3 Test 2 146

B.6 Parallelization Level 2 and Level 3 Test 3 148

LIST OF FIGURES

2.1 Non-Search Version of the Dynamic Programming Algorithm with Edit Distance. 13

2.2 The Dynamic Programming Algorithm with Approximate String Matching . 14

2.3 A NFA for Approximate String Matching of The Pattern *corner* with Allowing Two Edit Operations 16

2.4 The Suffix Array and Compressed SA of *acaaccg*\$ 25

2.5 A Generic GPGPU Architecture 35

2.6 Memory Hierarchy in CUDA 36

3.1 A Simplified Version of big data *GPGPU distributed BWA-MEM* Framework. 45

3.2 Hot Spot Analysis for *BWA-MEM* at a Read length of 100 bp 58

3.3 Hot Spot Analysis for *BWA-MEM* at a Read length of 250 bp 59

3.4 The Calculation of Edit Distance in Smith-Waterman Algorithm 60

3.5 GPU Version of Smith-Waterman Algorithm Computation with Same Read and Reference Sizes 63

3.6 GPU Version of Smith-Waterman Algorithm Computation with Different Read and Reference Sizes 64

3.7 Three Level of Parallelization 65

3.8 *ksw_ extend2* Algorithm *H* Matrix 68

3.9 *ksw_ extend2* Algorithm *M* Matrix 68

3.10 *ksw_ extend2* Algorithm *E* Matrix 69

3.11 *ksw_ extend2* Algorithm *F* Matrix 69

3.12 *ksw_ extend2* Algorithm *S* Matrix 70

3.13 *ksw_ extend2*'s Cell Computation 73

3.14 Prunable Cells in *H* Matrix 73

3.15 Three Level of Parallelization 75

3.16 Blocks in *Level 1* Parallelization 76

3.17 GPU *Level 2* Parallelization 77

3.18 GPU *Level 3* Parallelization 77

4.1 CPU and GPGPU versions alignment performance comparisons with without the pruning mechanism with different similarities at parallelization *Level 1*. . 97

4.2 GPU and CPU versions performance comparisons with or without the pruning mechanism at parallelization *Level 1*. 100

4.3 GPU and CPU versions performance comparisons with or without the pruning mechanism at parallelization *Level 1*. 103

4.4 Performance comparisons among different implementations at parallelization *Level 2* and *Level 3* of *ksw_ extend2* with alignments length at *8 bp*. 108

4.5 Performance comparisons among different implementations at parallelization *Level 2* and *Level 3* of *ksw_ extend2* with alignments length at *16 bp*. 110

4.6 Performance comparisons among different implementations at parallelization *Level 2* and *Level 3* of *ksw_ extend2* with alignments length at *32 bp*. 112

List of Algorithms

1 Non-search Version of The Dynamic Programming Approach 12
2 The Dynamic Programming Algorithm with Approximate String Matching . 15
3 Smith-Waterman Algorithm 61
4 the Algorithm for Computing Substitution Matrix 62
5 Simplified Smith-Waterman Algorithm 63
6 Maximum Number of Cells Computed in Each Step 64
7 *ksw_ extend2* Matrices Initialization 71
8 *ksw_ extend2* Matrices value Computation 72
9 *ksw_ extend2*'s Pruning Mechanism Algorithm 74

1 INTRODUCTION

In the past, the most challenging task in genome study is to obtain sequences of the genome. The *Human Genome Project* [1] lasted 13 years with Sanger sequencing. Nowadays, by using the massively parallel sequencing, the *Next Generation Sequencing* (NGS) platforms, such as *Pyrosequencing* (Roche/454) [2], *Reversible Dye Terminator Sequencing* (Illumina) [3], and *Ion Semiconductor Sequencing* (Ion Torrent/Proton) [4] can sequence the entire human genome at a fraction of the cost and much shorter time. Therefore, the bottleneck is no longer to obtain but to analyze all the sequence data [5]. The demand of development in new sequence alignment algorithms has led to BWA, Bowtie, and Partek [6]. The sequence alignment algorithm is a pattern-matching algorithm with patterns and texts over *A, C, G, and T*. Given a string pattern p (read) with length m, the *sequence alignment algorithm* determines its location on the string text t (reference) with length n. As the size of the *reference genome* is enormous[1], the sequence alignment process is a difficult task.

We choose to speed up BWA as it is a popular software package for mapping lower-divergent sequence against a large reference genome in bioinformatics [8]. In addition, this tool is being used in Doctor Liang's lab[2]. As NGS develops, the newer *NGS platforms* are getting better at obtaining longer reads. Both tools in the *BWA* package follows the typical seed-and-extend paradigm, but with *BWA-ALN* tailored towards shorter reads and *BWA-MEM* tailored towards longer reads. As a typical alignment has both exact and approxi-

[1]For example, the human genome has approximately 3.2 billion base pairs [7].

[2]One of the author's supervisor.

mate matches, the combination of two methods improve both the speed and accuracy. In both tools, exact string matching algorithm finds the exact match location on the reference genome, and *approximate string matching algorithm* extends the exact matched area at a lower speed, but each with different algorithms.

In the past decade, many advances has been made towards sequence alignment algorithm, which makes hard to create a new one or improve existing one. As newer hardware and software frameworks come out every single day, we decide to speed-up BWA-MEM with new parallelization technique. The four main types of parallelizations in *sequence alignment tools* are *CPU multi-threaded* [8], [9], *GPU multi-threaded* [10]–[14], *CPU distributed* [15], [16], and *GPU distributed* [17]–[19]. As a popular sequence alignment tool, *BWA-ALN* already has multiple parallelized versions, such as BarraCUDA [10], pBWA [9], BigBWA [15], SparkBWA [16]. However, there are no open-source versions of GPU *BWA-MEM* available [8].

As existing *GPGPU distributed software* has been shown to outperform other versions, the *GPGPU distributed BWA-MEM* has a high chance to outperform other *BWA* parallelization versions. *GPGPU distributed version* is a combination of *GPU multi-threaded* and *CPU distributed version*. *GPU multi-threaded* and *CPU distributed BWA-ALN* have shown a good speedup compared to their original program [10]–[16]. *BWA-MEM* and *BWA-ALN* follow *seed-and-extend paradigm*, which indicates that they have a similar software structure and they may have a similar speedup. The *GPU multi-threaded version* is executed on each node in the distributed network, and the *master node* uses OpenMPI [9] or *MapReduce* [15], [16] to communicate with others to oversee the entire process. As the software has already gained speedup at the single node level, the speed would quickly add up by having multiple nodes.

OpenMPI and big data are the two most commonly used frameworks to help nodes to communicate with each other [9], [15], [16]. The *big data version* of *BWA-ALN* has shown a better feature over OpenMPI as it is version agnostic [9], [15], [16]. The same design can be made with *BWA-MEM*, which allows it to be *version agnostic*. As GPGPU parallelized version is the *GPGPU distributed* version's predecessor, the first step is to implement the GPGPU parallelized version.

Therefore, we propose *GPGPU distributed BWA-MEM* as our solution, and it has not yet being studied in any previous research. The *GPGPU distributed BWA-MEM*'s structure has two layers. Layer one executes GPGPU *BWA-MEM*, which contains an adapter and an unmodified GPGPU *BWA-MEM*, and is written in C/C++ language. Layer two is the core of the execution, and is responsible to get user input and then performs map and reduce. It will set up RDDs, map data, and then reduces output into a single file.

As *GPGPU distributed BWA-MEM* is build based on GPGPU *BWA-MEM*, the first step is to parallelize *BWA-MEM* with NVIDIA CUDA. The five steps of *BWA-MEM* are: 1) loading both index and read files content into the program, 2) finding super-maximal exact matches (SMEM), 3) chaining, and chain filtering, 4) seed extension, 5) generating the output file [20]. In the hot spot analysis, *ksw_ extend2* has increased from *28.9%* to *51.9%* of the total execution time when the sequence length increased from *100 bp* to *250 bp* [21]. Therefore, the first step is to parallelize *chain2aln*[3] and its subsequent functions. *ksw_ extend2*, as part of *chain2aln*, is based on Smith-Waterman algorithm, and performs single seed side extension. After successful parallelization of *ksw_ extend2*, tests have shown that *ksw_ extend2* GPGPU version is roughly *9x*[4] faster.

As the new stable release version of *Spark* recognizes GPGPU as resources, a *Spark* standalone version of *GPGPU distributed BWA-MEM* version can be build and run on *SharcNET*. the test was performed to ensure the input and the output of *GPGPU distributed BWA-MEM* with parallelization *Level 1* is the same as *BWA-MEM*.

[3]*ksw_ extend2* is step 4) seed extension in *BWA-MEM*.

[4]The test was performed on intel i7-6700k CPU, NVIDIA GeForce GTX 1080, 32GB RAM, and 1TB solid state hard drive desktop.

1.0.1 Major Contributions

This thesis covers background information on the DNA sequence alignment tools and works undertaken to speedup *BWA-MEM* with *GPGPU distributed* framework. Detailed information is provided regarding the significant difference between *BWA-MEM* and *BWA-ALN*, especially in terms of approximate string matching, *exact string matching* and the details for each stage of the *BWA-MEM*. An overview of different parallelization methods are provided, followed by the parallelization approaches attempted to improve BWA's speed. We will show why and how parallelizing the *BWA-MEM*'s *seed extension algorithm*, *ksw_ extend2*, could enhance *BWA-MEM* performance.

As the NGS platforms produces large amount of information, the work in biology has shifted from obtaining sequence data to analyzing them. From algorithm aspect of view, many advances have been made, and it makes hard to create or improve existing one. From hardware aspect of view, many new hardware and software come out every single day. Therefore, we propose *GPGPU distributed BWA-MEM* as our solution, and this idea has not mentioned in any paper before. We have successfully parallelized seed-extension part of *BWA-MEM* on the GPU side, where the *GPU with time-saving* version of *ksw_ extend2* is roughly *9x*[5] faster. *GPU with memory-saving* version of *ksw_ extend2* is roughly *3x*[6] faster. And we have successfully build parallelization *Level 1* of *BWA-MEM* into *Spark* framework in the standalone mode, which means we can use the *SharcNET* in the future.

[5]The test was performed on intel i7-6700k CPU, NVIDIA GeForce GTX 1080, 32GB RAM, and 1TB solid state hard drive desktop.

[6]The test was performed on intel i7-6700k CPU, NVIDIA GeForce GTX 1080, 32GB RAM, and 1TB solid state hard drive desktop.

2 BACKGROUND

This Chapter serves as a literature review, which includes background information on sequencing platforms, algorithms, parallelization techniques, and BWA packages in general. it will explain why BWA package is chosen, why *BWA-MEM* is better than BWA-ALN, why Big Data framework is better than OpenMPI, and why we didn't choose any other languages.

2.1 DNA Sequencing

DNA sequencing determines the *nucleic acid sequences* or the arrangement of four bases, *adenine, guanine, cytosine, and thymine* in a DNA molecule [22]. *Sanger sequencing* [23], also known as the first generation sequencing, is still the gold standard, and is considered extraordinarily accurate but slow and expensive [24]. In contrast to the *first generation sequencing platforms*, by using the massively parallel sequencing technique, *NGS platforms* can sequence the entire human genome in a fraction of cost and a short amount of time[6].

2.1.1 Sanger Sequencing

Sanger sequencing uses a *classical chain termination method* by *polymerase chain reaction*[1] Chain termination halts the reaction by adding the modified ddNTPs to the end of a growing nucleotide chain. Sanger sequencing determines the sequence based on the ending base

[1]Polymerase chain reaction, or PCR is a method to rapidly-produce millions to billions of copies from tiny sample of DNA [25].

via radioisotope labelling initially but now via fluorescent labelling, and it determines the sequence of nucleotide bases for DNA length less than *1,000 bp*.

2.1.1.1 Capillary Sanger Sequencing

In the new version of Sanger sequencing, DNA is combined in a tube with *DNA primer*, *DNA polymerase*, *normal deoxynucleotide-tri phosphates*, and *modified dideoxynucleotides (ddNTPs)* for termination [24]. ddNTPs[2] are *chain-elongating inhibitors* of *DNA polymerase*[3]. The process of *Sanger sequencing* has multiple repeated cycles. In each repeated cycle, the mixture is first heated to denature the template DNA [24]. The mixture is cooled done for the primer to be combined with *single-stranded template.* The temperature is raised again allowing DNA polymerase to synthesize, and it will not stop until a ddNTPs is added [24].

At the end of the cycle, the tube contains sequence fragments with different lengths. Each end of the fragments is labelled with dye to indicate the final nucleotide. After the reaction is done, *capillary gel electrophoresis* is used to determine the DNA sequence. In *capillary gel electrophoresis*, the fragments run through a long, thin tube containing a gel matrix. The shorter fragments move quickly through the gel, and the longer fragments move slower. A laser illuminates the fragments as they reach the end of the tube, which allows the attached dye to be detected [24]. The signal generated by the detector is presented as a peak on the graph. *Capillary Sanger sequencing* supports up to *396* samples per run[4].

2.1.1.2 Microfluidic Sanger Sequencing

Microfluidic Sanger sequencing is a *wafer-scale chip* that integrated all sanger sequencing steps with *nanoliter-scale sample volumes* [26]. This technology keeps classical Sanger sequencing's benefits with increased capacity[5].

[2]ddATP, ddCTP, ddGTP, and ddTTP.

[3]As *nucleotides lacking a 3'-hydroxyl (-oh)* group.

[4]Provided the number to show the throughput/run for comparison.

[5]Provided the number to show the throughput/run for comparison.

2.1.2 *NGS platforms*

NGS, also known as massively parallel sequencing, can sequence *billions of DNA base-pair*s at a fraction of the cost and time[6]. At the cost of lower accuracy and shorter sequence length in most cases, NGS platforms have a good speedup. Popular NGS includes: *pyrosequencing*[7] [2], *reversible dye terminator sequencing*[8] [3] and *ion semiconductor sequencing*[9] [4]. *FASTQ format* [27] is the *de facto standard* for storing the output of *high-throughput sequencing instruments*, and is a *text-based format* for storing both raw sequence and corresponding base call quality scores.

2.1.2.1 Pyrosequencing (Roche/454)

Pyrosequencing relies on *sequencing by synthesis*, where the complementary strand of a single-stranded DNA is enzymatically synthesized [2]. A parallelized version of the *pyrosequencing* method is developed with emulsion PCR for *DNA amplification.* During the process, DNA strands are broken up into fragments of *400 bp.* These fragments are split across wells, where each well only contains one type of DNA fragment by sequence. The four types of dNTPs are added to the wells one by one for *polymerization*, releasing *pyrophosphate*, which resulted in light emission with *ATP Sulfurylase.* The light emitted is picked up by a detector, where the intensity of the light infers the number and type of *dNTP.*

2.1.2.2 Reversible Dye Terminator Sequencing (Illumina)

Illumina sequencing[10] uses a *fluorescent labelling method* with the *clonal amplification* of DNA on a surface, which is based on *DNA clusters* or *DNA colonies* [3]. The *polymerases* used in the process is specially engineered with the addition of reversible terminate bases.

[6]In comparison to Sanger Sequencing.

[7]Roche/454.

[8]Illumina

[9]Ion Torrent/Proton

[10]Reversible dye-terminators sequencing technology.

The sequencing is done in cycles by adding four types of *fluorescently* labelled dNTPs[11]. A laser camera captures the fluorescent colour to identify the newly added nucleotide.

2.1.2.3 Ion Semiconductor Sequencing (Ion Torrent/Proton)

Semiconductor sequencing, also known as *Ion Torrent Semiconductor sequencing*, sequences DNA strands by detecting the hydrogen ions released through the *DNA polymerization process* [4]. *Ion Torrent sequencing* is developed based on standard sequencing chemistry with an *ion-sensitive field-effect transistor (ISFET)*, and it does not require chemically modified nucleotides, optical devices, or special enzymes.

As ISFET can not tell the difference between nucleotides, the microwell containing a template DNA strand flooded with a single type of nucleotide. The detection of ions from *microwell* indicate newly added ddNTP, where a higher electronic signal indicates a higher number of hydrogens. Ion Torrent sequencing is considered inexpensive, but with the limitation of *much lower throughput* and *lack of pair reads*[12].

2.1.2.4 PacBio Sequencing

PacBio[13] is powered by single Molecule, real-time sequencing technology. First, for sample type, ranging from viruses to vertebrates, their DNA or RNA is isolated. Next, a SMRTbell library is created by ligating the adapters to double-stranded DNA, creating a circular template. The smart sequencing core is the smart cell, which contains millions of tiny wells called zero-mode waveguides (ZMWs) [28]. As single-molecule DNA is immobilized in the *ZMWs*, the polymerization incorporates labelled nucleotide, which emits light at each nucleotide [28]. With this approach, nucleotide incorporation is measured in real-time. PacBio can optimize

[11]dNTPs do not allow further extension of DNA synthesis before chemically removing the blocker.

[12]Single read length is longer than Illumina.

[13]also referred to as smart sequencing technology.

the result with two sequencing modes: circular consensus sequencing[14] and continuous long read sequencing[16].

2.1.2.5 Nanopore Sequencing

Nanopore sequencing uses nanopore-based DNA and RNA sequencing technology with the advantage of being *portable* and *producing longer reads.* Protein nanopores are tiny holes crossing membranes, which are embedded into a synthetic membrane [30]. The *synthetic membrane* is bathed in an electrophysiological solution, and the ionic current is passed through the nanopores. DNA and RNA molecules disrupt the current as they are passing through the *nanopores*, where the signal generated is analyzed in real-time [30]. *Nanopore sequencing* sequences from tens to hundreds of kilobases with the extreme long read length being its other major strength over other *NGS platforms* in addition to its high portability. Nanopore technology can sequence DNA and RNA directly without PCR, which removes the bias of PCR. Together with PacBio, it is called *the third generation sequence technologies* for *real-time single molecular sequencing without PCR amplification in long read length.*

2.1.3 De-novo Sequencing Versus Re-sequencing

Whole-genome sequencing aims to sequence all the DNA in an organism's genome, either with *de-novo sequencing* or *re-sequencing. De-novo sequencing* refers to sequencing a novel genome, for which being sequenced for the first time [31]. Re-sequencing is commonly used for sequencing individuals' genome that has being sequenced before in species, which is used to identify genomic variations of a test genome sample **re-sequencing**. Sequence alignment is one of the earliest step in analyzing re-sequencing data and is directly related to the

[14]circular consensus sequencing, or *CCS* is used to produce highly accurate long reads[15], which has an accuracy of *99%* [29].

[16]continuous long read sequencing, or *CLR* is used to generate the longest possible read, where half of the reads are longer than *50 kb* [29].

research involved in this thesis.

2.2 String Matching

Sequence alignment is a way to arrange the sequences to identify the similarity regions so that the functional, structural, and evolutionary relationships between two sequences can be inferred. In the context of sequence alignment, the DNA sequence can be regard as long texts over the alphabet *A, T, C, G. "Errors"* in DNA sequences are caused by insertion, where the sequence of one or more nucleotides are added between two adjacent nucleotides, or deletion, where the point at which one or more contiguous nucleotides exists, or substitution, which is a substitution of a single nucleotide at a specific position. The substitution of a single nucleotide is commonly referred to as *Single Nucleotide Polymorphism*[17] (SNP). Sequence alignment algorithm can be simplified as an algorithm that solves *pattern matching problem*, which takes a string pattern p (read) and a string text t (reference) with lengths of m and n as inputs, and returns all the positions in the text where the pattern appears.

Approximate string matching solves retrieval problems such as *sufficiently like* or *most like*, where the *non-exact string* comes from *error correction* or *information retrieval* or *file corruption.* During an approximate string match, the allowance for a maximum specified number of errors in each match is specified.

2.2.1 Brute Force Approach

The *brute force algorithm* checks all positions of the text between 0 and $n - m$ if the pattern occurs. After each *attempt*, in the repeating *attempt*, the *window* is shifted by exactly one position to the right. There is no pre-processing phase in the brute force algorithm[18], and the time complexity of the searching phase is $O(mn)$.

[17]Human genome has roughly 4 to 5 million *SNPs* of out the *3 billion nucleotides* [32].

[18]Algorithms that do not pre-process the text or pattern is referred to as online algorithms.

2.2.2 Dynamic Programming Algorithms

The first algorithm in dynamic programming approach has been rediscovered many times in the past [33]–[41]. However, this algorithm was first designed to compute the edit distance, and it was not being converted to a string matching algorithm until 1980 by Sellers [42].[19]

2.2.2.1 Edit Distance Computation

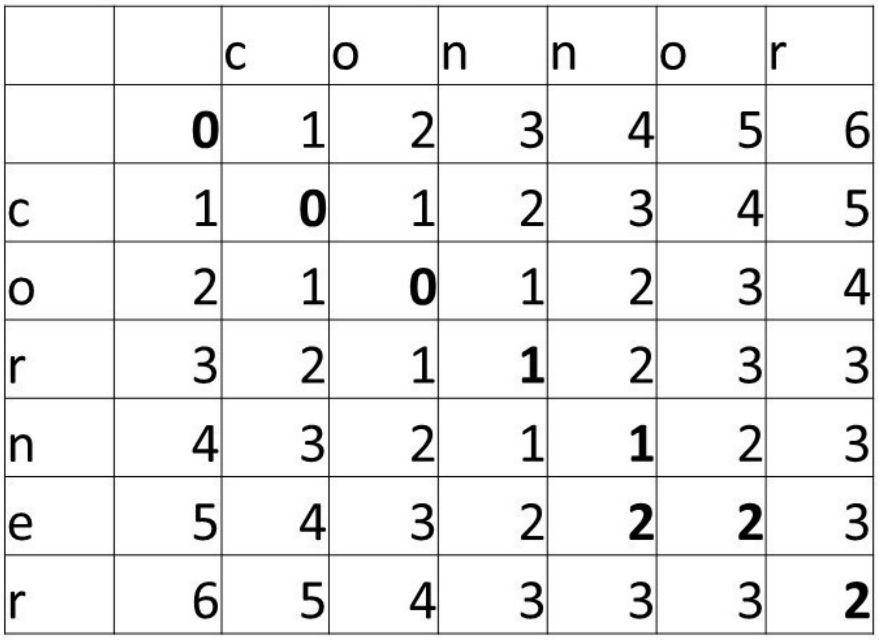
The early *edit distance function* was not used in string searching, we will refer them as non-search version to differentiate from Seller's algorithm. *Edit distance* describes how dissimilar of two strings by counting how many operations are needed to transform one to another. *Edit distance* (denoted as k) is computed by filling matrix C of size $n \cdot m$, where $C_{i,j}$ represents the edit distance between first i characters of p to the first j characters of t. The well-known algorithm is given below:

[19]Although the algorithm is not very efficient, it is still widely used for being easy to map the solution to different distance functions [43].

Algorithm 1 Non-search version of the dynamic programming approach. the first two lines initialize the first row and first column's value to the corresponding word's length, which represents the *edit distance* between p or t with an empty string. The third line calculates all the edit distance for the shorter substrings. If the i^{th} character of p and the j^{th} character of t are not equal, the *edit distance* is the minimum value from $C_{i-1,j}, C_{i,j-1}C_{i-1,j-1}$ plus one. On the other hand, if they are equal, the current edit distance is assigned at this location. An example is given in Figure 2.1 using Algorithm 1 to compute the edit distance between *connor* and *corner*.

```
$C_{i,0} = i$
$C_{0,j} = j$
$C_{i,j} = if\,(x_i = y_j)\,then\,C_{i-1,j-1}$
        $else\,1 + min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1})$
```

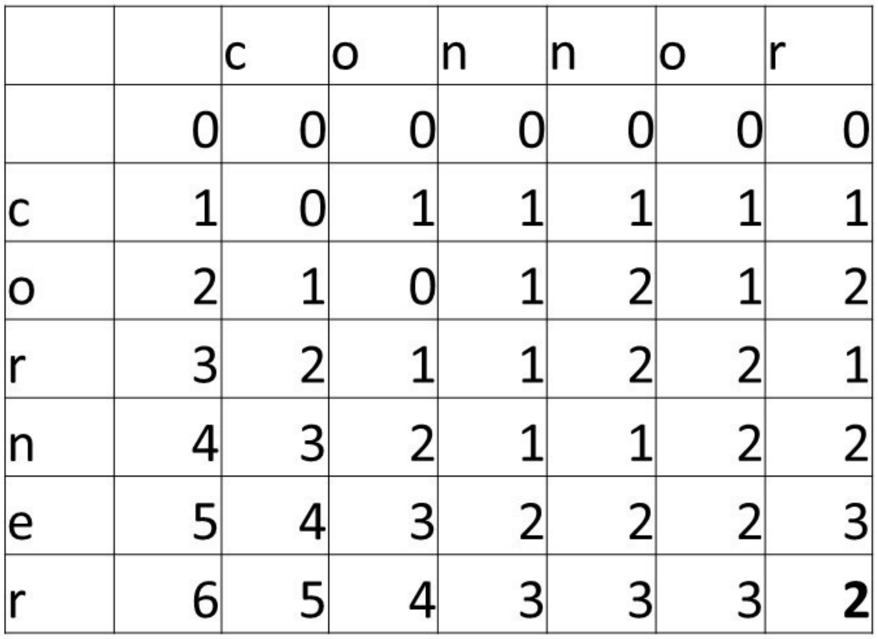
In Algorithm 1, the first two lines initialize the first row and first column's value to the corresponding word's length, which represents the *edit distance* between p or t with an empty string. The third line calculates all the edit distance for the shorter substrings. If the i^{th} character of p and the j^{th} character of t are not equal, the *edit distance* is the minimum value from $C_{i-1,j}, C_{i,j-1}C_{i-1,j-1}$ plus one. On the other hand, if they are equal, the current edit distance is assigned at this location. An example is given in Figure 2.1 using Algorithm 1 to compute the edit distance between *connor* and *corner*.



		c	o	n	n	o	r
	0	1	2	3	4	5	6
c	1	**0**	1	2	3	4	5
o	2	1	**0**	1	2	3	4
r	3	2	1	**1**	2	3	3
n	4	3	2	1	**1**	2	3
e	5	4	3	2	**2**	**2**	3
r	6	5	4	3	3	3	**2**

Figure 2.1 Non-Search version of the dynamic programming algorithm with edit distance. The dynamic programming algorithm to compute the edit distance between *connor* and *corner*. The bold entries show the path to the final result.

2.2.2.2 Text Searching with Edit Distance



		c	o	n	n	o	r
	0	0	0	0	0	0	0
c	1	0	1	1	1	1	1
o	2	1	0	1	2	1	2
r	3	2	1	1	2	2	1
n	4	3	2	1	1	2	2
e	5	4	3	2	2	2	3
r	6	5	4	3	3	3	**2**

Figure 2.2 The dynamic programming algorithm with approximate string matching.shows the search version of the *dynamic programming approach* using Algorithm 2. In comparison to non-search version, the search version must allow any character to be the potential starting point. To change non-search version to search version, $C_{i,0} = i$ is modified to $C_{i,0} = 0$. The algorithm worst case time complexity and space complexity are $O(mn)$ and $O(m)$.

Algorithm 2 The dynamic programming algorithm with approximate string matching. shows the search version of the *dynamic programming approach* using Algorithm 2. In comparison to non-search version, the search version must allow any character to be the potential starting point. To change non-search version to search version, $C_{i,0} = i$ is modified to $C_{i,0} = 0$. The algorithm worst case time complexity and space complexity are $O(mn)$ and $O(m)$.

```
$For\, all\, i \in \{0, 1, 2, ..., m\}:$
$C_i' = if(P_i = T_j)\, then\, C_{i-1}$
            $else\, 1 + min(C_{i-1}', C_i, C_{i-1})$
```

Figure 2.2 shows the search version of the *dynamic programming approach* with Algorithm 2. In comparison to non-search version, the search version must allow any character to be the potential starting point. To modify the non-search version into search version, $C_{i,0} = i$ is changed to $C_{i,0} = 0$. The algorithm worst case time complexity and space complexity are $O(mn)$ and $O(m)$[20].

2.2.3 Finite Automata Approach

Finite automata is a fairly old approach to *approximate string matching.* Finite automata constructs the *finite automation* out of the pattern and feed the text one character at a time through automation. The approximate match is determined with the final state of the *Non-deterministic Finite Automaton* (NFA).

[20]This is because only the previous columns are stored.

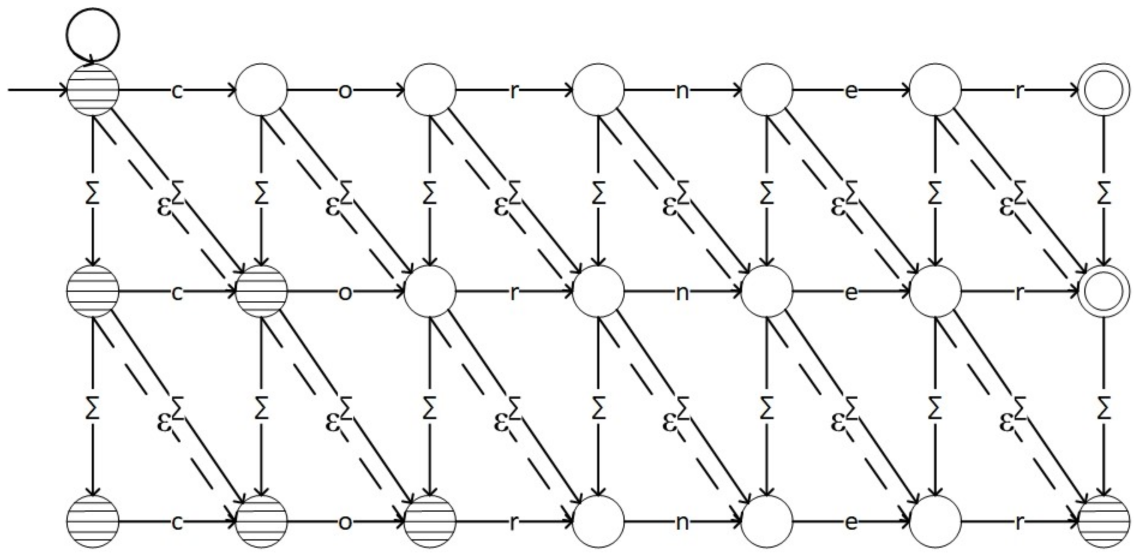


Figure 2.3 A NFA for approximate string matching of the pattern *corner* with allowing two edit operations. The shaded states are those active after reading the text *connor*.

Figure 2.3 is an NFA for approximate string matching of the pattern *corner* with allowing two edit operations. Each row represents the number of errors encountered, and every column represents the matching of a prefix of the pattern. An active state in column m and row i means the pattern has been approximately matched to the text with i errors. The automaton transitions describe different actions, where a horizontal arrow represents a character match, and a vertical arrow represents an insertion in the pattern. In Figure 2.3, a solid diagonal arrow represents a substitution in the pattern, and a dashed diagonal arrow represents a deletion in the pattern. *Finite automata* is easy to visualize in concept, but it is impractical. It has three different ways when moving from one cell to another in the *dynamic programming matrix*, which means an allowance for 3^m different states is made to ensure every combination of transitions are available. This approach is considered unfeasible as 3^m quickly explodes.

2.2.4 Hashing Based Approach

Instead of comparing each position of the text if the pattern occurs, the hashing based approach avoids a quadratic number of character comparisons by treading pattern or contents of the window as a single integer. *Karp-Rabin algorithm* is one of the hashing based approach algorithms [44], and it uses the hashing value[21] from left to right. In the pre-processing phase, the pattern p is divided by a pre-defined prime number q in constant space and $O(m)$ time. During the search phase, the remainders of pattern and text are compared for matching for each shift ranges from shift $s = 0$ to $n - m$. Once the match is found, it is still necessary to check each character to ensure a true match for the searching phase.

2.2.5 Bit Parallel Approach

bit-parallel approach is based on parallelizing another algorithm using bits, and its results, especially when deal with short patterns in text retrieval, have shown significant improvement. In computing, the *computer words' length* is an essential characteristic of the processor, determined by the processor's design. By combining multiple entries into a single word, the number of operations is reduced. The two branches of bit parallel approach are automation parallelization and matrix parallelization [43].

The first *bit-parallel algorithm*, Shift-Or, was introduced by Baeza-Yates, which takes advantage of the bit operations inside a computer word [45]–[47]. If computer word length is w, the number of operations can be reduced by a factor of at most w. In the algorithm, an NFA is parallelized to search a p[22] in t.

Wu and Manber [48] extended the *Shift-Or algorithm* based on simulating NFA for regular expressions with the wild cards, where each row i of the NFA fits in a computer word R_i (*row-wise bit-parallel algorithm*). Later on, Baeza-Yates [49] presented a *column-wise bit-parallel*

[21]Computed using *Horner's rules.*

[22]without error.

algorithm, but neither Wu and Manber's nor Baeza-Yates's can increase the parallelization level. Baeza-Yates and Navarro [50] proposed a *diagonal-wise bit-parallel algorithm*, where the states are calculated by diagonals instead of rows [48] or columns [49].

The first *bit-parallel approach* on the *dynamic programming matrix* is proposed in [51], where the secondary diagonal is computed using the two previous diagonals. The algorithm packs many patterns and text characters in a single computer word, where the results of the comparisons can update many cells of the diagonal simultaneously. Myers [52] proposed a new way of bit-parallel approach, where the computer words no longer represent the columns themselves, but the differences along with columns, which increased the number of cells in a single computer word (two bits per cell).

2.2.6 String Indexing

String indexing method, a relatively new approach, is handy when deal with frequent searches on a massive text. Indexing methods have been developed for extract string matching, but a recent development has modified it to accommodate *approximate string matching.* The string indexing approach is beneficial when deal with large patterns and long text in the sequence alignment. In most indexing methods, a traditional algorithm is needed to verify the matches once a set of candidate matches has been found.

The string indexing approach pre-scans both text or pattern to archive all occurrences of each sub-string with a specific length (also referred to as query size). These pattern occurrences are stored in a number format and sorted in descending order. For example, the text $AAAACCGAAAAG$ with a query size of four, the first pattern $AAAA$ will be converted to the number format of 1111 at position one. The next pattern $AAAC$ will be converted to the number format of 1112 at position two. After all sub-strings have been converted and sorted with numbers, their pattern, start index, and end index will be stored.

With the help of the pre-scanned archive, within the text, a specific pattern's occurrence locations can be determined very fast, and the advantages are obvious, especially dealing with larger text. However, the concern is the query and alphabet size. For example, in biology, the size of the alphabet is four, with a pattern size of *30*, there will be 4^{30} possible entries in the index[23] As the index of this size is not practical, a smaller query size is necessary. In addition, as approximate string matching is more common in biology, only searching for an exact match on the index is not practical. Therefore, many tools combine both exact and inexact matches at the same time to increase performance.

2.2.6.1 The Word Neighbourhood

The *word neighbourhood* is an approximate string matching algorithm with an index [53]. The *word neighbourhood for a pattern p and an edit distance k* will contain all words within k edit distance operations of p. Once the *word neighbourhood of k* for p is generated, every word in the neighbourhood is searched, and each hit is recorded for an approximate match within the location. Even though it sounds like a simple solution, the size of the neighbourhood can quickly explode. The size of *word neighbourhood* has been bounded at $O(m^k\alpha^k)$ [54]. Since the word neighbourhood's size increases rapidly, it is only practical to have an extremely small m and k.

2.2.6.2 Exact Partitioning

Exact partitioning is another method for approximate string matching over an index [54]. In biology, every approximate match of a pattern p, there are sections of p that match the text T exactly. In an approximate match, if k errors are allowed, and the pattern is split into $k + 1$ sections, and then one of the sections is guaranteed to match exactly by the pigeonhole principle. We assume that a query size of $\lceil \frac{m}{k+1} \rceil$, then each pattern is split into $k+1$ sections with length of $\lceil \frac{m}{k+1} \rceil$. Each of the sections is searched over the index. For each hit, the *surrounding text* is verified with an *inline approximate string matching algorithm* for

[23]If we want the query size equal to the pattern size.

a potential match. Exact partitioning is useful when the value of k is moderate. A smaller k value means a large query size, and a large k size means the *query size* will be so short and resulting in many *false-positive hits*. Other issue is when $k + 1$ does not divide evenly into m, which means overlap in some sections.

2.2.6.3 Intermediate Partitioning

The intermediate partitioning is the most recent string indexing approach [55], and is considered the combination of the two previous approaches and produces better results. Myers showed that the optimal query size for an index is equal to $\log_\alpha n$ in 1994. The intermediate partitioned approach by Navarro uses this optimal query size to build an index. Then the pattern is split into $j = \left\lceil \frac{m}{\log_\alpha n} \right\rceil$ sections of length $\left\lceil \frac{m}{\log_\alpha n} \right\rceil$. Just like the exact partitioned approach, one of the sections is guaranteed to have at most $\left\lfloor \frac{k}{j} \right\rfloor$ errors. Then, similar to the word neighbourhood approach, a $\left\lfloor \frac{k}{j} \right\rfloor$ neighbourhood is generated for each section. For each of the word in the neighbourhood, it returns a hit on the index, and the surrounding text is checked for an *approximate match* with a *traditional approximate string matching algorithm.*

2.3 BWA Package

BWA is one of the most famous sequence alignment packages for *mapping low-divergent sequence* against an extensive reference. The two similar tools within the package, *BWA-MEM* and *BWA-ALN*, both follow the typical seed-and-extend paradigm but use different algorithms for exact and approximate matching. As a popular tool, *BWA-ALN* already has multiple parallelized versions, such as BarraCUDA [10], pBWA [9], BigBWA [15], SparkBWA [16]. However, there are no parallelized open-source versions of GPGPU *BWA-MEM.* In *BWA-ALN*, backward search and bounded traversal/backtracking facilitate both exact and approximate matching[24]. In comparison, *BWA-MEM* seeds extension with SMEM

[24]Seed extension.

based on FMD-INDEX, and extend seeds with the Smith-Waterman algorithm.

2.3.1 Seed-and-extend Strategy

The idea of the seed-and-extend strategy is based on the observation that a good sequence alignment should contain both exact and inexact matches [56]. The seed-and-extend strategy's process contains four stages: seed generation, seed mapping, seed extension, and read alignment. Seeds are the shorter sequences extracted from reads in the seed generation stage. During the mapping stage, exact-matched seeds are identified. After successfully pinpointing each read's location with exact-matched seeds, in the seed extension stage, a standard dynamic program such as the Needleman-Wunsch algorithm [34] or Smith-Waterman algorithm [57] is used to extend each exact-matched seed on both ends. As seed extension is considered more time consuming than both seed generation and seed mapping, to reduce time consumption, seed filtration strategies are commonly used before actual seed extension. Also, the alignment tool's performance is affected by the seed's length, where shorter seeds increase the sensitivity, and longer seeds increase the speed.

2.3.1.1 Type of Seeds

A seed is a sub-string extracted from the read sequence that exactly matched a sub-string of the reference sequence. The two types of seeds, the *fixed-length exact matches or seeds* strategy is used in Novoalign and Bowtie2, and *maximal exact matches* (MEM) strategy is used in *BWA-MEM* and Cushaw2 [56]. The fixed-length seeds are substrings with the same length generated from the read, where the MEM is the longest exact matches that cannot be further extended [20]. Super-maximal exact match (SMEM) is a MEM that is not contained in any other MEMs on the query coordinate [32].

2.3.2 BWA-ALN

A k-mer inexactly match seed from a read is generated with Pigeonhole principle, which supports mismatches and indels in mapping [8]. The pigeonhole principle states that at least one container contains more than one item for putting j items into k containers, where $j > k$. Therefore, if the length of the read is n, the number of allowed mismatched bases between a read and a reference is m bp, at least one exact match k-mer exists[25].

By default, for *BWA-ALN*, in each seed, the number of allowed mismatches is *2*. During the mapping stage, the process is facilitated by a prefix directed acyclic word graph[26] [58]. To reduce the unnecessary seed extension for highly repetitive sequences to improve the performance, a seed filtration strategy is used. Not all the exact match locations are provided. Instead, *BWA-ALN* only gives the largely non-overlapped exact match locations. The newly scanned seed extensions are discarded if the overlapped region's length is shorter than the successfully aligned regions.

2.3.2.1 Burrows-Wheeler Transform

The *Burrows-Wheeler transform* (BWT) rearranges a character string into runs of similar characters, which is initially intended for data compression [59]. In *BWA-ALN*, BWT is used as it can approximately match DNA reads efficiently, which has a quadratic ($O(n^2)$) time and space.

[25]As the read can be separated into non-overlapping k-mer with the length of $n/(m+1)$.

[26]Prefix directed acyclic word graph, or*DAWG* is a special index structure that represents all the substrings extracted from a string.

Table 2.1 All Rotations of $T = \$banada$. Given an input string $T = \$banada$, rotate N times, where $N = 8$ is the length of the T string.

F						L
$	b	a	n	a	n	a
a	$	b	a	n	a	n
n	a	$	b	a	n	a
a	n	a	$	b	a	n
n	a	n	a	$	b	a
a	n	a	n	a	$	b
b	a	n	a	n	a	$

Table 2.2 A lexicographically sorted BWT matrix given an input string $T = \$banada$. The previous produced rotations are sorted lexicographically.

Row	Occurrence	F						L	Occurrence
3	3	a	n	a	n	a	$	b	1
2	2	a	n	a	$	b	a	n	1
1	1	a	$	b	a	n	a	n	2
6	1	b	a	n	a	n	a	$	1
5	2	n	a	n	a	$	b	a	1
4	1	n	a	$	b	a	n	a	2
0	1	$	b	a	n	a	n	a	3

A BWT of an input string T is denoted as $BWT(T)$, where T's characters are from alphabet $\sum$. In the first step, the algorithm forms all rotations with the input text T, where the character *$* is appended to the end of the text (Table 2.1). Table 2.1 describes an unsorted BWT matrix of all possible rotations T as *banana*, where $ represents the end of the string[27] Table 2.2 describes a sorted BWT Matrix of T as *banana*, where each row has been sorted alphabetically. After sorting, the first and last columns are kept, where the last column is considered the product of BWT. For example, if T is *banana*, $BWT(banana)$ is *bnn$aaa* (red column of Table 2.2).

If the result of BWT is known, to get the original string T back, an algorithm is used to reverse the output of BWT. F and L represent the first and the last columns in the BWT matrix. The last-to-first index mapping can be denoted as $LF(i)$. $LF(i)$ is obtained with

[27] *$* is not present in the alphabet $\sum$, and is lexicographically smaller than all symbols in the alphabet (Table 2.2).

the help of 1D array $C[c]$, which contains the number of lexicographically sorted characters' occurrence, and 2D array $Occ[c, k]$, which contains occurrences of character c in the $L[1..k]$. Since the first column is lexicographically sorted, the $LF(i)$ is computed as $LF(i) = C[L[i]] + Occ(L[i], i)$. If $LF(i)$ is known, where $L[i] = T[k]$, we would have $L[LF(i)] = T[k-1]$. It means that in each row of the BWT Matrix, the first character is followed by the last character in the original input string. Therefore, the sub-string of the input string T's position can be easily determined. For example, in Table 2.2, the first step is to locate the position of *\$* sign in the column F at row 0. The last character of T is a. The character a with the same occurrence is in row 1, which is followed by n (the second last character in T). It is followed by the same n in row 4, which defines the third character as a. The algorithm repeats these procedures until the whole string *banana* is recovered.

2.3.2.2 Suffix Arrays

BWT can be used to approximate matches DNA reads onto a reference sequence (real data) very efficiently. However, the generation of the *BWA* requires the use of a matrix, which is $O(n^2)$ in time and space. It is not feasible to store a matrix when deal with large values of n, such as with the human genome. A combination of compression and indexing is introduced to reduce both time and space complexity [60]. BWT can be generated from a compressed suffix array in $O(n)$ time and $O(n \log |\sum|)$ space[28]. However, it should be noted that the time complexity for generating a compressed suffix array is $O(n \log n)$.

In suffix arrays, a new simple symbol *\$* is placed at the end of the text t, which does not exist in any alphabet and is lexicographically smaller than all other characters in the alphabet. t is stored in an array as $T[0, 1, ..., n-1]$, where $T[n-1] = \$$. Let us assume that t is stored in an array as $T[0..n-1]$, where $T[n-1] = \$$. The suffix of T is defined as T_i as $T[i..n-1]$, which represents all of the characters starting from $T[i]$ until the end of the text. A suffix array of T is defined as $SA[0, 1, ..., n-1]$, which contains a sorted sequence of

[28] $\sum$ represents the alphabet size.

all the suffixes of T, where $SA[i]$ is the lexicographically-smallest suffix of T starting from $i+1)^{th}$. As an example, $SA[0] = (n-1)$, and $T[SA[0]] = \$$ for all texts, if j is the value of $SA^{-1}[i]$, where $SA[j] = i$, $SA^{-1}[i]$ is how many suffixes are lexicographically smaller than T_i. As any pattern matches the text at any point within at least one suffix of the suffix array, a prefix will occur, and prefixes are grouped contiguously within the lexicographically sorted suffixes. As one of the matching prefixes is found, other matching prefixes can be accessed in constant time, which makes searching very efficient.

The compressible suffix array of T can be simplified as $\psi[0..n-1]$, where $\psi[0]$ is equal to $SA^{-1}[0]$. For all other $i = 1, 2, ..., n-1$, $\psi[i] = SA^{-1}[SA[i] + 1]$. To store this array, the naive approach would take $O(n \log n)$ space. From previous example, we can see that if $i < j$ and $T[SA[i]] = T[SA[j]]$, then $\psi[i] < \psi[j]$. In other words, if two suffixes i and j have the same first character, and i is lexicographically smaller than j ($\psi[i] < \psi[j]$), the compressible suffix array would consists of a sequence of increasing numbers [60].

i	$T[i]$	T_i
0	a	acaaccg$
1	c	caaccg$
2	a	aaccg$
3	a	accg$
4	c	ccg$
5	c	cg$
6	g	g$
7	$	$

i	SA[i]	$T_{SA[i]}$
0	7	$
1	2	aaccg$
2	0	acaaccg$
3	3	accg$
4	1	caaccg$
5	4	ccg$
6	5	cg$
7	6	g$

i	$\Psi[i]$	$T[SA[i]]$
0	2	$
1	3	a
2	4	a
3	5	a
4	1	c
5	6	c
6	7	c
7	0	g

Figure 2.4 The suffix array and compressed SA of *acaaccg*$.

In Figure 2.4, letters $T[SA[i]$ are grouped together with lexicographically increasing order. Also, within each group, the $\psi[i]$ is increasing. The compressible suffix array use this fact to store with $O(n(H_0 + 1))$ bits in $O(n \log n)$ time, where H_0 denotes the entropy of the text with at most $\log|\sum|$ [60]. ψ generate the Burrows-Wheeler transform, W, using formula $W[\psi_k[P]] = T[k-1]$, where $p = \psi[0]$. The ψ array is generated in $O(n \log n)$ time and $O(n(H_0 + 1))$ space, Burrows-Wheeler transform can be generated from ψ array for t in $O(n)$ time and space.

2.3.2.3 Indexing

First step in *BWA-ALN* is to index the reference sequence (real data) by performing *BWT*. As the index is saved as a file, the reference needs to be indexed only once. In order to search over both of the strands, a *BWT* is built on the reverse reference. In order to build the *BWT index*, *FASTA* formatted reference sequence (real data) file is compressed. As the *DNA alphabet* is only of size four over *A, T, G, and C*, the sequence is translated into a two-bit sequence. If symbol N (non-determined nucleotide by *NGS platform*) is encountered during indexing, a random base is chosen and assigned to the position.

2.3.2.4 Exact and Approximate Matching

The suffix array S for a string t is a permutation of the integers 0 through $n-1$, such that $S[i]$ is the start position of the i^{th} lexicographically smallest suffix in t. Suffix array interval is defined as a pair $(R_s(W), R_e(W))$, where $R_s(W)$ is the minimum index (W is a prefix of $t_{S[s]}$), and $R_e(W)$ is the maximum index (W is a prefix of $t_S[e]$). In the suffix array, each exact occurrence of p in t is determined by retrieving the value given by $S[k]$, where k represents each value in the suffix array interval. There would be one suffix array interval per pattern at most for exact string matching. However, approximate string matching may have many intervals per pattern.

For exact matching, the suffix array intervals for an exact match is determined as follows. $C[c]$ is the number of symbols in t that are lexicographically smaller than c, and $Occ(c, i)$ is the number of symbols in the first i characters of the BWT compressed version of t that are lexicographically smaller than c. c is set equal to $p[m-1]$, where the beginning suffix array interval would be $(C[c], C[c+1]-1)$. Backward search starts searching at the end of it, $R_s(aW) = C(a) + Occ(a, R_s(W)-1) + 1$, and $R_e(aW) = C(a) + Occ(a, R_e(W))$. Once hits the beginning of the pattern, under the case of $R_s(P) < R_e(P)$, p exists in t in all locations referenced by the suffix array interval.

An extension of a backward search is used to facilitate approximate matching in *BWA*. *BWA-ALN* first generates the word-neighbourhood for each pattern, and the exact match algorithm is performed on each word in the word-neighbourhood. The efficiency is improved by estimating the lower-bound of the mismatches with the reverse sequence. It makes *BWA-ALN* very efficient as the resultant array crops out large amounts of the word-neighbourhood.

The algorithm for determining the *suffix array* intervals of a pattern against a *BWT-compressed* sequence is used for approximate matching. It is a repeated process for the reverse reference on a complement sequence. As it is a recursive algorithm, the exit case is being checked first, and the algorithm exits based on two conditions. Firstly, if the estimated lower bound for the mismatches in p is higher than a limit, k, the algorithm exits with *null* as a result. Secondly, the program would exit when the end of the backward search has reached, where the suffix array interval is returned as a result.

In the second step, a recursive call to the current function is made, which lowers the *backward search* character position without searching and decreasing the number of errors allowed. This function call covers an insertion in the pattern as a character is skipped. In the third step, each possible character b is cycled through at the current index, and the suffix array interval is calculated when b is added to the front of the currently processed suffix. The program validates the *suffix array interval.* If it is not valid, the algorithm continues to the next iteration of the loop. Otherwise, the algorithm moves on.

In the fourth step, as the current algorithm is not moving backward from the current character, the algorithm makes the recursive call with the assumption that deletion is in the pattern but also decreases the number of errors encountered. In the last step, the algorithm handles the substitution errors. Suppose b matches the character in p at the current position. In that case, there is no substitution error, so the algorithm calls recursively with i[29] decreased by one[30] and keeps the k as same. Under the case of a mismatch, the

[29]Position index.

[30]moved backward

algorithm is called with both parameters i, and k decreased by one. The algorithm would leave with a set containing all the suffix array intervals as a result.

2.3.2.5 Alignment Determination

BWA-ALN calculates the quality score of all possible matches based on criteria of the number of *gap-opens*, *gap-extensions*, and *mismatches*. For paired-end reading, each sequence is first aligned in the same way for single-end read alignment. Statistical methods estimate the maximum, average, and minimum insert sizes for the entire group of sequences. The single-end alignments and the insert size estimates are used to map one read to its other pair. The longer the reads are, the faster the alignment phase would be, as the smaller the chance that multiple *good* reads would be produced. On the other hand, the shorter the reads, the slower the alignment phase, as there will be more suffix array intervals to look up. The alignment is finished when all of the sequences have had their alignment determined.

2.3.3 BWA-MEM

In the early days, most mappers are developed for reads of *36 bp* in length, which is reasonable to require *end-to-end alignment*, and they only report hits within a certain edit distance. However, with emerging technologies and improved technology, *NGS* reads are not short anymore. *100 bp* or longer reads need to allow longer gaps under the *affine-gap penalty* and report multiple *non-overlapping local hits* in the reference genome. *BWA-MEM* is the latest development of the *BWA-MEM* software package for *100 bp* or longer reads, which utilizes *FMD-index* and *SMEM* for faster alignment. *BWA-MEM* process has five major stages: file loading and indexing, seeding and re-seeding, chaining and chain filtering, seed extension, and output generation.

2.3.3.1 Indexing and File Loading

In the *file loading and indexing stage*, both the reference and read sequences are loaded into memory. Between pattern p and text t, exact matches that cannot be further extended in both directions are *maximal exact matches*. Compared to seeds with pre-defined length, *BWA-MEM*'s key feature of variable seed length reduces each seed's mapping positions onto a reference genome. As invalid seed extensions are prevented, the speed of *BWA-MEM* is improved. FMD-index, a new index structure, facilitates the detection of all MEMs with an *80%* speedup, which indexes both the forward and the reverse strand DNA [20]. *FMD-index* is similar to the bi-directional BWT [61] used by SOAP and Bowtie2. The efficiency of generating MEM seeds plays a key role, where the frequently used strategy is indexing a sequence in a *full-text suffix tree*. Even with improved index structure, the *full-text suffix tree* still has a high memory usage as it stores every position of the text. *Space-sparse suffix array* was introduced to replace the *full-text index in the suffix tree* to reduce memory usage as it only stores every k^{th} position of the text. As read is commonly larger than the computer memory, read is separated into multiple blocks and processed one by one. After the first block's successful alignment, the next block is loaded for the next alignment section.

2.3.3.2 Seeding and Re-seeding

In the *seeding and re-seeding stage*, the *canonical seed-and-extend paradigm* is used in *BWA-MEM* [20]. The *canonical seed-and-extension paradigm* finds the exact matches and then extend the seed to the non-seed fragments within the selected candidate regions in the query read and the reference genome [62]. An algorithm is initially used to seed an alignment with *supper-maximal exact matches (SMEMs)*[31]. *SMEM* decreases time consumption by reducing the most invalid extensions of all other MEM in the read. When reads cannot be aligned by extension using SMEM, *BWA-MEM* uses a re-seeding process to generate new

[31]The longest MEM covering the position without overlapping.

seeds. In re-seeding, by default, when *SMEM*'s length is larger than 28 bp, the longest *MEM* covering the middle of the *SMEM* is used to initialize.

2.3.3.3 Chaining and Chain Filtering

In the chaining and chain filtering stage, seeds that are colinear to each other would be greedily chained together. The short chains in a long chain[32] are filtered out. At a later step, unsuccessful seed extension is reduced by chain filtering. However, chains detected are not accurate, which may not correspond to a final hit.

2.3.3.4 Seed Extension

In seed extension, seeds are ranked by the chain length and seed length. The seed is dropped if it is already contained in a previously found alignment. *BWA-MEM* differs from the standard seed extension as the extension stops when the score difference between the best alignment and the current alignment is larger than the pre-defined value. This process would avoid extension through a poorly aligned region. The pre-defined value is further adjusted by the number of gaps in the alignment. The algorithm accepts an alignment as a successful mapping between reads and references if the whole read is reached by extension, and the best improvement alignment score is larger than a pre-defined value.

Also, *BWA-MEM* traces the best extension score when reaching the end of the query. Even if a higher score is achieved, this strategy rejects the local alignment when the difference between the *best end-to-end alignment score* and the *best local alignment score* is less than a pre-defined value[33].

[32]Both *50%* and *38 bp* shorter than the long-chain.

[33]This process is used to choose between *local* and *end-to-end alignments* automatically.

2.3.3.5 Output

After every step completes, the raw output is produced and ready to be processed with pre-defined parameters in the output generation stage.

2.4 Parallel Computing

As most software executes instructions in sequence, the hardware quickly reaches limits as only one instruction is executed at any given time [63]. Since the room for speed improvement from a computational algorithm approach is limited, the alternative approach is to parallelize the existing algorithm. As the parallelization concept is dated back to the 1950s-1960s, the parallel accelerators have become prominent and ubiquitous only recently. The development of parallel computing has a substantial impact on software/hardware design. The trend's essence can be attributed to the physical limits of further increasing the operating frequency of processors and the shifted focus on integrating more computing units on one chip.

Driven by the trend, commercial parallel accelerators[34] have become commonplace in computing systems. Due to the multi-core processors' massive computational powers, a wide variety of dense matrices and vectors-based applications has been parallelized. These algorithms that mainly focus on solving linear algebra, stencil computations, and image processing have been extensively investigated. As hardware becomes cheaper and the distributed network becomes widely acceptable, they are being used to further speed-up computation-intensive applications[35].

[34]Such as multi-core *CPUs* and *GPUs*.

[35]Still, many irregular algorithms or irregular data structures problems cannot be paralleled. These applications do not exhibit enough static and runtime parallelism, which make them hard to parallelize.

2.4.1 Definition of Parallel Computing

Parallel computing uses multiple processors to execute the instructions from the same algorithm, making the time consumption a fractional of the originals. A great example of parallel computing could be the relation between workers and woods for a wood chopping job in a limited area. If only one worker is allowed at any given time, the time consumption is relatively high compared to two workers. The processing speed may be increased in a linear relationship until they start to affect each other as resources are limited, where the effect of speed improvement starts to diminish.

2.4.2 Classification of Parallel Computing

The four computers or processors classifications are *SISD, SIMD, MISD, and MIMD* [64]. SISD stands for a single instruction stream and single data stream, which means only one stream of instructions is used on one data stream. One example of SISD would be a traditional computer, where only one single thread is allowed. However, this type of computer system no longer exists in daily life as a multi-core processor gets cheaper.

SIMD stands for a single instruction stream and multiple data stream, where the data sets are distributed across multiple processors. MISD stands for multiple instruction streams and a single data stream, where it has many function units being performed on the same data. Examples of MISD includes shuttle flight computers, GPUs and general wavefront processors. MIMD stands for multiple instructions with the multiple data stream, where different instructions are performed on different data. MIMD CPU is commonly used for systems needing high calculation power, such as servers and cluster computers. An example of such a processor is the Intel Xeon Phi server chip, which has up to 72 cores and 288 threads.

2.4.3 CPU multi-threaded

In modern operating systems, a process is defined as an entity that *groups resources together* [65]. multi-threaded is either provided by a single control processing unit (CPU) or a single core within a multi-core processor. Within the single core, multiple processes are executed simultaneously as the processor quickly switches back and forth between multiple processes[36]. This setting provides an illusion of parallelism, but it is not true parallelism. In a more modern setting, a process can have multiple threads of control, where each thread is independent of others [65]. Both *BWA-MEM* and *BWA-ALN* provide such functionality in their original package.

2.4.4 CPU Distributed

With the two types of parallelization systems, *shared memory multiprocessor system (SMP)* is interconnected through shared physical memory, and *message-passing multicomputer (MPM)* is interconnected through a network connection with message-passing libraries. SMP is a specially designed machine as processors have to be directly connected, such as mesh and hypercube. However, in MPM, a computer cluster is a loosely connected network containing many regular workstations together through protocols.

2.4.4.1 Message Passing Interface

Message Passing Interface (MPI) is a widely used communication protocol for parallel computing architectures, which supports both point-to-point and collective communications [66]. *MPI* provides the message-passing application programmer interface, allowing programmers to use high-performance message passing options on advanced machines. In 2012, Peters et al.[9] developed pBWA on Compute Canada's cluster using *MPI*, which is considered the first efficient parallel version of *BWA*. However, as *pBWA* was a modified

[36]Only one task is being executed at any given time.

version of *BWA-ALN* in 2011, it is soon outdated as a newer version of *BWA-ALN* comes out.

2.4.4.2 Big Data

big data framework is a collection of open-source framework for distributed storage and data processing[37]. *Hadoop* is the most successful open-source implementation of the *MapReduce* programming model. *Hadoop* supports large data sets shared across clusters using the *MapReduce* framework, which is designed to scale up from a single node to thousands of nodes, where each node offers computation power and local storage [67]. The *MapReduce* framework is designed to handle node failures at the application layer.

BigBWA (BWA + Hadoop) [14] and SparkBWA (BWA + Spark) [16] are the tools that use big data technology to boost the performance of *BWA*. Important reductions in the execution times were observed when using both tools. In both BigBWA and SparkBWA, no modifications to the original *BWA-ALN* source code is required[38], which assures its compatibility with any *BWA-ALN* version[39].

2.4.4.3 GPU multi-threaded

CPUs like Intel Core series are good at doing a few tasks concurrently. In comparison, GPGPU contains many arithmetic logic units[40], which enables millions of threads to be launched at the same time. In simple and computation-intensive work, GPGPU is more powerful and cost-efficient than an equivalent CPU.

[37]There are few implementations of combining big data with BWA, which includes SparkBWA and BigBWA.

[38]As they are using two independent software layers.

[39]Future or legacy.

[40]Figure 2.5 shows a generic GPGPU architecture.

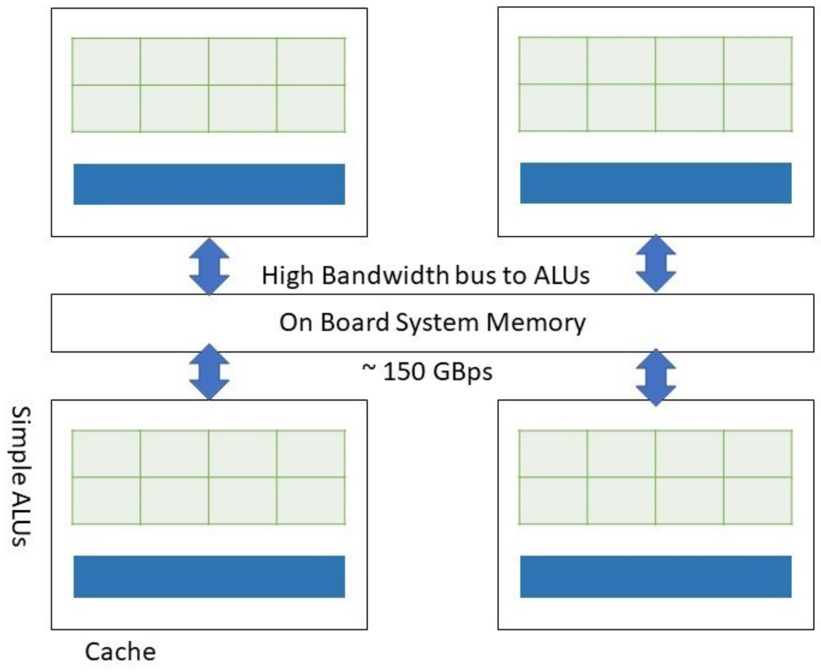


Figure 2.5 A generic GPGPU architecture. In comparison to a CPU, a GPGPU works with fewer, and relatively small cache layers. GPGPU has more transistors dedicated to computation and it cares less how long it takes to retrieve data from memory.

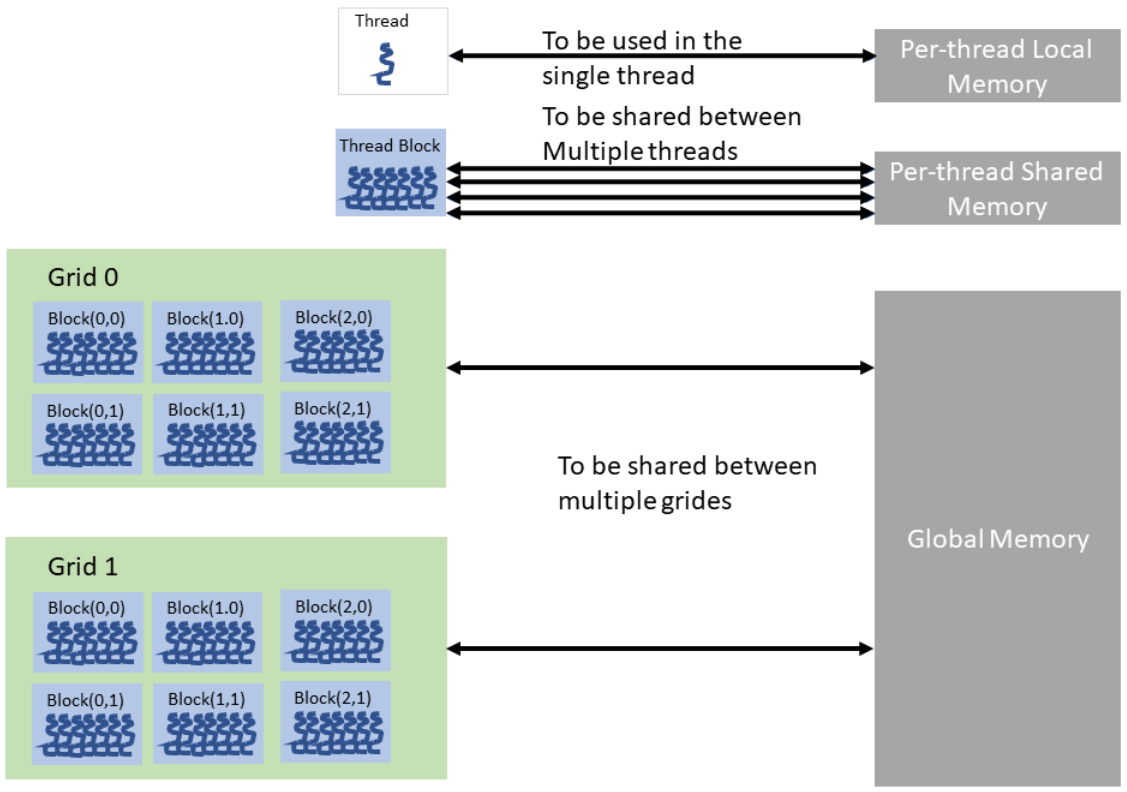


Figure 2.6 Memory hierarchy in CUDA. GPGPU contains multiple grid, and each grid contains multiple thread block. A thread block is a programming abstraction that presents a group of threads that can be executed serially or in parallel.

A single GPGPU contains multiple of computation units (or blocks), and each blocks containing multiple threads. Figure 2.6 is the memory hierarchy in *GPU architecture*. Like CPU memories have *three different cache levels*, GPGPU also has its memory hierarchy with *different data transfer speeds*, including local memory[41], shared memory[42], and global and texture memory. All the threads share *global memory*, *constant memory* and *texture memory*. Threads also have access to multiple registers and local memory at per thread level.

Compared to the local/global memory, the shared/register memory is around *150 times*

[41] At per-thread level.

[42] At per-block level.

faster. *Registers* are the fastest form of memory on the GPGPU [68]. The threads can only access a parallel data cache or shared memory within the same computation unit, and is as fast as a register when there are no bank conflicts or when reading from the same address. The global memory is slow and uncached, commonly used for massive memory transfer, especially when transfers in and out of GPGPU. Texture memory (read-only) is cache optimized for 2D access, and is used to store textures during *3D processing* [68]. The constant memory (read-only) is slow, but it is cached and shared with all threads. The local memory stores the data that does not fit into registers, and is slow but cached. If one does not need to modify the variables, the read-only memories are the best option. The shared/register memory should be used as much as possible to ensure fast access. When the local memory and shared memory are used, synchronization is not needed as no race condition can occur. However, the user has to use __*syncthreads()* keyword to synchronize the thread to avoid deadlocks when dealing with global memory and shared memory. The shared variables are declared with __*shared*__ keyword. The global memory is declared with __*device*__ keyword, and *cudaMelloc()* is used to copy a chunk of memory from the host to the global memory.

The two main components of the CUDA program are host and kernel, where the host is responsible for moving data between host and device, and the kernel is a parallelized code meant to be executed concurrently on the GPGPU device. The version number represents the computing capability of a device, which helps to determine what features are supported. CUDA allows for three hardware parallelization levels, where kernel functions are executed on the grids of threads and blocks.

When designing a GPGPU program, high overhead on data transferring must be considered, especially data transfer between host and device. Also, the data transferring speed within the GPGPU is varied when using different types of memories. Therefore, *Block shared memory* is used for calculation, and *texture memory* stores pre-calculated information. *global shared memory* stores the input and the final result. However, unlike other programs, the

GPGPU program has to track memory usage as the hardware does not track how much memory is used. The device would terminate the program, or worse, crash, once the program exhausts memory within the GPU.

3 RESEARCH DESIGN

This chapter presents the design idea of *GPGPU distributed BWA-MEM* and methodology used. After successfully understood how *BWA-MEM* works, we need to find out which part of the code takes most of the time, that's where hot spot analysis comes in. As both *BWA-MEM* and *BWA-ALN* follow the seed-and-extend paradigm, the GPGPU *BWA-ALN* is investigated. After analyzing different parallelization techniques, we find that the distributed GPU *BWA-MEM* is the fastest among other parallelization techniques.

Two hot spot analysis are performed at *100 bp* and *250 bp* [21], where *ksw_ extend2* has increased from *28.9%* to *51.9%* of the total execution time. We can conclude that *ksw_ extend2* increases as the length of the alignment increases. therefore, our main goal is to parallelize *chain2aln*.

As the *ksw_ extend2*, part of the *chain2aln*, can be considered as a simplified smith-Waterman algorithm, the *ksw_ extend2* follows the typical Wave-front technique. The *BWA-MEM* seed extension is parallelized at three different levels. At parallelization *Level 1*, function *ksw_ extend2* is parallelized, which performs seed extension on the single side. At parallelization *Level 2*, function *ksw_ extend2* is parallelized, which performs seed extension on both side of the seeds within the same chain. At parallelization *Level 3*, function mem_align1_core is parallelized, which performs seed extension in all chains. Under the assumption of enough GPGPU resources, the time complexity would remain at $O(L_0 + L_1)$[1].

A *Spark* version of *BWA-MEM* has two layers. The first layer is responsible for execut-

[1]This is for $sequence_0$ and $sequence_1$.

ing *GPGPU distributed* software, and Java native interface (JNI) as an adapter to communicate with layer two. Layer two is the core of the execution. This part of the program is responsible to get user input and then perform map and reduce. it will set up RDDs, map data, and then reduce them into a single file.

3.1 Technical Road-Map

Table 3.1 Types of parallelization techniques. The parallelization techniques can be categorized into four types: *CPU multi-threaded*, *CPU distributed*, *GPU multi-threaded*, and *GPU distributed*.

	CPU	GPU
multi-threaded	POSIX Thread	CUDA
Distributed	Spark, Hadoop, OpenMPI	Spark, Hadoop, OpenMPI

Table 3.2 The description of each parallelization type's benefits.

Versions	**Benefits**	**Drawbacks**
CPU Multi-threaded	Most software comes with CPU multi=threaded version, which can be run on most single desktops.	CPU multi-threaded is considered as slow, and not able to use GPU resources.
CPU Distributed	CPU distributed is usually easy to set up and run as most clusters support such type. The most used two types of CPU distributed, OpenMPI and Big Data, the first one is widely supported by many clusters as it is a mature technology, but the later one has the benefit of better resource management, especially in data management.	When software is converted into CPU distributed, OpenMPI based software needs to be updated when the original version of the software changed. On the other hand, Big Data software is version agnostic. However, Big Data software is not widely supported by clusters as they may cause conflicts with the existing resource management system.
GPU Multi-threaded	GPU Multi-threaded is typically faster than CPU multi-threaded when the algorithm can be parallelized.	GPU multi-threaded needs to run on a GPU equipped device, which brings up the cost of hardware.
GPU distributed	By using multiple nodes, GPGPU distributed can speed up the performance of the software by using multiple nodes rather than the single-node version in GPU Multi-threaded. Also, GPU distributed can be made as it is version agnostic towards GPU multi-threaded.	GPU distributed software could be considered as a combination of GPU multi-threaded and distributed framework, such as a combination of OpenMPI and Big Data). Therefore, its drawback is a combination of CPU distributed and GPU distributed. The costs are further increased as a GPU capable cluster is needed.

The four types[2] of parallelization technique are CPU multi-threaded [8], [9], GPGPU parallelized [10]–[14], CPU distributed [15], [16], and *GPGPU distributed* [17]–[19]. To understand what the best option may be, analyzing the benefits and drawbacks of each technology was necessary.

Among all parallelization techniques, *GPU-based software* are generally considered faster than *CPU-based software*, and they solve problems with high-level parallelization effectively and efficiently. The two commonly used GPGPU languages, CUDA and OpenCL, are designed to simplify the GPGPU related operations. CUDA stands for Compute Unified Device Architecture and is created to simplify NVIDIA-related operations. OpenCL stands for Open Computing language and is supported by multiple GPGPU types, such as AMD and Intel. NVIDIA has stopped supporting the OpenCL framework for some time. NVIDIA CUDA is the core for many frameworks and libraries that needed a high performance, such as TensorFlow and OpenCV. As a typical example, *NVIDIA GTX 1080*, a gaming GPU, owns *2058 cores*. In most cases, a single or a small group of connected GPGPU can solve a problem faster than a multi-core CPU.

There are a few reasons for *GPGPU distributed BWA-MEM* version to has a high chance to outperform existing *GPGPU distributed* BWA-MEM version implementations. First, a good performance is already shown in the GPGPU multi-threaded version and *GPGPU distributed* version of *BWA-ALN*, indicating distributed GPU *BWA-MEM* having a good odds to outperform other parallelized versions. *BarraCUDA* is a GPGPU paralleled version of BWA-ALN. The paper on *BarraCUDA* claimed it is up to three times faster and *60%* more accurate than BWA-ALN [10]. It is reported that *BarraCUDA* can align short paired-end NextGen sequences up to ten times faster than *BWA* when it runs on a GPGPU 12 core K80 Tesla server [10].

Big data *BWA-ALN*, as a distributed CPU version, such as the SparkBWA, is *2.5X* faster than *BWA-ALN* with 64 mappers [16]. *SparkBWA* is *1.4x* faster than pBWA [16].

[2]Table 3.1 shows the different techniques for parallelization, and Table 3.2 shows each of their benefits

Since both parallelized *BWA-MEM* versions have various speed-ups, the distributed version of GPGPU *BWA-MEM* has a high chance to outperform existing tools. There are few implementations of combining big data with *BWA-ALN*, which includes SparkBWA and BigBWA. *Hadoop* is the most successful *open-source* implementation of the *MapReduce programming model* introduced by Google.

Rather than relying on hardware to deliver high availability, the big data library is designed to detect and handle hardware failures at the application layer, which provides high availability on top of a cluster of computers. *BigBWA* (*BWA-ALN* + Hadoop), *SparkBWA* (*BWA-ALN* + Spark) are new tools that use the big data framework to boost the performance of the BWA-ALN. The reductions in the execution times were observed when using both tools. The design of both BigBWA and SparkBWA has two independent software layers, which ensures no modifications is needed towards the original *BWA-ALN* source code (version agnostic) [15], [16].

Existing studies have shown that *GPGPU distributed* tools have a better performance than others. *GPGPU distributed* BLAST improves the performance of *BLASTP* on a single GPGPU with high availability and fault tolerant [69]. BLASTP claims it is *1.5x* faster than Hadoop-BLAST [19]. Therefore, the *GPGPU distributed* structure could boost *BWA-MEM* under the correct implementation. However, *BWA-MEM* does not have any open source GPGPU multi-threaded version available, which is the key component for developing *GPGPU distributed*. Therefore, GPGPU multi-threaded *BWA-MEM* needs to be built first.

GPGPU distributed *BWA-MEM* version, especially under the big data framework, just like big data *BWA-ALN* versions mentioned previously, can be version agnostic. GPGPU distributed is designed with two layers, where the first layer corresponds to GPGPU software package, and the second layer is responsible for executing MapReduce framework. However, OpenMPI GPGPU *BWA-MEM* version can not achieve this goal as modification towards BWA-MEM itself is needed. *GPGPU distributed* is the future of parallel computing as cloud computing become popular. In order to gain further speedup, scientists rely on large GPGPU clusters[3].

The big data GPGPU *BWA-MEM* consists of three main stages: *resilient distributed dataset (RDD) creation*, *map*, and *reduce.* In the *RDD phase*, input data is uploaded onto *RDD*. Then, *map phase* carries out the actual alignment process. *RDD* is a *read-only multi-set* of data items distributed over a cluster, which is maintained in a fault-tolerant way [70]. The *Map phase* uses the *parallelized GPGPU BWA-MEM* to perform the alignment. In the *reduce phase*, all the produced files are combined into a single output file.

[3]Cloud service providers such as Amazon, Google, and Tencent offer high-performance GPGPU clusters at a relatively lower price.

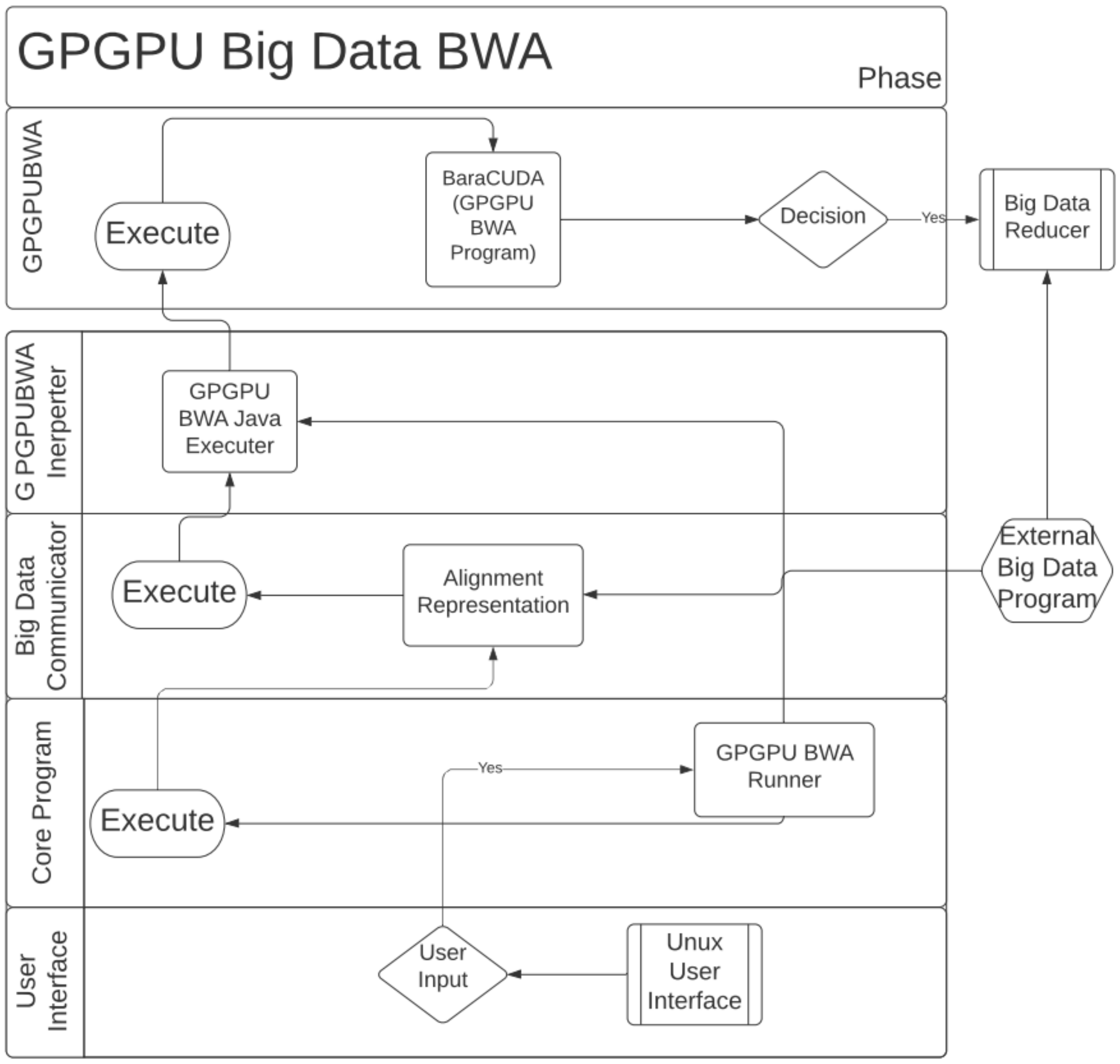


Figure 3.1 A simplified version of big data *GPGPU distributed BWA-MEM* framework. The big data *GPGPU distributed BWA-MEM* framework has two main layers. Layer one contains a GPGPU version of *BWA-MEM*, and the layer two contains *MapReduce* framework.

Table 3.3 The description of the simplified version of big data *GPGPU distributed BWA-MEM* framework.

Layer one	Layer one is responsible for executing GPGPU BWA software, which contains some adapters and unmodified GPGPU software. This layer is in C/C++ language. A GPGPU BWA is needed for this layer.	
	GPGPU BWA	GPGPU BWA
	Adapters	The adapter is created using Java Native Interface (JNI). It is responsible to communicate with program in layer two.
Layer two	This part of program is responsible to get user input and then perform RDDs creation, map, and reduce. The layer is implemented in Java language.	
	GPGPU BWA Interpreter	The interpreter allows execution of GPGPU program using JAVA Native Interface (JNI).
	Big Data Communicator	This part is composed of alignment representation. Its major task is to communicate with external Big Data program.
	Function	Basically, this part is the core of execution. It will set up RDDs, map data, and then reduce them into a single file.
	User Interface	User Interface is responsible for obtaining file path, and other inputs.
External Software	This part is external software packages, including Big Data and its related set-up, which includes Spark, Hadoop, Hadoop Yarn server.	

Figure 3.1 is the designed software structure for *GPGPU distributed* BWA, followed by Table 3.3 as an explanation. Layer one (*GPGPU BWA Layer*) contains an *unmodified GPGPU software*. *Layer two* is the *MapReduce framework* built upon layer one, and the execution of *layer one* depends on *JAVA Native Interface* (JNI). The *MapReduce* framework handles data communication on a GPGPU Cluster.

To implement GPGPU distributed BWA-MEM, GPGPU multi-threaded *BWA-MEM* and a suitable cluster is needed. After researching the GPGPU multi-threaded *BWA-MEM*, there are no open-source versions available. Spark and Hadoop, the two Big Data framework, both require additional resource manager before September, 2020[4]. In the most recent *Spark*

[4]The additional resource manager is for managing GPU resources, which will cause a conflict with existing

release allows us to run the GPGPU cluster in Spark standalone mode. Thus, *Spark* framework stood out and caught our attention.

3.2 Hardware Setup

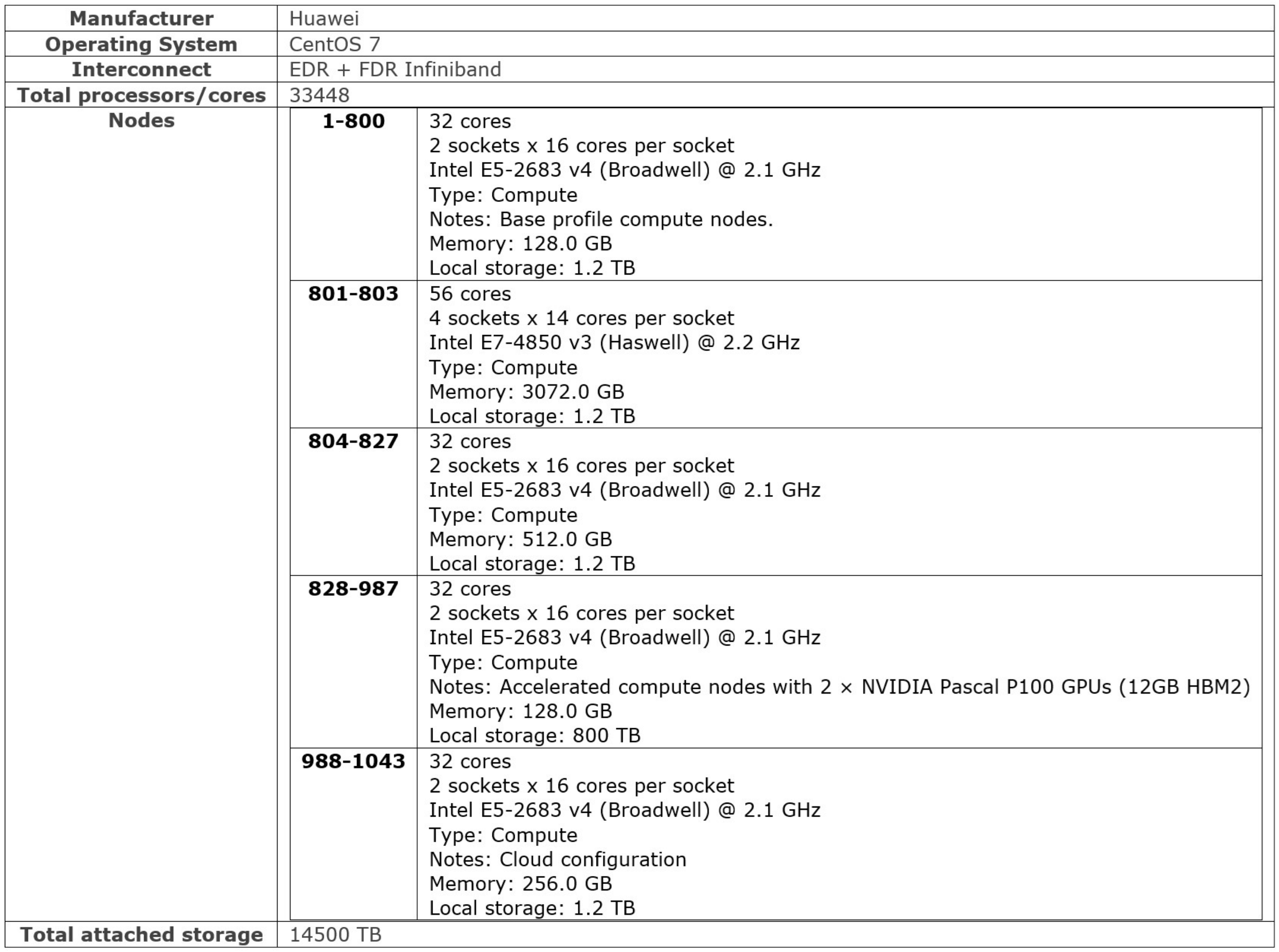
This Section describes the effort in looking for a suitable cluster to build and run *GPGPU distributed BWA-MEM*. The first step is to analyze how the big data cluster interacts with BWA, especially how it is transferred within the cluster. There are four options available: 1) *SharcNET*; 2) personal PC; 3) commercial cloud services; 4) *Brock University*. and each of them is described below.

3.2.1 SharcNET

SharcNET is a consortium of universities in Ontario, that aggregate funding to purchase super-computer systems, which are shared among their members [71]. As *Brock University* is a member of *SharcNET*, the four such clusters on *SharcNET* that may be suitable for the program are 1) copper cluster [72] (decommissioned on March 29, 2019); 2) vdi-centos6 cluster [73] (one node is available); 3) mosaic cluster[74]; 4) graham cluster [75]. Both mosaic and graham clusters provide a *full scale* cluster with NVIDIA GPGPU installed, but graham is with newer hardware.

software on SharcNET.

Table 3.4 Hardware specification for graham cluster[1].



Manufacturer	Huawei	
Operating System	CentOS 7	
Interconnect	EDR + FDR Infiniband	
Total processors/cores	33448	
Nodes	**1-800**	32 cores 2 sockets x 16 cores per socket Intel E5-2683 v4 (Broadwell) @ 2.1 GHz Type: Compute Notes: Base profile compute nodes. Memory: 128.0 GB Local storage: 1.2 TB
	801-803	56 cores 4 sockets x 14 cores per socket Intel E7-4850 v3 (Haswell) @ 2.2 GHz Type: Compute Memory: 3072.0 GB Local storage: 1.2 TB
	804-827	32 cores 2 sockets x 16 cores per socket Intel E5-2683 v4 (Broadwell) @ 2.1 GHz Type: Compute Memory: 512.0 GB Local storage: 1.2 TB
	828-987	32 cores 2 sockets x 16 cores per socket Intel E5-2683 v4 (Broadwell) @ 2.1 GHz Type: Compute Notes: Accelerated compute nodes with 2 × NVIDIA Pascal P100 GPUs (12GB HBM2) Memory: 128.0 GB Local storage: 800 TB
	988-1043	32 cores 2 sockets x 16 cores per socket Intel E5-2683 v4 (Broadwell) @ 2.1 GHz Type: Compute Notes: Cloud configuration Memory: 256.0 GB Local storage: 1.2 TB
Total attached storage	14500 TB	

[1]Retrived from https://www.sharcnet.ca/my/systems/show/114.

Table 3.4 describes the hardware specification of the graham cluster. The nodes from *828* to *987* are equipped with 2 *NVIDIA Pascal P100* GPUs, 128 GB of memory, and total local storage of 800 TB. For implementing a *GPGPU distributed BWA-MEM* onto the graham cluster, the nodes from *801* to *803* can be used as a RAM drive (as each of them has *3072* GB of RAMs), and nodes from *828* to *987* can be used as a computation cluster.

Table 3.5 Hardware specification for mosaic cluster[1].

Mosaic Cluster Hardware Specification		
Operating System	CentOS 6	
Interconnect	QDR InfiniBand	
Total processors/cores	528	
Nodes	**1-20**	20 cores 2 sockets x 10 cores per socket Xeon E5-2680 v2 @ 2.8 GHz Type: Compute Notes: Each node has one NVIDIA Tesla K20m GPU installed. Run time limited to four (4) hours for non-contribution users. Memory: 256.0 GB Local storage: 200 GB
	21-24	32 cores 4 sockets x 8 cores per socket Intel Xeon E5-4650 @ 2.7 GHz Type: Compute Notes: Run time limited to four (4) hours for non-contribution users. Memory: 768.0 GB Local storage: 200 GB
Total attached storage	4.69 TB	

[1]Retrived from https://www.sharcnet.ca/my/systems/show/106.

Table 3.5 describes the hardware specification of the mosaic cluster. Within the cluster, there are two types of equipment, nodes from *1* to *20* with *NVIDIA Tesla K20m GPU* and nodes from *21* to *24* with no GPGPU resources but with a high amount of RAM, where the second one can be treated as a storage cluster.

Table 3.6 **Software specification for graham cluster**[1].

Mosaic Cluster Software Specification		
BarraCuda	**NVIDIA CUDA toolkit SDK version 6 or above**	5.5.22, 6.5.14, 6.0.37, 7.5.18
	GCC and G++ Version 4.5 or above	8.2.0 (el6) , 6.3.0 (el6), 5.3.0 (el6), 8.1.0 (el6), 5.5.0 (el6), 4.8.4 (el6), 4.9.3 (el6), 5.1.0 (el6), 4.8.1 (el6), 4.3.4 (el5), 6.4.0 (el6), 4.8.2 (el6), 4.9.2 (el6), 7.3.0 (el6), 6.1.0 (el6), 6.2.0 (el6), 4.9.4 (el6), 5.4.0 (el6), 4.8.5 (el6), 7.1.0 (el6), 7.2.0 (el6)
	NVIDIA graphics driver version 340 or above	NVIDIA-SMI 352.93
	zlib-devel (or zlib1g-dev) library	zlib-1.2.11
SparkBWA	**MAVEN 3.6.0 or above (for software compiling)**	MAVEN is not on the system, however, we can compile it offline.
	Spark and Hadoop (for database system)	Hadoop is not on the system
	JAVA JRE	OpenJDK version "1.8.0_91" OpenJDK Runtime Environment (build 1.8.0_91-b14) OpenJDK 64-Bit Server VM (build 25.91-b14, mixed model)

[1]Retrived from https://www.sharcnet.ca/my/systems/show/106.

Table 3.6 describes the software installation status of the *mosaic cluster*. The *mosaic cluster* is installed with CUDA toolkit, GCC and G++, NVIDIA graphics driver, Zlib, and Java. However, *BarraCUDA* cannot be compiled without MARVEN installed on the mosaic cluster. As the *Hadoop system* is not installed, the big data cluster is not supported on the cluster. Technical support staff at *SharcNET* indicated that it is impossible to install any new task scheduler (Hadoop and *Spark* require a different task scheduler) as they are conflicting with the existing one unless it does not require installation. *Spark* standalone cluster can be setup without installation, but it has not yet come to support GPGPU resources.

Table 3.7 Software setup process on Mosaic Cluster.

We use following commands to load supported software.
ssh mos1
module unload intel
module load gcc/4.9.4
gcc -v
module load cuda/7.5.18
which nvcc
module load spark/python2714/2.3.0
spark-submit --version
module load zlib

Table 3.7 describes the process to setup the necessary environment. In order to gain full access, the user needs to submit a request to acquire related resources. Also, *SharcNET* limits each user's workload by limiting execution with a time restriction (which is done through a task scheduler). The command *module load* is used to load necessary software packages and specific versions onto the system, and the command *module unload* can unload separate software packages from the environment.

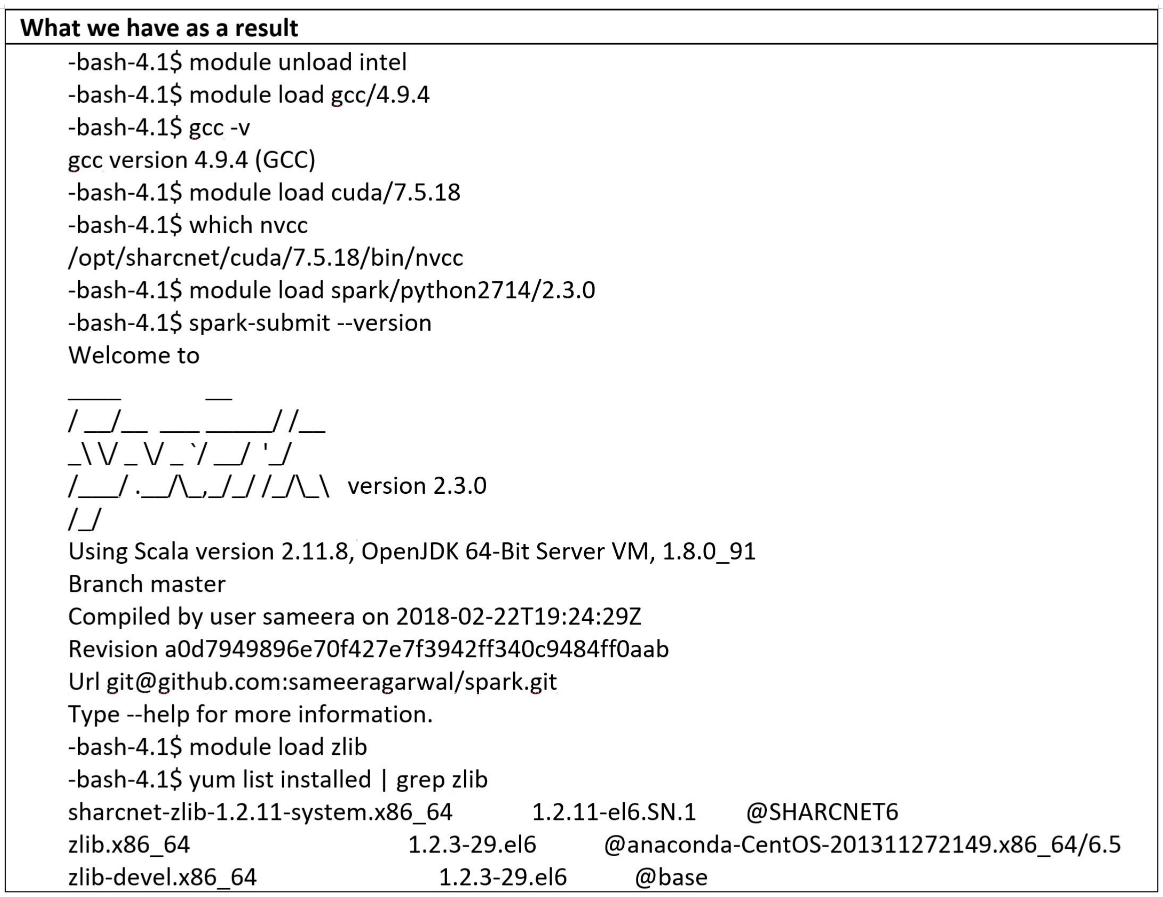
Table 3.8 Output of software setup on Mosaic cluster

What we have as a result

```
-bash-4.1$ module unload intel
-bash-4.1$ module load gcc/4.9.4
-bash-4.1$ gcc -v
gcc version 4.9.4 (GCC)
-bash-4.1$ module load cuda/7.5.18
-bash-4.1$ which nvcc
/opt/sharcnet/cuda/7.5.18/bin/nvcc
-bash-4.1$ module load spark/python2714/2.3.0
-bash-4.1$ spark-submit --version
Welcome to
____          __
/ __/__  ___ _____/ /__
_\ \/ _ \/ _ `/ __/  '_/
/___/ .__/\_,_/_/ /_/\_\   version 2.3.0
/_/
Using Scala version 2.11.8, OpenJDK 64-Bit Server VM, 1.8.0_91
Branch master
Compiled by user sameera on 2018-02-22T19:24:29Z
Revision a0d7949896e70f427e7f3942ff340c9484ff0aab
Url git@github.com:sameeragarwal/spark.git
Type --help for more information.
-bash-4.1$ module load zlib
-bash-4.1$ yum list installed | grep zlib
sharcnet-zlib-1.2.11-system.x86_64          1.2.11-el6.SN.1       @SHARCNET6
zlib.x86_64                          1.2.3-29.el6         @anaconda-CentOS-201311272149.x86_64/6.5
zlib-devel.x86_64                        1.2.3-29.el6         @base
```

Table 3.8 describes the output after running the set-up code. As the output has shown, the *GCC compile version* is *4.9.4*, and the *Spark* version is *2.3.0*. The *Zlib* versions are *1.2.11*, *1.2.3-29*, which means it has the required version of *Zlib* by BarraCUDA.

Table 3.9 SparkBWA environment setup on mosaic.



What we have as a result

```
-bash-4.1$ module unload intel
-bash-4.1$ module load gcc/4.9.4
-bash-4.1$ gcc -v
gcc version 4.9.4 (GCC)
-bash-4.1$ module load cuda/7.5.18
-bash-4.1$ which nvcc
/opt/sharcnet/cuda/7.5.18/bin/nvcc
-bash-4.1$ module load spark/python2714/2.3.0
-bash-4.1$ spark-submit --version
Welcome to
____          __
/ __/__  ___ _____/ /__
_\ \/ _ \/ _ `/ __/  '_/
/___/ .__/\_,_/_/ /_/\_\   version 2.3.0
/_/
Using Scala version 2.11.8, OpenJDK 64-Bit Server VM, 1.8.0_91
Branch master
Compiled by user sameera on 2018-02-22T19:24:29Z
Revision a0d7949896e70f427e7f3942ff340c9484ff0aab
Url git@github.com:sameeragarwal/spark.git
Type --help for more information.
-bash-4.1$ module load zlib
-bash-4.1$ yum list installed | grep zlib
sharcnet-zlib-1.2.11-system.x86_64          1.2.11-el6.SN.1       @SHARCNET6
zlib.x86_64                        1.2.3-29.el6         @anaconda-CentOS-201311272149.x86_64/6.5
zlib-devel.x86_64                      1.2.3-29.el6         @base
```

Table 3.9 describes SparkBWA's environment setup on mosaic.

3.2.2 Personal PCs

There are two advantages of setting up a test environment on personal computers. Firstly, the work can be started while looking for a cluster. Secondly, we have the full control of the system configuration, and software installation is available.

Table 3.10 Offline PC 1's specification.

Operating System	CentOS 7.6
Interconnect	Stand Alone
Total processors/cores	8
Nodes	**1** 8 cores 1 sockets x 8 cores per socket Intel Core i7-6700K CPU @ 4.00 GHz 4.01 GHz Notes: Each node has one NVIDIA GTX 1080. Memory: 32.0 GB. Local storage: 1 TB.
Total attached storage	N/A

Table 3.10 describes the specification of *offline PC 1*. *Offline PC 1* is a home desktop with an *Intel Core i7 6th Gen* equipped with a *dedicated GPU*. It also has *32 GB RAM* and 1 TB *Local Storage* installed, which will be the primary desktop for testing. *Offline PC 1* meets the hardware specifications for both BarraCUDA, and our new software, *Spark* GPGPU *BWA-MEM*.

Table 3.11 Offline PC 2's specification.

Operating System	CentOS 7.6
Interconnect	Stand Alone
Total processors/cores	4
Nodes	**1** 4 cores 1 sockets x 4 cores per socket Intel Core i7 3ed Gen CPU Notes: each with NVIDIA GTX 620M. Memory: 16.0 GB. Local storage: 200 GB.
Total attached storage	N/A

Table 3.11 describes the specification of *offline PC 2*. *Offline PC 2* is a laptop equipped with an *Intel i7 3rd Gen* and an *on-board NVIDIA GPU*. However, laptop GPGPU uses *hy-*

brid technology, where the video output is a combined effort of both *integrated GPU*[5] and the *disintegrated GPU*. Also, the performance of this GPGPU is lower than the *offline PC 1*'s GPU. Furthermore, under the Linux environment, there is no graphics driver for this *combined GPU* available[6]. As the PC 1 has an advantage in terms of hardware, we performs test on this machine.

3.2.3 Cloud Services

In order to find a suitable cluster, the possibility of renting cloud services had been researched. The two most popular cloud service providers are *Amazon* and *Tencent*. Among them, the cheapest option is Tencent's server. setting up a testing cluster[7] is still costly, and we have estimated around *3,000 dollars* for a single month.

3.2.4 Brock University Department of Computer Science

Computer Science Department at *Brock University* has just setup new PCs for Master students in 2019, but the GPGPU equipment needs to be requested. Also, there is not enough GPGPU equipment available to setup a small GPGPU cluster.

3.2.5 Summary

A few options for possible hardware had been looked at. We did looked into *SHARCNET*, which did own few such clusters, but they lack the necessary software packages. A typical example is the *Mosaic cluster*. *Mosaic cluster* does have *Spark* installed on all the nodes but it lacks compatible resource manager for GPGPU equipment. We have looked at hardware owned by the author, but only (*offline PC 1*) is equipped with the necessary hardware. At last, the online paid services are costly.

[5]The GPGPU is inside the Intel CPU.

[6]NVIDIA does not provide a graphics driver for Linux.

[7]With minimum three GPGPU equipped nodes.

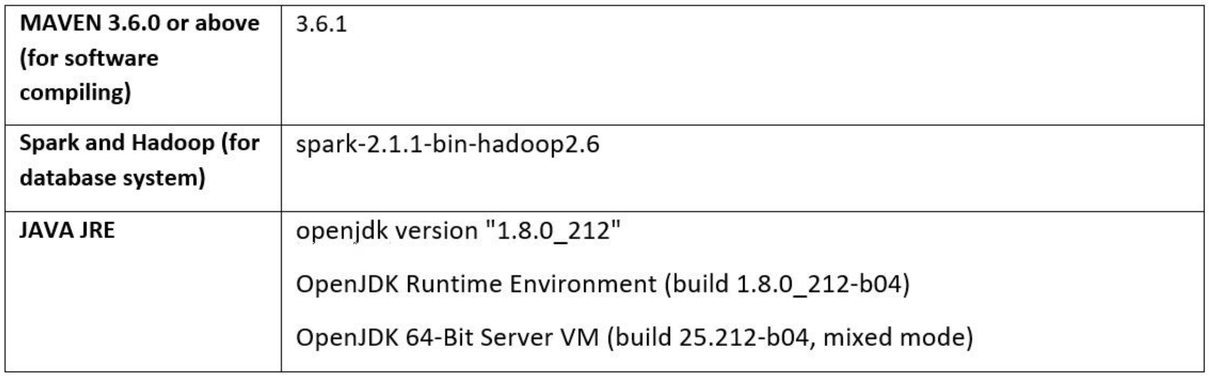
In this chapter, author mentioned configurations for clusters on *SharcNET*, PERSONAL PCs, labtop, and department facilities. Mosaic cluster is part of *SharcNET* computer network, and it is part of the Ontario's supper-computer system. The test facilities for laptop and personal PCs were tested at author's home. *Brock University*'s Facility is located at *Brock University*'s *MCJ* block. Due to the COVID 19 pandemic, the majority of tests was performed at *Brock University* and author's home on PC 1.

3.3 Software Setup

This Section introduces the installation process step by step.

3.3.1 SparkBWA

Table 3.12 SparkBWA environment setup on PC 1.



MAVEN 3.6.0 or above (for software compiling)	3.6.1
Spark and Hadoop (for database system)	spark-2.1.1-bin-hadoop2.6
JAVA JRE	openjdk version "1.8.0_212" OpenJDK Runtime Environment (build 1.8.0_212-b04) OpenJDK 64-Bit Server VM (build 25.212-b04, mixed mode)

Table 3.12 describes the installed software version for SparkBWA, where compatible MARVEN, Spark, *Hadoop* OpenJDK version has been successfully installed on PC 1[8].

[8]The test was performed on intel i7-6700k CPU, NVIDIA GeForce GTX 1080, 32GB RAM, and 1TB solid state hard drive desktop.

3.4 BWA-MEM GPGPU Parallelization

As we have analyzed how *BWA-MEM* interacts with big data, the next step is to parallelize *BWA-MEM* with GPU. *BWA-MEM* can be separated into five stages, which are 1) loading index and read file content into the program, 2) finding SMEM, 3) chaining and chain filtering, 4) seed extension, 5) generating the output file. However, due to the thesis' time limitation, it is impossible to start GPGPU parallelization for all parts. Therefore, a hot spot analysis from [21] is used to determine the code's highly used.

3.4.1 Hot Spot Analysis

Hot spot analysis determines a *high proportion of executed instructions region* within a computer program. As *BWA-MEM* is a large program, it is hard to determine which stage has taken a large amount of time. As mentioned in [21], *BWA-MEM*'s performance depends on the length of the read, and is very suitable for reads with *70 bp* length. In [21], a *hot spot analysis* on the average length of *100 bp* was performed [21] (Figure 3.2,) and another *hot spot analysis* was performed on the average length of *250 bp* for comparison. In both graphs, the pie represents the total execution time. As shown in Figure 3.2, *chain2aln* takes *28.9%* of total execution time. As the length of the read increased to *250 bp*, *chain2aln* has increased to *51.9%* of the total time. The test also verifies the fact mentioned from [76] that *seed extension* is the most time-consuming component of *BWA-MEM*.

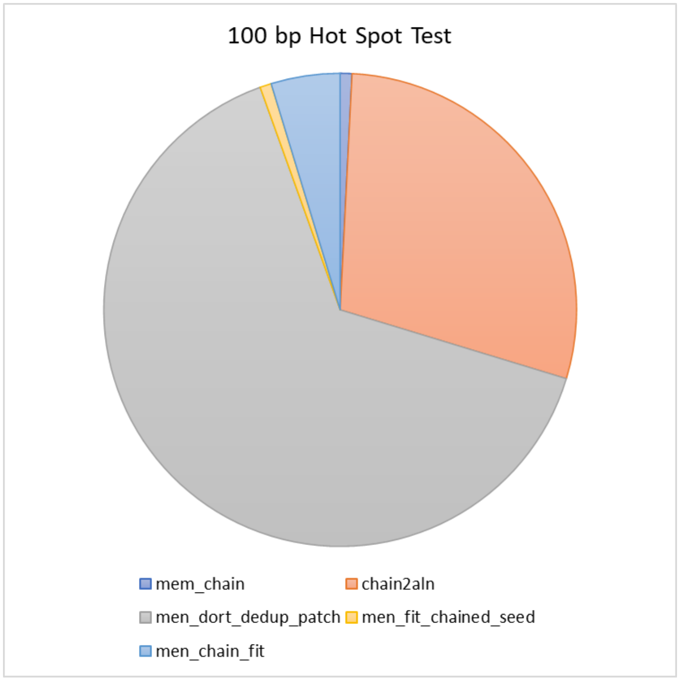


Figure 3.2 Hot spot analysis for *BWA-MEM* at a read length of 100 bp[1].

[1]Data is from [21].

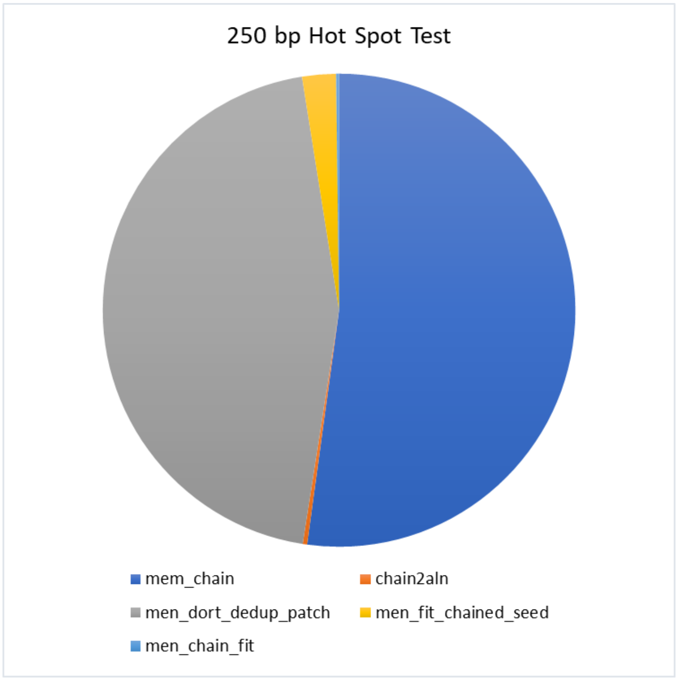


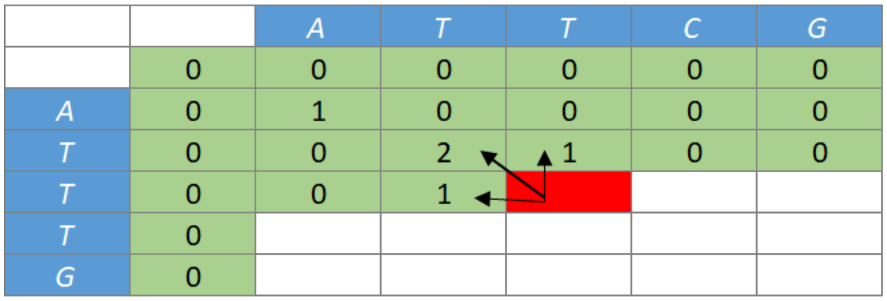
Figure 3.3 Hot spot analysis for *BWA-MEM* at a read length of 250 bp[1].

[1]Data is from [21].

We have determined that *chain2aln* is the most *time-consuming component* of the *BWA-MEM*. *chain2aln*, a modified version of the Smith-Waterman algorithm, is part of step 4 of *BWA-MEM*. The test result from *Level 1* showed that the *GPU multi-threaded version* of *ksw_extend2* is not getting any faster than the original CPU version, which is due to an insufficient parallelization level. Therefore, the parent functions, *chain2aln*, and mem_align1_core, are investigated.

3.4.2 Smith-Waterman Algorithm

The *Smith-Waterman algorithm* is a *dynamic programming algorithm* that determines similar regions between two strings by comparing all possible length segments. As a dynamic programming algorithm, it is guaranteed to find the optimal local alignment with the scoring system being used. The *Smith-Waterman algorithm* has four steps: *substitution matrix gap* *penalty scheme determination*, *scoring matrix initialization*, *scoring*, and *traceback*.



		A	T	T	C	G
	0	0	0	0	0	0
A	0	1	0	0	0	0
T	0	0	2	1	0	0
T	0	0	1	🟥		
T	0					
G	0					

Figure 3.4 The calculation of *edit distance* in Smith-Waterman algorithm.

A score is assigned based on *match/mismatch* for each pair of bases in the *substitution matrix*, where a *positive score* is assigned for matches, whereas a *lower/negative score* is assigned for mismatches. A *gap penalty score* is assigned when a *gap opening* or *extension* is found. The *scoring matrix* records the optimal alignment result by comparing all components one by one, where the *new optimal alignment* is based on the *previous optimal alignment*. In other words, the current alignment is based on deciding which path (match/mismatch or gap) provides the highest score from the previous alignment. The scoring matrix is $1 + L$ of each sequence, where L is the length of the sequence. The extra first row/column allows sequence to be searched. During initialization, both the first row and the first column are set to 0, making the terminal gap free from penalty. The matrix is scored from left to right, top to bottom (Figure 3.4) with the highest score from substitutions, adding gaps. The *traceback* generates the highest similarity score based on the given *scoring system*, which starts at the element with the highest score, and recursively traces back until 0 is encountered.

3.4.2.1 Algorithm

The Smith–Waterman algorithm is described as follows. let us assume that $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$ are the aligning sequences, where n and m represents the lengths of A and B. The matrix $s(a, b)$ is the similarity score of the elements in the two sequences, and the W_k is the penalty of a gap with length k. The scoring matrix H has the size of $(n+1)*(m+1)$, where the first row/column is initialized to 0. The following equation is used to describe the initialization: $H_{k0} = H_{0l} = 0 \quad for \quad 0 \leq k \leq n \quad and \quad 0 \leq l \leq m$. The scoring matrix is filled using the equation described in Algorithm 3.

Algorithm 3 Smith-Waterman algorithm.

$$H_{ij} = max \begin{cases} H_{i-1,j-1} + s(a_i, b_j), \\ max_{k \geq 1}\{H_{i-k,j} - W_k\}, \\ max_{l \geq 1}\{H_{i,j-l} - W_l\}, \\ 0 \end{cases} \qquad (1 \leq i \leq n, 1 \leq j \leq m)$$

In Algorithm 3, $H_{i-1,j-1} + s(a_i, b_j)$ represents the score of aligning a_i and b_j. $H_{i-k,j} - W_k$ represents the score when a_i is at the end of a gap of length k. And $H_{i,j-l} - W_l$ is the score when b_j is at the end of a gap of length l. 0 is assigned when there is no similarity up to a_i and b_j. After successful computation of the scoring matrix, the traceback is used to find the highest similarity score. The traceback starts at the highest score in the scoring matrix H and ends when a score of 0 is met.

3.4.2.2 Substitution Matrix

In the substitution matrix, *matches* are assigned with a positive score, and *mismatches* are assigned lower or negative. Table 3.13 shows an example of the *subsection matrix*, with the assumption of matching score +1, and mismatch score −1. The substitution matrix follows Algorithm 4.

Table 3.13 An example of the substitution matrix.

	A	**G**	**C**	**T**
A	1	-1	-1	-1
C	-1	1	-1	-1
G	-1	-1	1	-1
T	-1	-1	-1	1

Algorithm 4 the algorithm for computing substitution matrix.

$$s(a_i, b_j) = \begin{cases} +1, & a_i = b_j \\ -1, & a_i \neq b_j \end{cases}$$

Algorithm 4 describes a simple algorithm for *substitution matrix* computation. String $A = a_1a_2...a_n$ and string $B = b_1b_2...b_m$ are formed from the alphabet. Table 3.13 is a typical example of *substitution matrix* for DNA. $s(a_i, b_j)$ is the *matching score* for the i's character in string a and j's character in string b. In the example, a match is assigned with a score 1, and a mismatch is assigned with -1. The *substitution matrix* is different when the *Smith-Waterman algorithm* is used for both DNA and protein sequences.

3.4.2.3 Gap Penalty

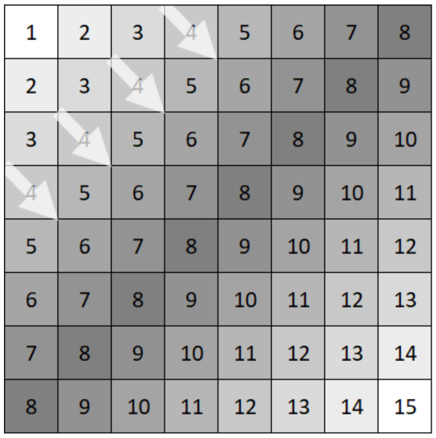
Linear and affine are the two most commonly used gap penalty strategies in the *Smith-Waterman algorithm.* The most straightforward one is the linear gap penalty strategy, where all gaps have the same penalty weight. As the connected gaps formed by a long gap are preferable to multiple short scattered gaps, the concepts of gap opening and gap extension are introduced into the scoring system. The affine gap penalty strategy has a different penalty weight on *gap opening* and *gap extension.* Assuming that the gap penalty function is denoted as W_k where the length of the gap is denoted as k, the linear gap penalty strategy calculates the penalty score as $W_k = kW_1$, and the Smith-Waterman algorithm is simplified as Algorithm 5, where the time complexity is $O(mn)$. Affine gap penalty strategy calculates the penalty score as $W_k = uk + v \quad (u > 0, v > 0)$, where v is the gap opening penalty, and u is the gap extension penalty which has a time complexity of $O(m^2n)$.

Algorithm 5 **Simplified Smith-Waterman algorithm.**

$$H_{ij} = max \begin{cases} H_{i-1,j-1} + s(a_i, b_j), \\ H_{i-1,j} - W_1, \\ H_{i,j-1} - W_1, \\ 0 \end{cases}$$

3.4.2.4 Smith-Waterman Algorithm Parallelization

ksw_ extend2 has a certain similarities to *Smith-Waterman algorithm.* In *parallelization Level 1* of the *Smith-Waterman algorithm*, the parallelization is done within the single alignment. In *Level 2*, the multiple alignments have been aligned at the same time. The common parallelization technique used for smith-Waterman algorithm is called the wave front method, which is parallelism on anti-diagonal.

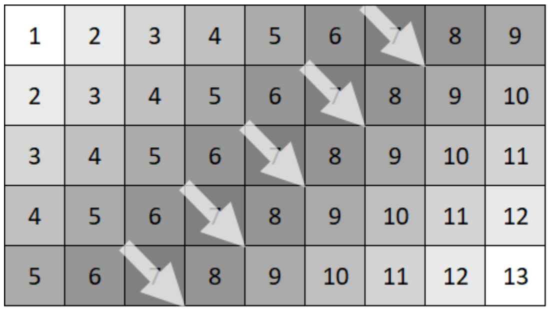


1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14
8	9	10	11	12	13	14	15

Figure 3.5 GPU version of Smith-Waterman algorithm computation with same read and reference sizes. The lengths of the reference and read are the same. The number in each cell represents which step it is being computed.

Figure 3.5 illustrates the wave front method and how it calculates for two strings with the same length of *8 bp*. The total number of steps needed to compute the whole matrix is

15. The number of cells that can be computed in each step is increased by 1 from step 1 to step 8. After step 8, the number of cells will decrease by one until step 15.



1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	8	9	10
3	4	5	6	7	8	9	10	11
4	5	6	7	8	9	10	11	12
5	6	7	8	9	10	11	12	13

Figure 3.6 GPU version of Smith-Waterman algorithm computation with different read and reference sizes. The lengths of the reference and read are different. The number in each cell represents which step it is being computed.

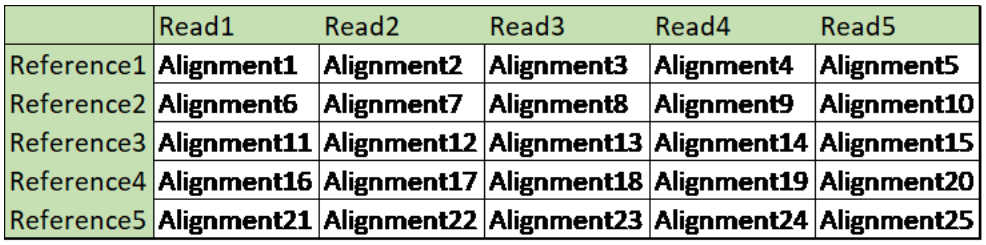
However, not all alignments will have the same length. Figure 3.6 is an example of an alignment having different sizes of strings (where the first one is 5, and the second one is 9). From step *1* to step *4*, the number of cells that are computed together is equal to the step number. From step *5* to step *9*, the number of cells that are computed together is equal to the matrix width. From step *10* to step *13*, the number of cells can be parallelized start to decrease by 1 in each step.

Algorithm 6 Maximum number of cells computed in each step. The number of cells being computed together increases first, and then start to decrease.

$$
\begin{aligned}
L_{total} =& L_0 + L_1 \\
L_s =& Min\{L_0, L_1\} \\
L_l =& max\{L_0, L_1\} \\
for\, 1 \leq i \leq L_s : &\quad C_i = i \\
for\, L_s < i \leq L_l : &\quad C_i = L_s \\
for\, L_l < i \leq L_{total} : &\quad C_i = L_{total} - i
\end{aligned}
$$

Algorithm 6 describes maximum cells computed, where the letter i represents the i^{th} step. The two strings S_0, and S_1's length are denoted as $|S_0| = L_0$, and $|S_1| = L_1$, and the dimension of solving matrix is $L_0 \cdot L_1$. The number of the paralleled cell at step i is labelled as C_i. The maximum number of steps taken is equal to the length of the matrix diagonal, which is the total length of both strings $L_{total} = L_0 + L_1$. The maximum number of cells being computed at the same time is equal to the shortest length of L_0 and L_1, which is labelled as L_s. L_l is equal to the longest length of L_0 and L_1. The number of the paralleled cells starts to decrease at L_l. The number of cells computed at the same time is increased by 1 from step 1 to step L_s. The number of cells computed at the same time remains the same from step L_s to step L_l, and it starts decreasing at step L_l. Therefore, the time complexity of the GPGPU version is $L_0 + L_1$, the CPU version is $L_0 \cdot L_1$, and the parallelized version of *ksw_ extend2* has a time complexity of $L_0 + L_1$.

As the number of threads is limited, the matrix is divided into blocks, where the dimension depends on the GPGPU computation power. Each block or a single solving unit is being solved by one CUDA block, where $B \cdot T$ cells can be calculated at the same time (B is the number of CUDA blocks, and T is the number of CUDA threads). If the best performance is to have an $8 \cdot 8$ cell block (the number of threads is 64 per block), the solving matrix would be divided into cell blocks of a dimension $8 \cdot 8$, which means the number of blocks is $\lceil \frac{L_0}{8} \rceil \cdot \lceil \frac{L_1}{8} \rceil$.



	Read1	Read2	Read3	Read4	Read5
Reference1	**Alignment1**	**Alignment2**	**Alignment3**	**Alignment4**	**Alignment5**
Reference2	**Alignment6**	**Alignment7**	**Alignment8**	**Alignment9**	**Alignment10**
Reference3	**Alignment11**	**Alignment12**	**Alignment13**	**Alignment14**	**Alignment15**
Reference4	**Alignment16**	**Alignment17**	**Alignment18**	**Alignment19**	**Alignment20**
Reference5	**Alignment21**	**Alignment22**	**Alignment23**	**Alignment24**	**Alignment25**

Figure 3.7 Three level of parallelization. *Level 2* parallelization contains multiple alignment pairs, and all pairs are the same length. Once the alignment started, all the alignments are started in the same time in all alignment pairs.

Figure 3.7 shows the relationship between read and reference in a GPGPU version of Smith-Waterman in *Level 2*. As the previous method is mainly used for single alignment, multiple alignments are done using combinations of multiple sequences. For example, there are 4 pairs of sequences and references with the length of n, a matrix with dimension $4n * 4n$ is used while solving.

3.4.3 ksw_extend2

ksw_ extend2 is based on the Smith-Waterman extension algorithm but with a pruning mechanism and a complex scoring system. In the linear gap penalty strategy, all gaps have the same penalty weight. In the affine gap penalty strategy, gap opening and gap extension have a different penalty weight. In *ksw_ extend2*, gaps are classified into gap opening, gap extension, gap insertion, and gap deletion, and each type has a different penalty wight. The gap penalty score system for *ksw_ extend2* is described as follows.o_ins is the gap opening insertion score, which has a default value of 6.
e_ins is the gap extension insertion score, which has a default value of *1.*
o_del is the gap opening deletion score, which has a default value of *6.*
e_del is the gap extension deletion score, which has a default value of *1.*
oe_ins is the sum of the gap opening insertion score and gap extension insertion score.
oe_del is the gap opening deletion score plus gap extension deletion score.
gapo is the gap opening score, which is either *o_ ins* or *o_ del* (with the default value of *6*).
gape is the gap extension score, which is either *e_ ins* or *e_ del* (with the default value of *1*).

Table 3.14 Seed Extension in *ksw_ extend2*. The extensions are performed for both ends of the seed in *ksw_ extend2*.

	Left Extension[1]	Seed[2]	Right Extension[3]	Length[4]
Reference Sequence[5]	TT	ATCCTATTACATTATCAATCCTTGC	ATTTCAGCTTCTT	40
Seed[6]		ATCCTATTACATTATCAATCCTTGC		25
Queue[7]	G	ATCCTATTACATTATCAATCCTTGC	GTTTCAGCT	34
Left Reference Sequence Alignment Fragment[8]	TT			2
Left Queue Sequence Alignment Fragment[9]	G			1
Right Reference Sequence Alignment Fragment[10]			ATTTCAGCTTCTT	13
Right Queue Sequence Alignment Fragment[11]			GTTTCAGCT	9

[1]Sequence fragments for left extension.

[2]Exact matched region from both seed and reference simulated data.

[3]Sequence fragments for right extension.

[4]The length of sequences, unit is in *bp*.

[5]The part of the reference simulated data.

[6]The exact matched sequence from read.

[7]Queue, sequence fragment pulled from read.

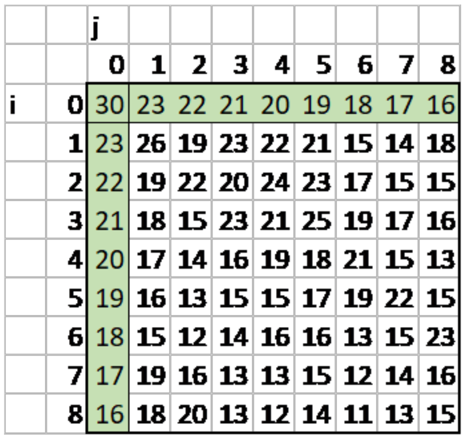
[8]Sequence fragment from reference for left extension.

[9]Sequence fragment from queue for left extension.

[10]Sequence fragment from reference for right extension.

[11]Sequence fragment from queue for right extension.

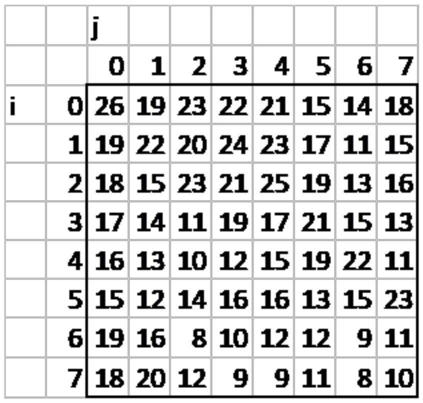
Table 3.14 shows how seed extension is done in *BWA-MEM*. Seed is obtained from previous *exact match*. Then the extensions are performed on both ends of the seed with the *ksw_ extend2*. The exact match score is obtained using the length of the seed.



		j								
		0	1	2	3	4	5	6	7	8
i	0	30	23	22	21	20	19	18	17	16
	1	23	26	19	23	22	21	15	14	18
	2	22	19	22	20	24	23	17	15	15
	3	21	18	15	23	21	25	19	17	16
	4	20	17	14	16	19	18	21	15	13
	5	19	16	13	15	15	17	19	22	15
	6	18	15	12	14	16	16	13	15	23
	7	17	19	16	13	13	15	12	14	16
	8	16	18	20	13	12	14	11	13	15

Figure 3.8 ***ksw_ extend2*** **algorithm** H **matrix**[1]**.** The numbers highlighted in green are initialized based on previous exact matches.

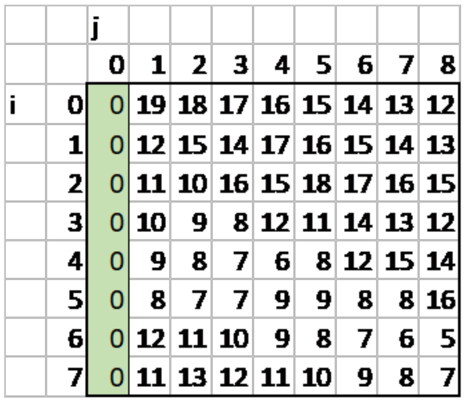
[1]The result is based on aligning sequence fragment *AACCCTTC* and sequence fragment *CCCGTCAA*.



		j							
		0	1	2	3	4	5	6	7
i	0	26	19	23	22	21	15	14	18
	1	19	22	20	24	23	17	11	15
	2	18	15	23	21	25	19	13	16
	3	17	14	11	19	17	21	15	13
	4	16	13	10	12	15	19	22	11
	5	15	12	14	16	16	13	15	23
	6	19	16	8	10	12	12	9	11
	7	18	20	12	9	9	11	8	10

Figure 3.9 ***ksw_ extend2*** **algorithm** M **matrix**[1]**.**

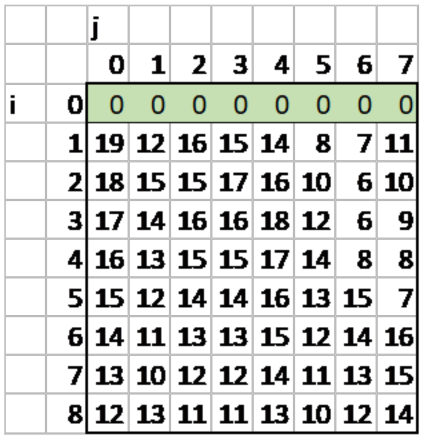
[1]The result is based on aligning sequence fragment *AACCCTTC* and sequence fragment *CCCGTCAA*.



		j								
		0	1	2	3	4	5	6	7	8
i	0	0	19	18	17	16	15	14	13	12
	1	0	12	15	14	17	16	15	14	13
	2	0	11	10	16	15	18	17	16	15
	3	0	10	9	8	12	11	14	13	12
	4	0	9	8	7	6	8	12	15	14
	5	0	8	7	7	9	9	8	8	16
	6	0	12	11	10	9	8	7	6	5
	7	0	11	13	12	11	10	9	8	7

Figure 3.10 *ksw_ extend2* algorithm E matrix[1]. The numbers highlighted in green are initialized to 0.

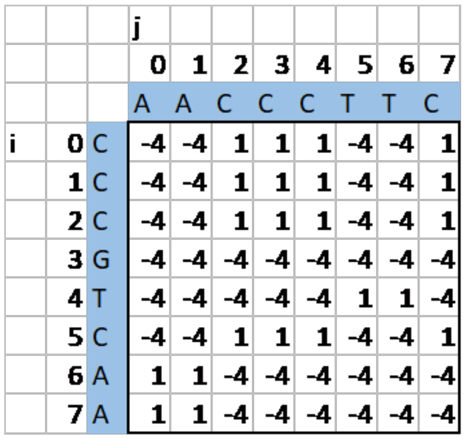
[1]The result is based on aligning sequence fragment *AACCCTTC* and sequence fragment *CCCGTCAA*.



		j							
		0	1	2	3	4	5	6	7
i	0	0	0	0	0	0	0	0	0
	1	19	12	16	15	14	8	7	11
	2	18	15	15	17	16	10	6	10
	3	17	14	16	16	18	12	6	9
	4	16	13	15	15	17	14	8	8
	5	15	12	14	14	16	13	15	7
	6	14	11	13	13	15	12	14	16
	7	13	10	12	12	14	11	13	15
	8	12	13	11	11	13	10	12	14

Figure 3.11 *ksw_ extend2* algorithm F matrix[1]. The numbers highlighted in green are initialized to 0.

[1]The result is based on aligning sequence fragment *AACCCTTC* and sequence fragment *CCCGTCAA*.



			j							
			0	1	2	3	4	5	6	7
			A	A	C	C	C	T	T	C
i	0	C	-4	-4	1	1	1	-4	-4	1
	1	C	-4	-4	1	1	1	-4	-4	1
	2	C	-4	-4	1	1	1	-4	-4	1
	3	G	-4	-4	-4	-4	-4	-4	-4	-4
	4	T	-4	-4	-4	-4	-4	1	1	-4
	5	C	-4	-4	1	1	1	-4	-4	1
	6	A	1	1	-4	-4	-4	-4	-4	-4
	7	A	1	1	-4	-4	-4	-4	-4	-4

Figure 3.12 ***ksw_ extend2*** **algorithm** S **matrix**[1]**.** The sequences highlighted on blue are the sequence fragments used for alignment.

[1]The result is based on aligning sequence fragment *AACCCTTC* and sequence fragment *CCCGT-CAA*.

As the *ksw_ extend2*'s gap penalty strategy considers *gap opening, extension, insertion and deletion* separately, the three matrices, H, E, and F are used to represent final, gap insertion, and gap deletion scores. Let us take two sequences *AACCCTTC*, *CCCGTCAA*, and an exact matching score of 30 as an example, the result of H, E, and F are shown in Figure 3.8, Figure 3.10, and Figure 3.11 respectively. Substitution matrix S (Figure 3.12) represents the match/mismatch score, and the default value of a match is 1, and a mismatch is -4. M Matrix represents the *alignment matching score* (Figure 3.9).

Algorithm 7 *ksw_ extend2* matrices initialization. The algorithm initializes the numbers highlighted in green in H, M, E, F, and S matrices.

$$\begin{aligned}
E_{i,0} &= F_{0,j} = 0 \quad for \quad 0 \leq i \leq n+1 \quad and \quad 0 \leq j \leq m+1 \\
H_{0,0} &= h_0 \\
H_{1,0} &= Max\{H_{0,0} - oe\_ins, 0\} \\
H_{0,1} &= Max\{H_{0,0} - oe\_del, 0\} \\
H_{i,0} &= Max\{H_{i-1,0} - e\_ins, 0\} \quad for \quad 2 \leq i \leq n+1 \\
H_{0,j} &= Max\{H_{0,j-1} - e\_del, 0\} \quad for \quad 2 \leq i \leq m+1
\end{aligned}$$

Algorithm 7 initializes each matrix before the computation. $E_{i,0} = F_{0,j} = 0\, for \quad 0 \leq i \leq n+1 \quad and \quad 0 \leq j \leq m+1$, initializes the first column of E matrix and first row of F matrix to 0. As the extension is based on the previous exact matches, $H_{0,0} = h_0$ sets cell $(0,0)$ in the H matrix to h_0. Let us assume that the length of exact match is k, h_0 is equal to k multiplied by a predefined score a. $H_{1,0} = Max\{H_{0,0} - oe\_ins, 0\}$ and $H_{0,1} = Max\{H_{0,0} - oe\_del, 0\}$ sets the second value to the first value minus gap penalty score (oe_ins for the first one, and oe_del for the second one) in the first column or row of the H matrix when it is larger than 0 (otherwise it is set to 0). The last two columns compute the rest of the first row or column by deducting the gap extension score from the previous one. The part that is being initialized is highlighted with green in Figure 3.8-3.11.

Algorithm 8 *ksw_ extend2* matrices value computation. The algorithm computes the rest of the numbers in H, M, E, F, and S matrices.

$$\begin{aligned}
H_{i,j} &= Max\left\{M_{i-1,j-1}, E_{i-1,j-1}, F_{i-1,j-1}\right\} \quad for \quad 1 \leq i \leq n+1 \quad and \quad 1 \leq j \leq m+1 \\
M_{i,j} &= Max\left\{H_{i,j} + S_{i,j}, 0\right\} \quad for \quad 0 \leq i \leq n+1 \quad and \quad 1 \leq j \leq m+1 \\
E_{i,j+1} &= Max\left\{M_{i,j} - oe\_ins, E_{i,j} - e\_ins\right\} \quad for \quad 1 \leq i \leq n+1 \quad and \quad 0 \leq j \leq m \\
F_{i+1,j} &= Max\left\{M_{i,j} - oe\_del, F_{i,j} - e\_del\right\} \quad for \quad 1 \leq i \leq n \quad and \quad 0 \leq j \leq m+1
\end{aligned}$$

After successful initialization, the rest of the cells in H, M, E, and F matrices are being filled using Algorithm 8. The first row in the algorithm fills the $H_{i,j}$ cell from the maximum value of $M_{i-1,j-1}$, $E_{i-1,j-1}$ and $F_{i-1,j-1}$. The $M_{i,j}$ cell is filled with the max value of $H_{i,j}+S_{i,j}$ and 0. The $E_{i,j+1}$ cell is equal to the maximum value of $M_{i,j} - oe\_ins$, and $E_{i,j} - e\_ins$. The $F_{i+1,j}$ is the maximum value of $M_{i,j} - oe\_del$, and $F_{i,j} - e\_del$. Cell $M_{i,j}$ is the score obtained comparing the character i and character j. $E_{i,j+1}$ cell is the maximum score from a new gap insertion ($M_{i,j} - oe\_ins$) or a gap insertion extension ($E_{i,j} - e\_ins$). $F_{i+1,j}$ cell is the maximum score from a new gap deletion ($M_{i,j} - oe\_del$) or a gap deletion extension ($F_{i,j} - e\_del$). Figure 3.13 describes how the cells are computed using Algorithm 8.

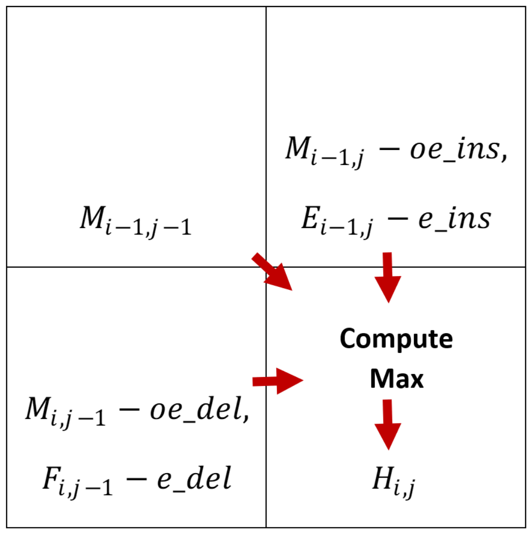


Figure 3.13 *ksw_ extend2*'s cell computation. The value of $H_{i,j}$ is depends on its surrounding cells.

3.4.3.1 Pruning Optimization

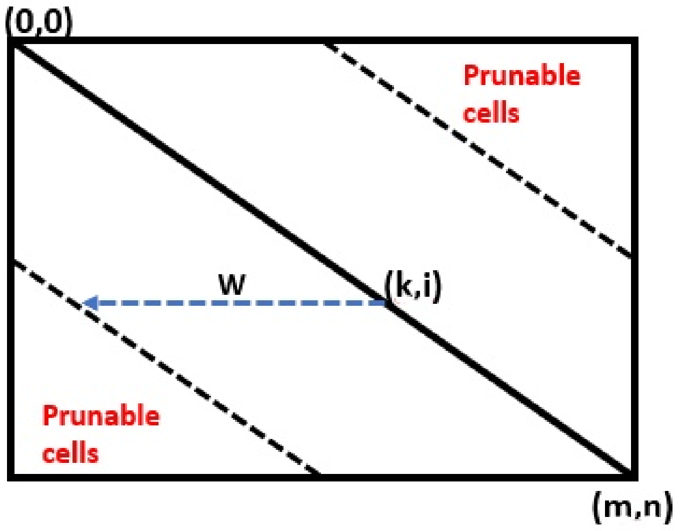


Figure 3.14 Prunable cells in H matrix. The cells labeled with *Prunable cells* are the prunable cells that can be ignored during the computation. The mechanism controlling the computing area is called the *pruning mechanism.*

Pruning optimization eliminates cells that are mathematically impossible to produce a higher score than the current maximum score. Therefore, for each row in the matrix, the calculation is only performed between *beg* and *end* (Figure 3.14), where the *beg* and *end* are computed as Algorithm 9. Because of the pruning mechanism, *ksw_ extend2*'s time complexity has been reduced. Table 3.15 shows the time complexity of *ksw_ extend2* and GPGPU version of *ksw_ extend2* under the case with pruning or without the pruning mechanism.

Algorithm 9 *ksw_ extend2*'s pruning mechanism algorithm. This algorithm determines the area for computation.

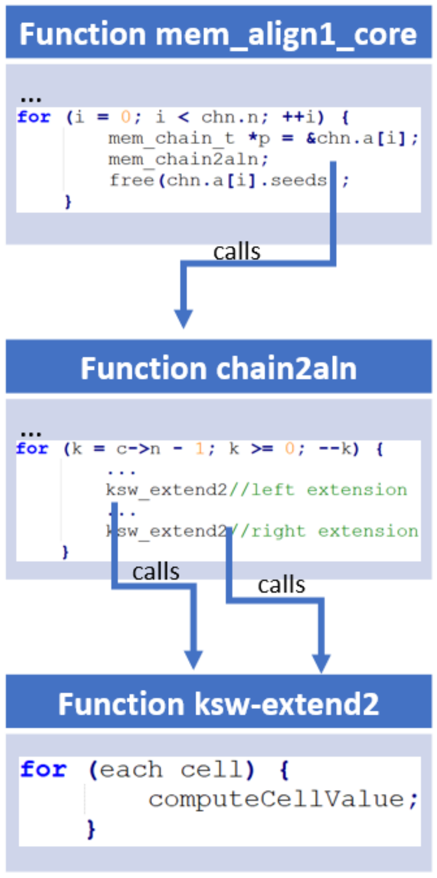
$$
\begin{aligned}
w =& Max\{max_{insertion\,score}, max_{deletion\,score}\}\\
max_{insertion\,score} =& \frac{Length_{read} \cdot score_{match} + score_{end\,Bonus} - score_{open\,insertion}}{score_{extension\,insertion} + 1}\\
max_{deletion\,score} =& \frac{Length_{read} \cdot score_{match} + score_{end\,Bonus} - score_{open\,deletion}}{score_{extension\,deletion} + 1}\\
for\,1 \leq i \leq M :&\\
beg =& i - w\\
end =& i + w + 1
\end{aligned}
$$

Table 3.15 Time Complexity of GPGPU and CPU version. With the *pruning mechanism*, the time complexity for CPU version of *ksw_ extend2* changed from quadratic time to linear time. However, GPGPU version remains the same with or without the *pruning mechanism*.

	ksw_extend2	GPU *ksw_ extend2*
With The Pruning Mechanism	Linear	$L_1 + L_2$
Without The Pruning Mechanism	$L_1 \cdot L_2$	$L_1 + L_2$

3.4.4 Program Design ksw_extend2 Parallelization

Based on the functionality of *ksw_ extend2*, the parallelization is divided into three levels: *seed extension* (seed-extend2), *chain extension* (chain2aln), and *alignments extension* (mem_align1_core). Figure 3.15 shows the simplified structure of the *BWA-MEM* software, where *ksw_extend2* is for *single-sided seed extension*, *chain2aln* deals with *chain level extension*, and part of function mem_align1_core performs extension on all alignments.



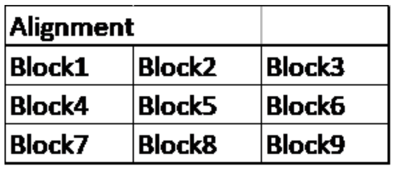
```
...
for (i = 0; i < chn.n; ++i) {
        mem_chain_t *p = &chn.a[i];
        mem_chain2aln;
        free(chn.a[i].seeds);
    }
```

```
...
for (k = c->n - 1; k >= 0; --k) {
        ...
        ksw_extend2//left extension
        ...
        ksw_extend2//right extension
    }
```

```
for (each cell) {
        computeCellValue;
    }
```

Figure 3.15 Three level of parallelizations. The *Level 1* of parallelization is function *ksw_extend2*, which handles the *single side seed extension*. The *Level 2* of parallelization is function *chain2aln*, which calls function *ksw_extend2* and performs seed extension on both ends of all seeds in the same chain. The *Level 3* of parallelization is performed on multiple chains.

3.4.4.1 *Level 1* Parallelization: Seed Extension



Alignment		
Block1	Block2	Block3
Block4	Block5	Block6
Block7	Block8	Block9

Figure 3.16 Blocks in *Level 1* parallelization. When the size of the sequence fragments are really long, they will be separated into small computation blocks.

The parallelization technique for *Level 1* parallelization is similar to the *Smith-Waterman algorithm*'s parallelized version. Similar to the GPGPU version of the *Smith-Waterman algorithm*, cells are separated into blocks (Figure 3.16). The pruning mechanism does not reduce the time complexity as it does not reduce the number of steps taken[9]. The time consumption may increase as the *beg*, and *end* need to be computed for each line.

A GPGPU version of the *Smith-Waterman algorithm*, PasWAS [77] has been redesigned to fit the needs. *PasWAS* is a simple GPGPU version of the Smith-Waterman algorithm with the *linear gap penalty strategy.* Just like a *typical Smith-Waterman algorithm*, the *wave front technique* is applied towards *ksw_ extend2*'s parallelization.

3.4.4.2 *Level 2* and *Level 3* Parallelization: Chaining and Chain Filtering

In the step of *chaining and chain filtering*, seeds that are close to each other are grouped into chains, and the seeds that are not going to have successful extensions are filtered out. Furthermore, function *ksw_ extend2* is called by function *chain2aln*, which performs seed extension.

In the Smith-Waterman algorithm, all combinations are tested for finding matches. However, in *ksw_ extend2*'s version, each reference has only one read to be aligned. After the redesign, instead of processing alignments in a *wave front* manner (like the Smith-Waterman

[9]The number of steps taken is equal to the length of the diagonal.

algorithm), and all alignments are processed simultaneously.

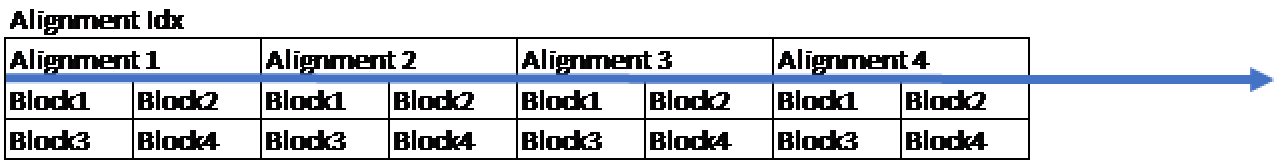

Alignment Idx
Alignment 1 | Alignment 2 | Alignment 3 | Alignment 4
Block1 | Block2 | Block1 | Block2 | Block1 | Block2 | Block1 | Block2
Block3 | Block4 | Block3 | Block4 | Block3 | Block4 | Block3 | Block4

Figure 3.17 GPU *Level 2* parallelization. At parallelization *Level 2*, multiple extensions from the same chain are performed at the same time.

Figure 3.17 shows an example of four alignments being aligned in a single GPGPU run, where each alignment contains four computation blocks. Each of the blocks represents a single solving unit, which is solved by one CUDA block. All alignments are in a single row as they are from the same chain. As the computation proceeds, in step 1, all the *block*1 blocks are computed simultaneously. All the *block*2 and *block*3 blocks are computed at the same time in step 2. After completing step 2, the *block*4 are computed simultaneously in step 3.

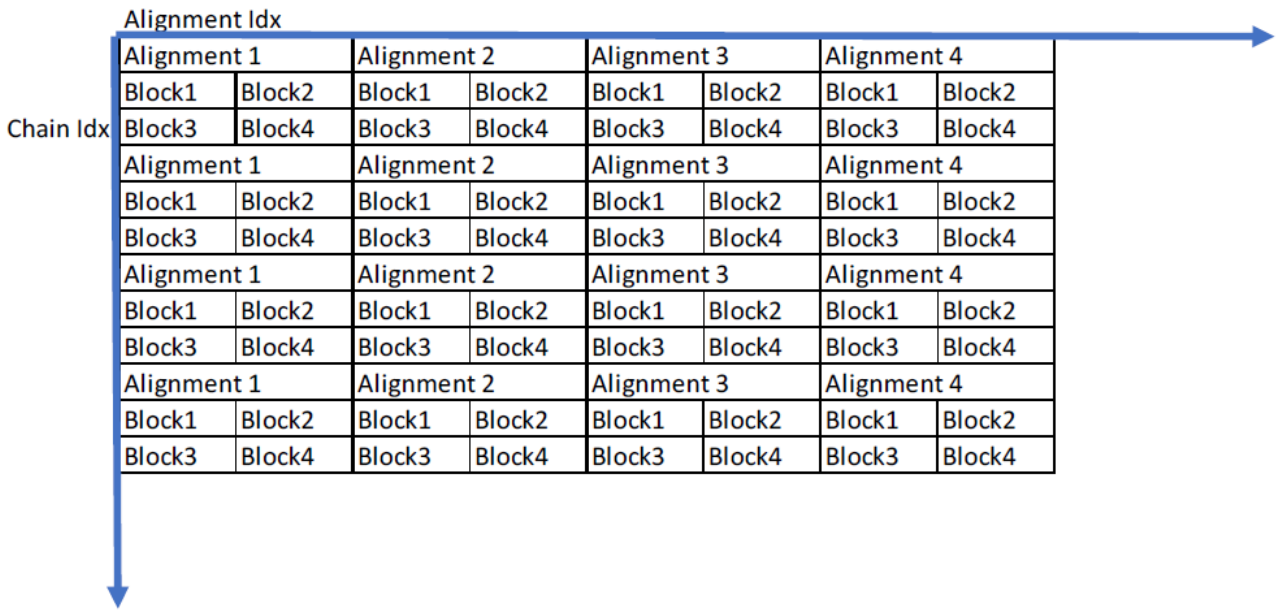

Alignment Idx
Chain Idx
Alignment 1 | Alignment 2 | Alignment 3 | Alignment 4
Block1 | Block2 | Block1 | Block2 | Block1 | Block2 | Block1 | Block2
Block3 | Block4 | Block3 | Block4 | Block3 | Block4 | Block3 | Block4
Alignment 1 | Alignment 2 | Alignment 3 | Alignment 4
Block1 | Block2 | Block1 | Block2 | Block1 | Block2 | Block1 | Block2
Block3 | Block4 | Block3 | Block4 | Block3 | Block4 | Block3 | Block4
Alignment 1 | Alignment 2 | Alignment 3 | Alignment 4
Block1 | Block2 | Block1 | Block2 | Block1 | Block2 | Block1 | Block2
Block3 | Block4 | Block3 | Block4 | Block3 | Block4 | Block3 | Block4
Alignment 1 | Alignment 2 | Alignment 3 | Alignment 4
Block1 | Block2 | Block1 | Block2 | Block1 | Block2 | Block1 | Block2
Block3 | Block4 | Block3 | Block4 | Block3 | Block4 | Block3 | Block4

Figure 3.18 GPU *Level 3* parallelization. At parallelization *Level 3*, multiple chain extensions are performed at the same time.

Level 2 parallelization assumes each row as a *single chain*, a matrix represents all the chains for computation (as shown in Figure 3.18). The number of steps taken to solve all chains in the matrix is the same as in *Level 2* parallelization. After the experiment on standard data, on average, only three extensions are being computed in each chain at *Level 2 parallelization*. Therefore, for both *Level 2* and *Level 3* parallelization, all alignments that have been performed are collected. Because the length of the extension is usually small, what matters now is the length of alignment lower than 32 and the number of alignments.

The process can be summarized into three steps. First, all alignments are collected and stored in arrays. Multiple alignments are then carried out at the same time. In the last step, the program finishes the final calculation. In *Level 2* parallelization, only one cell is computed in step 1. The number of cells calculated is increased by how many steps are taken before the step $widthofalignment$. However, in the *Level 3* parallelization, assuming that k alignments are being performed simultaneously, each step's computed cell is multiplied by 100 times.

Two different versions of GPGPU multi-threaded *ksw_ extend2* are created, where one is targeted at reducing memory usage, and another one is focusing on saving time. The *GPU with time-saving version* has its name ending with *_ v2*. The purpose of having two different versions is to find out the effect of data transferring cost.

```
CalculateScoreHost(){
  for (unsigned int i=1; i < XdivSHARED_X+YdivSHARED_Y; ++i) {
    //compute trhe max number of blocks.
    if (i <= maxNumberOfBlocks)
      numberOfBlocks = i;
    else if( i >= startDecreaseAt) numberOfBlocks = XdivSHARED_X+
   YdivSHARED_Y - i;
    else numberOfBlocks = maxNumberOfBlocks;
    dim3 dimGridSW(NUMBER_SEQUENCES,NUMBER_TARGETS*numberOfBlocks , 1);
    calculateScore<<<dimGridSW, dimBlock>>>();
    cudaThreadSynchronize();
    //increase y's starting position when x reaches the left corner
    if (x == XdivSHARED_X - 1)
      ++y;
    //increase x's starting position before x reaches the left corner
    if (x < XdivSHARED_X - 1)
      ++x;
  }
}
__global__ void calculateScore(){
  //initialize the shared matrix
  //calculate the blockx, blocky, tIDx, tIDy, bIDx, bIDy
  //initialize the surrounding
  for (int i=0; i < DIAGONAL; ++i) {
    //compute matrix score
  }
  //copy the result to the global memory
}
```

Listing 3.1 A simplified GPGPU Smith-Waterman algorithm framework from PasWAS

Listing 3.1 is the basic framework from *PaSWAS*. A few different versions of *Smith-Waterman algorithm* implementation have been looked into, but PasWAS [77] was chosen for its simple structure and because it is intuitive.

3.4.5 ksw_extend2 with Time-saving Version Implementation

ksw_extend2 consists of three separate phases: (1) alignment collection and preparation, (2) calculation of alignment scores, and (3) production of profiles as the output of results. For the explanation of the application, the following set-up is used. On the horizontal axis, there are x number of sequences, each at a length of N. These are part of the reads from a sequencing platform that needs an extension. If a sequence is shorter than N, it is padded to length N with a unique character. All sequences are placed in a single string x with length $X * N$. On the vertical axis, the target sequences are placed. There are Y target sequences, each with length M. These sequences are part of the reference that needs an extension, and they are padded when shorter than M. They are placed in a single string Y of length $Y * M$. In the first phase, all align collected, and they are prepared in an array of structures (struct sw_ext). The strings x and y (for all alignments) are combined and copied to the main (global) memory of the GPU.

In the second phase, all sequence alignments are calculated in parallel. Cells are anti-diagonally updated from the upper left to the lower right of each alignment. Let us assume that the number of alignments is K. At the beginning, there will be K threads active, and at peak performance, there are $K * minimum(N, M)$ threads active. During the entire phase, the maximum value of each row and its position is tracked.

The last phase runs on the host, which produces the alignment profiles. The output contains additional information about each profile, including the number of gaps, mismatches, and the start and end of the alignments. The required outputs (such as best global alignment in the read, the best global alignment in the reference, full alignment score, and maximum off-diagonal distance) are calculated and stored in the array of structures (struct sw_ext).

3.4.5.1 Phase 1: Alignment Collection and Preparation

In Phase 1, alignments are collected from all chains. Once the maximum amount of memory is reached, collecting them into the queue is stopped. As the algorithm needs to modify each alignment's parameters multiple times, they were stored as pointers to be picked up in later computation. There will be processed and prepared in gpu_sw_seed_extend. All the parameters and sequences are stored in arrays.

```
int *d_col = 0, *d_row = 0;
uint8_t *d_sequences = 0, *d_references = 0;
*(h_shared +oe_del) = *(h_shared +o_del) + *(h_shared +e_del);
*(h_shared +oe_ins) = *(h_shared +o_ins) + *(h_shared +e_ins);
//[1]search for max_qlen and max_tlen for all element
//we are looking for the max qlen and max tlen
*(h_shared +max_qlen) = swext->qlen;
*(h_shared +max_tlen) = swext->tlen;
//starts from 1
for(int i = 1; i < *(h_shared + dims3); i++){//skip the first one
  if((swext+i)->qlen > *(h_shared +max_qlen)) *(h_shared +max_qlen) = (
   swext+i)->qlen;
  if((swext+i)->tlen > *(h_shared +max_tlen)) *(h_shared +max_tlen) = (
   swext+i)->tlen;

}
*(h_shared +block_x_len) = (int) ceil((double)*(h_shared +max_qlen)/
   SHARED_X);//how many 8*8 block on x div
*(h_shared +block_y_len) = (int) ceil((double)*(h_shared +max_tlen)/
   SHARED_Y);//how many 8*8 block on y div
*(h_shared +max_x) = *(h_shared +block_x_len) * SHARED_X;//for seed
   extension part
*(h_shared +max_y) = *(h_shared +block_y_len) * SHARED_Y;//reference
   extension part
*(h_shared +alignment_x) = *(h_shared +max_x) * *(h_shared +dims3);//TOTAL
    LENGTH OF X
*(h_shared +alignment_y) = *(h_shared +max_y) * *(h_shared +dims3);//TOTAL
    LENGTH OF Y
*(h_shared +block_diagnal_len) = max(*(h_shared +block_x_len), *(h_shared
   +block_y_len));//TOTAL LENGTH OF Y
*(h_shared +alignment_diagnal_len) = *(h_shared +block_diagnal_len)**(
   h_shared +dims3);//TOTAL LENGTH OF Y
int h_row[*(h_shared +alignment_x)];
int h_col[*(h_shared +alignment_y)];
uint8_t h_seq[*( h_shared +alignment_x)];
```

```
uint8_t h_ref[*( h_shared +alignment_y)];
LocalMatrix t_sc[*(h_shared +dims4)][*(h_shared +dims3)][*(h_shared +block_y_len)][*(h_shared +block_x_len)];
//LocalMatrix *h_scoringMatrix = (LocalMatrix*) calloc(sizeof(LocalMatrix), *(h_shared +dims4) *  *(h_shared +dims3) * *(h_shared +block_y_len) * *(h_shared +block_x_len));
LocalMatrix *h_scoringMatrix = &t_sc[0][0][0][0];
LocalMatrix *d_scoringMatrix;
```

Listing 3.2 Phase 1 - Stage 1: alignment parameter initialization

Listing 3.2 shows the code used for the first stage, which initialises alignment parameters. In the first stage, the necessary parameters from the alignment (such as alignment x length, alignment y length, padded sequence, and reference string length) are initialized, where the matrix $h\_scoringMatrix$ stores all the computed matrices after computation.

```
//we are initializing for everyone
//init h_row, h_col
for(int align_idx = 0; align_idx < *(h_shared +dims3); align_idx++){//go though each alignment option here
  int start_x_pos = align_idx * *( h_shared +max_x);//ours 403//starting position
  int start_y_pos = align_idx * *( h_shared +max_y);//current alignment
  struct sw_ext *curr_sw_ext = swext + align_idx;
  int curr_h0 = curr_sw_ext->sc0;
  h_row[start_x_pos+0] = curr_h0;//position is 0
  h_row[start_x_pos+1] = LIKELY(curr_h0 > *(h_shared +oe_ins))? curr_h0 - *(h_shared +oe_ins) : 0;//404, position as 1
  for(int curr_x_loc = start_x_pos + 2; LIKELY(curr_x_loc < start_x_pos + *(h_shared +max_x)); ++curr_x_loc)
    h_row[curr_x_loc] = (curr_x_loc <= (start_x_pos + curr_sw_ext->qlen) && h_row[curr_x_loc - 1] > *(h_shared +e_ins))? h_row[curr_x_loc - 1] - *(h_shared +e_ins) : 0;
  // adjust $w if it is too large
  // generate the first row
  h_col[start_y_pos+0] = curr_h0; //eh[0].e = highest possible score
  h_col[start_y_pos + 1] = LIKELY(curr_h0 > *(h_shared +oe_del)) ? curr_h0 - *(h_shared +oe_del) : 0;
  for (int curr_y_loc =start_y_pos + 2; LIKELY(curr_y_loc < start_y_pos + *(h_shared +max_y)); ++curr_y_loc)
    h_col[curr_y_loc] = (curr_y_loc <= (start_y_pos + curr_sw_ext->tlen) && h_col[curr_y_loc - 1] > *(h_shared +e_del))? h_col[curr_y_loc - 1] - *(h_shared +e_del) : 0;
  for(int curr_x_loc = start_x_pos; LIKELY(curr_x_loc < start_x_pos + *(h_shared +max_x)); ++curr_x_loc)
    h_seq[curr_x_loc] = LIKELY(curr_x_loc < start_x_pos + curr_sw_ext->qlen)? curr_sw_ext->query[curr_x_loc-start_x_pos] : FILL_CHARACTER;

  for(int curr_y_loc = start_y_pos; LIKELY(curr_y_loc < start_y_pos + *(h_shared +max_y)); ++curr_y_loc)
    h_ref[curr_y_loc] = LIKELY(curr_y_loc < start_y_pos + curr_sw_ext->tlen)? curr_sw_ext->target[curr_y_loc-start_y_pos] : FILL_CHARACTER;
  curr_sw_ext->h_col = &h_col[start_y_pos];
}
```

Listing 3.3 Phase 1 - Stage 2: sequence combination

Listing 3.3 is the code for combining sequences[10] and for initializing matrices. In Stage 2, each alignment's sequence and reference were copied into two padded strings, and *h_col* and *h_row* is computed, which represent the first row and first column of the *h* matrix.

```
checkCudaErrors(cudaMalloc((void**)&d_scoringMatrix, sizeof(t_sc)));
checkCudaErrors(cudaMalloc((void **)&d_row, sizeof(h_row)));
checkCudaErrors(cudaMalloc((void **)&d_col, sizeof(h_col)));
checkCudaErrors(cudaMemcpy(d_row, &h_row[0], sizeof(h_row),
    cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(d_col, &h_col[0], sizeof(h_col),
    cudaMemcpyHostToDevice));
checkCudaErrors(cudaMalloc((void** ) &d_sequences,sizeof(h_seq)));
checkCudaErrors(cudaMalloc((void** ) &d_references,sizeof(h_ref)));
checkCudaErrors(cudaMemcpy(d_sequences, &h_seq[0], sizeof(h_seq),
    cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(d_references, &h_ref[0], sizeof(h_ref),
    cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpyToSymbol(d_mat, mat, 25 * sizeof(int8_t),0,
    cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpyToSymbol(d_shared, h_shared, 21 * sizeof(
    unsigned int),0, cudaMemcpyHostToDevice));
}
```

Listing 3.4 Phase 1 - Stage 3: data transfering

Listing 3.4 shows how data transfer (from host to device) was done. In Stage 3, the computed matrices are transferred from the host to the device. In Phase 1, all three stages are done in function gpu_sw_seed_extend. After Phase 1, all sequence alignment data and pre-calculated values are transferred into matrices and ready to be aligned, and phase 2 starts the GPGPU computation on the matrices.

3.4.5.2 Phase 2: Calculation of Alignment Scores

The GPU's two major memories, global memory, and shared memory, are used to store different data types as they have different speed. The global memory is the largest memory located on the GPGPU device, with several hundred megabytes up to 32 gigabytes in size. The shared memory is relatively small and located close to GPGPU processors, which is 100 times faster than the global memory. The global memory stores computed scoring matrices, parameters and strings from Phase 1. Because global memory has slower memory access, the use of global memory is minimized, and the use of faster shared memory is maximized.

[10]Combining all read and reference sequences (real data) into two string.

Constant memory type, such as texture memory, is slightly faster than the global shared memory used to store frequently used read-only data. Therefore, the intermediate computed values are stored in the shared memory, and final results are stored and accessed from global memory.

Each sequence alignment is divided into blocks with the same dimension of $SHARED\_X \cdot SHARED\_Y$, which is predefined in the smaithwaterman.h. The values of $SHARED\_X$ and $SHARED\_Y$ is calculated using the occupancy calculator provided by NVIDIA (www.nvidia.com), which provides the most efficient setting to optimize the current hardware. This $SHARED\_X \cdot SHARED\_Y$ cell matrices are mapped to thread blocks of $SHARED\_X \cdot SHARED\_Y$ threads. The SHARED_(X or Y, usually X and Y have the same value) characters of the two sequences and other settings are stored using shared memory. Without neighbouring block's scores, it is impossible to compute of the border cells' score that are surrounded by others[11].

h_col and h_row aeries are used to initialize the first column and the first row of the h matrices. *h_col* and *h_row* store pre-calculated scores based on the maximum exact matching score, and the gap insertion scores[12], gap deletion scores[13]. In each block, the calculations are done anti-diagonally. To make use of the idle threads in each block, each row's maximum values and position are determined. Upon completion, the resulting information is transferred to the global memory.

Like the cells within the matrix, each block depends on the three surrounding blocks (except the border matrices). At the start, K blocks of

$$SHARED\_X \cdot SHARED\_Y$$

[11]Except for the border matrices (for example, the matrices close to the y-axis initialize the first columns to the computed scores).

[12]Opening and extension.

[13]Opening and extension.

threads are launched. The maximum number of thread blocks is

$$y \cdot \frac{min(N, M)}{SHARED\_X \cdot SHARED\_Y}$$

For example, if there are 2000 ($k = 2000$) alignments, and each alignment has a sequence length of 32 bp, there will be 2000 blocks launched at the start, and $2000 \cdot \frac{min(N, M)}{SHARED\_X \cdot SHARED\_Y}$ blocks at maximal[14], with each block, containing 64 threads. After the computation is done, the matrix is transferred from the global shared memory to the host memory.

```
for (unsigned int i = 1; LIKELY(i < halfZhouChang); ++i) {
  numBlocks = i <= xiaoBian? i : i >= daBian? halfZhouChang - i : xiaoBian
    ;
  //reserve dim4 for anything beyound chain
  dim3 dimSWGrid(*(h_shared +dims4), *(h_shared +dims3) * numBlocks, 1);//
    numBlocks
  calculateScore_v2<<<dimSWGrid, dimBlock>>>(
      d_scoringMatrix,
      d_row, d_col,
      x, y,
      numBlocks,
      d_sequences, d_references);
  cudaDeviceSynchronize();

  if (x == *(h_shared +block_x_len) - 1)
    ++y;
  if (x < *(h_shared +block_x_len) - 1)
    ++x;
}
```

Listing 3.5 Phase 2: framework from PaSWAS for computing matrices anti-diagonally on CPU side

Listing 3.5 describes the basic framework from PaSWAS, which computes anti-diagonally in the blocks. All the blocks that are located on the current diagonal would be computed when the calculateScore_v2 is finished. The primary function, *calculateScore_v2*, is separated into five steps. In the first step, the shared matrices are initialized to zero, and the blockx, blocky, tIDx, tIDy, bIDx, and bIDy are calculated for the exact location in the matrix. In the second step, the computation matrices are initialized. In the third step, a *for* loop is used to compute the scoring matrix diagonal by diagonal. In the last step, the computed result is copied to the global memory.

```
__shared__ int h_matrix[SHARED_Y+1][SHARED_X+1];
__shared__ int e_matrix[SHARED_Y+1][SHARED_X+1];
```

[14]When all alignments are computing the middle diagonals

```
__shared__ int f_matrix[SHARED_Y+1][SHARED_X+1];
__shared__ int s_maxima[SHARED_Y];
__shared__ int x_maxloc[SHARED_Y];

memset(&h_matrix[0][0], EPT_SCORE, tmp_1*sizeof(int));
memset(&e_matrix[0][0], EPT_SCORE, tmp_1*sizeof(int));
memset(&f_matrix[0][0], EPT_SCORE, tmp_1*sizeof(int));
memset(&s_maxima[0], EPT_SCORE, SHARED_Y*sizeof(int));
memset(&x_maxloc[0], EPT_SCORE, SHARED_Y*sizeof(int));
```

Listing 3.6 Phase 2 stage 1: matrices initialization

Listing 3.6 is the code for initialization in matrices, and they are stored in shared memory for faster access. h_matrix, e_matrix and f_matrix store the calculated h, e and h values. s_maxima and the x_mxloc store the maximum values and maximum locations of current row. As CUDA does not automatically reset values in the matrix back to zero, *memset* is used to reset these values.

```
//if there are only one item, block y is the number of block we needed
//x, y is our axises
unsigned int currAlign = blockIdx.y%d_shared[dims3];//which alignment we
    are looking at
unsigned int currBlockOrder = blockIdx.y/d_shared[dims3];//compute
    diagnally, the current MingCi of the block
unsigned int blockx = x - currBlockOrder;//the block pos of x
unsigned int blocky = y + currBlockOrder;//the block pos of y
unsigned int tIDx = threadIdx.x;//current thread id of x
unsigned int tIDy = threadIdx.y;//current thread id of y
int seqIdx = tIDx + currAlign * d_shared[max_x] + blockx * SHARED_X;//
    shorter read
int refIdx = tIDy + currAlign * d_shared[max_y] + blocky * SHARED_Y;//
    longer ref
```

Listing 3.7 Phase 2 Stage 2: cell assignment for thread in the matrix

Listing 3.7 is for computing the exact location of the thread in the matrix, which calculates the $blockx$, $blocky$, $tIDx$, $tIDy$, $bIDx$, and $bIDy$. These parameters are important as they pinpoint which nucleotide in the sequences for each threads.

```
//first block row first row
//we have multiple blocks, therefore, we have to be very carefull
if(!blocky && !tIDy) {//when tIDy is 0, which would be the first line
  h_matrix[0][tIDx] = d_row[seqIdx];
}
if(!blockx && !tIDx){//tIDx is 0, left column
  h_matrix[tIDy][0] = d_col[refIdx];
  //if(!tIDx&&!tIDy) h_matrix[0][SHARED_X] = d_col[refIdx]
}
//surrounded line that we have to copy them from computed d_row and d_col
//blocky is > 0
int idx = 0;
if (blocky && !tIDy){
  //(x, y-1)
  idx = get1DIdx(0, currAlign,blocky-1,blockx);
  if(tIDx) h_matrix[0][tIDx] = d_scoringMatrix[idx].h_value[SHARED_Y-1][
    tIDx-1];
    f_matrix[0][tIDx] = d_scoringMatrix[idx].f_value[SHARED_Y-1][tIDx];//
    for restoring previous h

}
else if(blockx && !tIDx && tIDy){
  idx = get1DIdx(0, currAlign,blocky,blockx-1);
  //(x-1, y)
  h_matrix[tIDy][0] = d_scoringMatrix[idx].h_value[tIDy-1][SHARED_X-1];

}

if(blockx&& ! tIDx){
  idx = get1DIdx(0, currAlign,blocky,blockx-1);
  //(x-1, y)
    e_matrix[tIDy][0] = d_scoringMatrix[idx].e_value[tIDy];//for restoring
    previous e

}
if (blockx && blocky && ! tIDx && !tIDy){
  idx = get1DIdx(0, currAlign,blocky-1,blockx-1);
  //(x-1,y-1)
  h_matrix[0][0] = d_scoringMatrix[idx].h_value[SHARED_Y-1][SHARED_X-1];

}
/**
 * tXM1 and tYM1 are to store the current value of the thread Index. tIDx
    and tIDy are
 * both increased with 1 later on.
 */
unsigned int tXM1 = tIDx;
unsigned int tYM1 = tIDy;
// shared location for the parts of the 2 sequences, for faster retrieval
    later on:
__shared__ uint8_t s_seq[SHARED_X];
__shared__ uint8_t s_ref[SHARED_Y];

// copy sequence data to shared memory (shared is much faster than global)
if (!tIDy){
  s_seq[tIDx] = d_sequences[seqIdx];
}
if (!tIDx){
  s_ref[tIDy] = d_references[refIdx];

}
__syncthreads();
// set inner score (aka sequence match/mismatch score):
uint8_t charSeq = s_seq[tIDx];
uint8_t charRef = s_ref[tIDy];


innerScore = charSeq == FILL_CHARACTER || charRef == FILL_CHARACTER ?
    FILL_SCORE : d_mat[charSeq+charRef*5];
// transpose the index
++tIDx;
++tIDy;
// set shared matrix to zero (starting point!)
// wait until all elements have been copied to the shared memory block
/**** sync barrier ****/
```

Listing 3.8 Phase 2 Stage 3: surrounding cells initialization in matrices

Listing 3.8 is for border cells initialization. For border blocks, part or all of their cells are initialized using *h_col* and *h_row*, and other cells are initialized using previously computed matrices. After successful initialization, the current matching score is computed and stored into the variable *innerscore*.

```
1  for (int i=0; i < DIAGONAL; ++i) {
2  if(innerScore!=FILL_SCORE){
3    if (i == tXM1+ tYM1) {
4      // calculate only when there are two valid characters
5      // this is necessary when the two sequences are not of equal length
6      // this is the SW-scoring of the cell:
7      // At the beginning of the loop: eh[j] = { H(i-1,j-1), E(i,j) }, f = F
     (i,j) and h1 = H(i,j-1)
8      // Similar to SSE2-SW, cells are computed in the following order:
9      // H(i,j) = max{H(i-1,j-1)+S(i,j), E(i,j), F(i,j)}
10     // E(i+1,j) = max{H(i,j)-gapo, E(i,j)} - gape
11     // F(i,j+1) = max{H(i,j)-gapo, F(i,j)} - gape
12     int M = h_matrix[tYM1][tXM1]? h_matrix[tYM1][tXM1] + innerScore: 0;
13     //m_matrix[tYM1][tXM1] = M;
14     h_matrix[tIDy][tIDx] = max(max(M, e_matrix[tYM1][tXM1]), f_matrix[tYM1
     ][tXM1]);
15     e_matrix[tYM1][tIDx] = max(max(M-d_shared[oe_ins], e_matrix[tYM1][tXM1
     ]-d_shared[e_ins]), 0);
16     f_matrix[tIDy][tXM1] = max(max(M-d_shared[oe_del], f_matrix[tYM1][tXM1
     ]-d_shared[e_del]), 0);
17   }
18 }
19
20 if(i-1 == tXM1 + tYM1){
21   if(!tXM1){
22     s_maxima[tYM1] = h_matrix[tIDy][1];
23     x_maxloc[tYM1] = tXM1;
24   }
25   else if(getHigher(h_matrix[tIDy][tIDx], s_maxima[tYM1], &s_maxima[tYM1])
    )
26       x_maxloc[tYM1] = tXM1;
27
28 }
29 // wait until all threads have calculated their new score
30   /**** sync barrier ****/
31 __syncthreads();
32 }
```

Listing 3.9 Phase 2 Stage 4: matrices value computation

Listing 3.9 is for computing matrices' value, where the cells are computed anti-diagonally. The first code block (in Listing 3.9, from line 2 to line 18) computes the H, E, F values. The second code block (in Listing 3.9, from line 20 to line 28) finds its maximum value and location from computed cells, which compares the previous highest value with the current value. *__syncthreads()* is executed at the end of each cycle as we wanted to use the previously calculated values.

```
//pass on the information to the next block
//here we modify for our diagnalLine
//int idx = get1DIdx(blockx, blocky, XdivSHARED_X);
idx = get1DIdx(0, currAlign,blocky,blockx);
d_scoringMatrix[idx].h_value[tYM1][tXM1] = h_matrix[tIDy][tIDx];
d_scoringMatrix[idx].f_value[tYM1][tXM1] = f_matrix[tIDy][tXM1];

//stored for next time computation
if(!tXM1){
  d_scoringMatrix[idx].e_value[tYM1]=e_matrix[tYM1][SHARED_X];
  d_scoringMatrix[idx].s_value[tYM1]=s_maxima[tYM1];
  d_scoringMatrix[idx].x_value[tYM1]=x_maxloc[tYM1];

}
__syncthreads();
```

Listing 3.10 Phase 2 Stage 5: data transferring

The code in Listing 3.10 copies the final result back to the global memory. Phase 2 is the main GPGPU core program for computing matrices.

3.4.5.3 Phase 3: Production of Profiles as Output of Results

As *ksw_ extend2* produces the best global alignment in the query and reference, the query's target length, query's full alignment score, and the max off-diagonal distance have to be produced GPGPU version as well. Therefore, after successfully obtaining the scoring matrix, an algorithm is needed to compute these final values. Such an algorithm is implemented in Phase 2.

3.4.6 ksw_extend2 with Memory-saving Version Implementation

The major difference between the memory-saving version and the time-saving version is in Phase 2. Instead of storing matrices in all blocks[15], only one array of blocks is kept, where the number of blocks is twice the size[16] of the number of blocks on the longest diagonal in the matrix. However, only storing single array of blocks increase the number of times for data transfer. The sacrifice of saving memory is the overhead caused by data transferring between host and GPGPU.

[15]We define a block as the smallest data storage that stores a square matrix of computed values.

[16]At most, two diagonals of blocks are computed.

```
checkCudaErrors(cudaMemcpy(&h_diagnalLine[0], d_diagnalLinePre, sizeof(
    h_diagnalLine), cudaMemcpyDeviceToHost));

LocalMatrix* temp = &*d_diagnalLinePre;
d_diagnalLinePre = &*d_diagnalLine;
d_diagnalLine = temp;
for(int currBlockOrder = 0; currBlockOrder < numBlocks; currBlockOrder++){
  for(int currAlign = 0; currAlign < *(h_shared +dims3); currAlign++){
    int blockIdx_y = currBlockOrder * *(h_shared +dims3) + currAlign;
    int calgn = currAlign * *(h_shared +block_y_len) * *(h_shared +
    block_x_len);
    int blocky = (y + currBlockOrder) * *(h_shared +block_x_len);
    int blockx = (x - currBlockOrder);
    memcpy((h_scoringMatrix + calgn + blocky + blockx), (&h_diagnalLine
    [0]+blockIdx_y), sizeof(LocalMatrix));
  }
}
```

Listing 3.11 Phase 3: save RAM version data transferring

Listing 3.11 is the major difference between the two versions. At the beginning of the $CalculateScore$, the memory-saving version only passes in scoring matrix size of two diagonals instead of the whole square. However, the computed value is needed to be copied into the host memory at the end of each computation cycle, which increased the time dramatically.

4 TEST RESULTS

In this chapter, three series of tests were performed. The first series of tests were done at parallelization *Level 1*. The second series of tests were performed at parallelization *Level 2* and *Level 3*. The last series of tests were for *GPGPU distributed* version of *BWA-MEM* parallelization *Level 1*.

Three tests were performed at parallelization Level 1. Parallelization *Level 1* test 1 was performed to understand the effect of different sequence similarity levels without the pruning mechanism. The GPGPU versions' time consumption increases with the alignment's length in a linear relation, and it is not affected by the level of sequence similarity. the CPU version's time consumption has a positive correlation with the alignment length, and further increases with the sequence similarity level. The second test aimed to compare the performance with or without the pruning mechanism. And the third test was to check if the second test's remains the same for longer sequence. With pruning mechanism, the time consumption of GPGPU version is always higher than the CPU version. this is because the parallelization level is low.

Another three tests were also performed at parallelization Level 1 and Level 2. In the test with real data, *BWA-MEM* has shown that there are only few extensions on average at the chain level. Therefore, before *Level 3* parallelization started, we have to collect as much fragment pairs as possible. During the test, it has shown that the length of the fragments is rarely longer than *32 bp* in seed extension. Therefore, we designed this test to find out the performance of aligning multiple pairs in the same time.

In the last series of tests, two different version of GPU *ksw_ extend2* is created. The tests check out performance on alignment pair with *8, 16*, and *32 bp* length. *GPU with time-saving* version is aimed at reducing the time consumption, which is assuming the time is the most important aspect of the program. *GPU with memory-saving* version is aimed at reducing RAM usage, which is assuming that GPGPU has a limited number of RAM. From the test, CPU version's time consumption increases very fast, *GPU with memory-saving* version is slower as it has a huge overhead transferring information in and out of GPU.

As the new stable release version of *Spark* recognizes GPGPU as resources, it just give us an opportunity to build a *GPGPU distributed* version of *BWA-MEM*. The issue before is that *SharcNET* only support *Spark* standalone mode. The test result we have given is not based on performance, but the correctness of the input and output. Both CPU version of *BWA-MEM* and the *Spark* standalone version of GPGPU *BWA-MEM* is tested, and their results are the same. However, there is limitations, as the parallelization level for *BWA-MEM* is only at *Level 1*. However, it is proof of concept that our program is run-able in standalone mode.

4.1 The Generation of the Performance Data

The read and reference sequences (real data) are obtained from NCBI Sequence Reads Archive, where the reads were obtained from *SRR002062*[1], and reference is obtained from *SGD1.01:2*[2]. We have performed a series of tests to understand the alignment process of *BWA-MEM*. In these tests, only two to three alignments are performed in each chain. As the GPGPU version of the seed extension will only show performance improvement when multiple extension alignments are performed simultaneously, it is not ideal for testing with data obtained from the NCBI Sequence Reads Archive. The structural change is needed in *BWA-MEM* as we

[1]Ranged from *SRR002062.1.1* to *SRR002062.30.1*, cDNA fragmentation by DNase I (SRR002062) from the Sequence Reads Archive.

[2]SGD1.01.2 is a yeast reference genome sequence.

want to perform as much extension as possible.

In the following tests, the test data sets used are generated from the random number generator (from C). We are using simulated data because we want to have a large amount of these sequences to show performance improvement. The differences between simulated reference fragment and query fragment within the same pair are primarily single-nucleotide mutations. Therefore, in each pair of the test data sets, a simulated reference fragment is generated with a random number generator, and the read sequence is basically a copy of the simulated reference fragment with single nucleotide mutations introduced.

The performance results are obtained by measuring the run time of both the CPU version and the GPGPU version of *ksw_ extend2*. As previously mentioned, in terms of time measured in the performance test, the time measurement tool is different for CPU and GPGPU programs. Different tools are used because GPGPU programs could only be measured using CUDA Tool Kit, and CPU programs could only be measured using their tool. In Listing 4.1, the time is recorded starting line 1 and line 4 and stopped at line 29 for the GPGPU program. However, the CPU part of the *ksw_ extend2* GPGPU version is not measured as the measurement tool is different. Listing 4.2 shows how the time is measured with the CPU program, which starts from line 4 and stops at line 9. The way of measurement is the same for all the tests. When testing the code, the time spent by both versions are recorded using the same set of data.

```
1  cudaEvent_t start, stop;
2  cudaEventCreate(&start);
3  cudaEventCreate(&stop);
4  cudaEventRecord(start);
5  // adjust $w if it is too large
6  //locate memory for d_eh
7  for (unsigned int i = 1; LIKELY(i < halfZhouChang); ++i) {
8       numBlocks = i <= xiaoBian? i : i >= daBian? halfZhouChang - i :
      xiaoBian;
9
10      //reserve dim4 for anything beyound chain
11      dim3 dimSWGrid(*(h_shared +dims4), *(h_shared +dims3) * numBlocks, 1);
      //numBlocks
12
13      calculateScore_v2<<<dimSWGrid, dimBlock>>>(
14          d_scoringMatrix,
15          d_row, d_col,
16          x, y,
        numBlocks,
        d_sequences, d_references);

    cudaDeviceSynchronize();

    if (x == *(h_shared +block_x_len) - 1)
    ++y;
    if (x < *(h_shared +block_x_len) - 1)
    ++x;
}
checkCudaErrors(cudaMemcpy(h_scoringMatrix, d_scoringMatrix, sizeof(t_sc),
    cudaMemcpyDeviceToHost));

cudaEventSynchronize(stop);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
printf("%3.1f,", milliseconds);
```

Listing 4.1 How the GPGPU version's time measurement is obtained

```
struct timeval stop, start;
gettimeofday(&start, NULL);

for (int i = 0;i< *(h_shared + dims3); i++) {
    (h_swext+i)->score = ksw_extend2((h_swext+i)->qlen, (h_swext+i)->query
   , (h_swext+i)->tlen, (h_swext+i)->target, 5, opt->mat, opt->o_del, opt
   ->e_del, opt->o_ins, opt->e_ins, (h_swext+i)->aw[1],
    opt->pen_clip3, opt->zdrop,(h_swext+i)->sc0, &((h_swext+i)->qle),&(
   h_swext+i)->tle, &(h_swext+i)->gtle, &(h_swext+i)->gscore, &(h_swext+i)
   ->max_off[1]);
}

gettimeofday(&stop, NULL);
printf("%lu\n", (stop.tv_sec - start.tv_sec) * 1000000 + stop.tv_usec -
   start.tv_usec);
```

Listing 4.2 How the CPU version's time consumption is obtained

4.2 Level 1 Parallelization

The first step is to generate alignment data sets before the test is started using a random number generator, which provides two similar sequences with some mutations in each alignment data set. However, the effect of different mutation rates are needed to be tested in controlled environments. Therefore, the first test determines the performance differences between 90, 92, 96, and 98 percent similarities. How the pruning mechanism affects the overall performance has to be determined as well.

Therefore, the following tests were performed:

- Relations between sequence alignment length with or without the pruning mechanism, and how mutation rate affects the time consumption.
- Relation between GPGPU version and CPU version with the pruning mechanism.
- Relation between GPGPU version and CPU version with or without the pruning mechanism in the long run.

4.2.1 Test Data Set Generation with Random Number Generator for Parallelization *Level 1*

As described in Section 4.1, we generate test data sets with random number generator. The test data sets we are going to generate are the fragment pairs. For example, in Table 3.14, *simulated left reference sequence alignment fragment* and *simulated left read sequence alignment fragment* is one pair of alignment fragments, and *simulated right reference alignment fragment* and *simulated right read alignment fragment* is another pair of fragments. In each pair, two fragments may having a different length. For example, the length of *simulated left read sequence alignment fragment* is 1, and the length of *simulated left reference sequence alignment fragment* is 2. also, the difference between *simulated right reference sequence alignment fragment* and *simulated right read sequence alignment fragment* is only the first nucleotide, which is caused by single nucleotide mutation.

In generating each pair of fragments, we first generate the first fragment's base at location i. If the length if the second fragment is bigger than i, we either copy the first fragment's base at location i or *generate another random base*[3]. Listing 4.3 is the function for generate these data sets. In Listing 4.3, from Line 23 to Line 32, a pair of fragments is generated with length of *total_length*. A seed is defined and used at Line 17 to produce a consistent testing dataset. Two char arrays were being prepared (two sequences, or a pair of alignments) with the length of *total_length* (from Line 19 to Line 21). For each character of the sequence fragment, we

[3]This depends on our predefined mutation *chance* value.

do the following. First, at Line 25, a random number from 1 to 100 is generated. Then, on line 28, a random number between 0 and 4 is generated, which represents the nucleotide on the sequence. On Line 30, a predefined value *percent* and the randomly generated number *chance* is used to determine if we want to replace the current nucleotide in the seed with a new random number of 1 to 4. The higher the value of *percent*, the more similar of two generated fragments are in the pair.

```
 1 void lvl_1_test(int total_length, int factor, int percent){
 2     //options for computing alignment
 3     mem_opt_t *opt;
 4     //these variables are the will be the output from our computation
 5     int bqle, btle, bgtle, bgscore;
 6     opt = mem_opt_init();
 7     bwa_fill_scmat(opt->a, opt->b, opt->mat);
 8     int l_query = total_length; //length of the query
 9     int qe = 0; //qe is the starting position of the right query
10     int re = 0; //re is the starting position of the right reference
11     int64_t rmax[2] = { 0, total_length };
12     int aw[2] = { 100, 100 };
13     int sc0 = 100;  //highest score
14     int qle, tle, gtle, gscore;
15     int max_off[2] = { 0, 0 };
16     //set seed to be 6
17     srand(6);
18     //reference sequence
19     uint8_t rseq[total_length];
20     //read sequence
21     uint8_t query[total_length];
22     //the code down below will generate a sequence with length of "
    total_length"
23     for(int k = 0; k < total_length; k++){
24         //we generate a random number between 1 and 100
25         int chance = rand()%100;
26         //and we randomly generate a random number between 1 and 4
27         //each of them is representing a nucleotide
28         rseq[k] = rand()%4;
29         //if the random generated number is bigger than the current
    percent, we substitute the number with another random number
30         if(chance > percent){query[k]= rand()%4;}
31         else query[k] = rseq[k];
32     }
33     //CPU version of the code
34     score = ksw_extend2(l_query - qe, query + qe, rmax[1] - rmax[0] - re,
    rseq + re, 5, opt->mat, opt->o_del, opt->e_del, opt->o_ins, opt->e_ins,
     aw[1], opt->pen_clip3, opt->zdrop, sc0, &qle, &tle, &gtle, &gscore, &
    max_off[1]);
35     //GPU version of the code
36     score = gpu_sw_extend(l_query - qe, query + qe, rmax[1] - rmax[0] - re
    , rseq + re, 5, opt->mat, opt->o_del, opt->e_del, opt->o_ins, opt->
    e_ins, aw[1], opt->pen_clip3, opt->zdrop, sc0, &bqle, &btle, &bgtle, &
    bgscore, &bmax_off[1]);
37 }
```

Listing 4.3 *Level 1* parallelization test data generation

4.2.2 Test 1: Sequence Alignment Similarity and Time Cost without the pruning mechanism

The effect of the mutation rate is studied in the first test, and the main goal of this test is to examine the influences of different similarities. The assumption is that the CPU version might increase when the similarities increase (the higher the similarities, the more cells are needed to compute). However, the GPGPU version's time consumption may not increase as the number of steps taken has not changed.

Table 4.1 CPU and GPGPU versions alignment performance comparisons without the pruning mechanism with different sequence similarity levels at parallelization *Level 1*[1].

Length (bp)[2]	CPU 90%[3]	GPGPU 90%[4]	CPU 92%[5]	GPGPU 92%[6]	CPU 94%[7]	GPGPU 94%[8]	CPU 96%[9]	GPGPU 96%[10]	CPU 98%[11]	GPGPU 98%[12]
80	0.08	0.2	0.07	0.2	0.07	0.2	0.08	0.2	0.08	0.2
160	0.27	0.4	0.27	0.4	0.27	0.4	0.27	0.4	0.26	0.4
320	0.81	0.8	0.81	0.8	0.8	0.8	0.81	1.3	0.85	0.8
480	1.56	1.2	1.56	1.2	1.6	1.7	1.62	1.2	1.73	1.2
640	2.54	1.6	2.54	1.6	2.66	1.6	2.71	2.1	2.87	1.7
800	3.74	2.5	3.78	2.1	4.04	2.5	4.07	2.5	4.34	2.1
960	5.21	2.5	5.26	2.5	5.55	2.5	5.69	2.5	5.99	2.6

[1]Reduced Data Set, the full data set is displayed in Appendix B.1.

[2]Sequence alignments length ranges from 80 to *960 bp*.

[3-12]CPU and GPGPU Versions performance ranges from 90 to 98 percent.

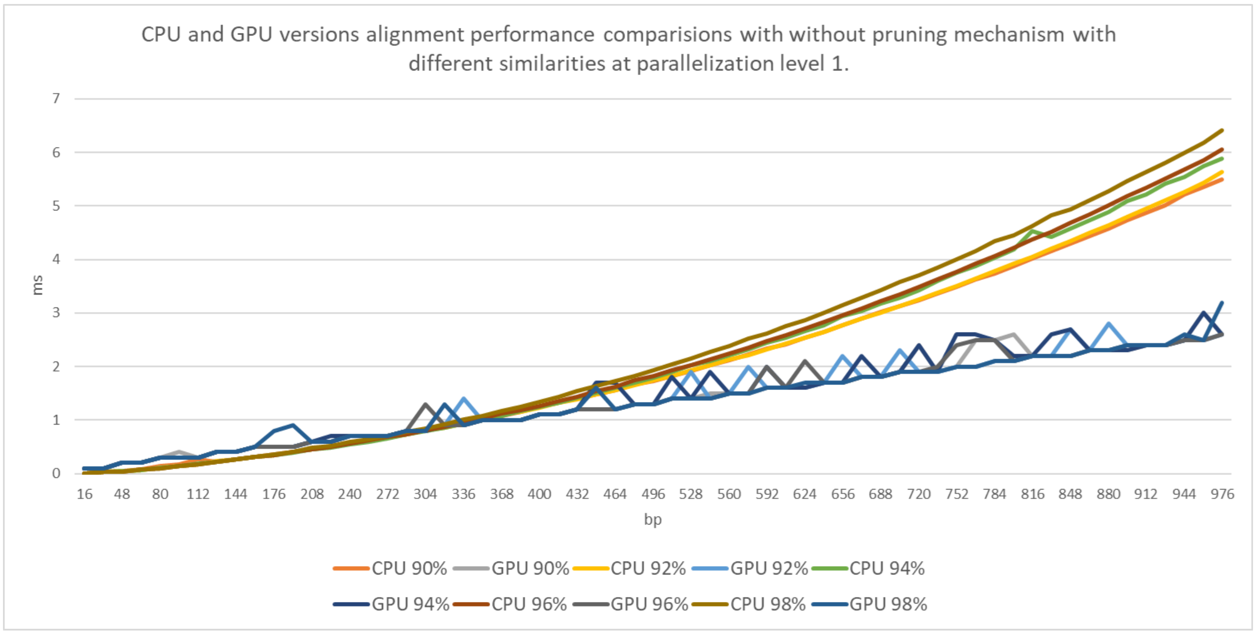


Figure 4.1 CPU and GPGPU versions alignment performance comparisons with without the pruning mechanism with different similarities at parallelization *Level 1*[1]. The *x-axis* represents the time consumption in *ms*, and the *y-axis* represents sequence length from *16* to *976 bp*. Sequence similarity level represented by colour ranges from *92* to *98* percent.

[1]This figure is basically the combination of two previous figures.

Table 4.1 compares the performance of the CPU and GPGPU versions without the pruning mechanism with different sequence similarity levels at parallelization *Level 1*. These performance data are generated using the code from Listing 4.3. For GPGPU versions, the time is obtained using *cudaEvent*. For CPU versions, the C function $gettimeofday$ is being used. The test is performed once for each different length. However, for each length, the test data set remains the same for both versions. A more detailed version of the data is shown in Appendix B.1. However, the test data changes as the random-generated-sets change for each length, which may contribute to some of the time consumption differences across the different length.

The time for the CPU version[4] has a slight increase when the similarity increases. At *90*

[4]The test was performed on intel i7-6700k CPU, NVIDIA GeForce GTX 1080, 32GB RAM, and 1TB

percent similarity, the alignment time at *960 bp* is *5.21 ms*, and this time is *5.26, 5.55, 5.69, 5.99 ms* at *92, 94, 96, and 98* percent similarity, respectively. The increasing similarities cause a slight increase in time consumption as more nucleotides are needed to be compared. On the opposite, when similarity is low, the alignment reaches zero scores (with the gap penalty) much faster, resulting in an alignment of fewer nucleotides. As shown in Figure 4.1, the time for the GPGPU version has not changed even when the similarity increases. At 90 percent similarity, the alignment time at *960 bp* is *2.5 ms*, and this time is *2.5, 2.5, 2.5, 2.6 ms* at *92, 94, 96, and 98* percent similarity, respectively. it can be seen that GPGPU lines overlap with each other, which means the coefficients remain the same for all similarity levels. Therefore, without the pruning mechanism, the GPGPU version of the code has a linear relation (from the graph), and the time consumption does not change when the sequence similarity changes.

From Figure 4.1 and Table 4.1, the result can be summarised as follows:

1. Under the case without pruning mechanism, GPGPU version's time consumption increases with the alignments' length in a linear relation, which is not affected by the level of sequence similarity.

2. For the CPU version, the time consumption also shows a positive correlation with the alignment length, and it further increases with the sequence similarity level.

4.2.3 Test 2: The Effect of the Pruning Mechanism Towards CPU and GPGPU versions Alignment Performance

After knowing the influence of similarities, this Section examines the pruning mechanism's influence, and the similarity of 98% is used for this test. Therefore, both the CPU and GPGPU versions' performance both with the pruning mechanism and without the pruning mechanism is tested. As the pruning mechanism skips the cells in the areas that are not

solid state hard drive desktop.

possible to produce the optimal score, for the CPU version, the order of time complexity should decrease when the pruning mechanism is applied. However, the time complexity for GPGPU without the pruning mechanism should remain the same.

Table 4.2 CPU and GPGPU versions alignment performance comparisons with or without the pruning mechanism at parallelization *Level 1*[1].

length (bp)[2]	CPU version with the Pruning Mechanism (ms)	CPU version without the Pruning Mechanism (ms)	GPU version without the Pruning Mechanism (ms)	GPU version with the Pruning Mechanism (ms)
48	0.03	0.03	0.1	0.1
192	0.33	0.36	0.5	0.5
336	0.65	0.92	0.9	0.9
480	0.97	1.73	1.7	1.3
624	1.29	2.75	1.6	1.6
768	1.62	4	2	2
960	2.05	5.99	2.5	2.5

[1]Reduced Data Set, Full Set Data is Displayed in Appendix B.2.

[2]Sequence alignments length ranges from *48* to *960 bp*.

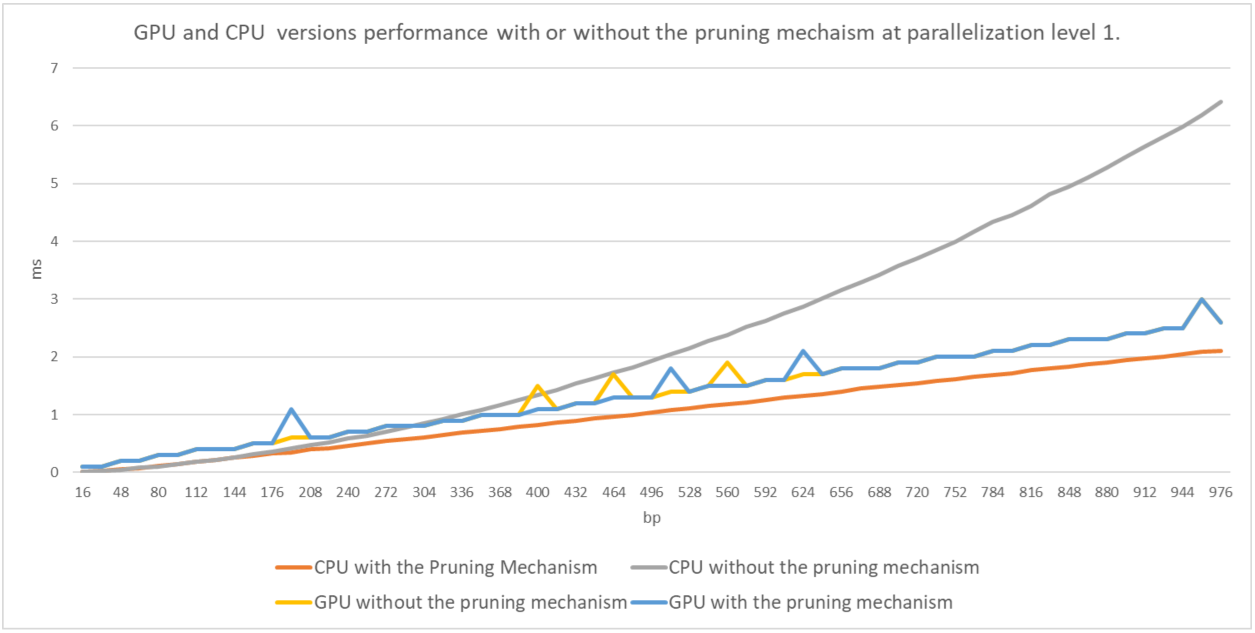


Figure 4.2 GPU and CPU versions performance comparisons with or without the pruning mechanism at parallelization *Level 1*. The y-axis represents the time consumption in *ms*, and the *x-axis* represents sequence length from *16* to *976 bp*.

Table 4.2 describes the result from test 2 generated using the code from Listing 4.3. Figure 4.2 shows the difference between CPU with the pruning mechanism and CPU without the pruning mechanism. The CPU version without the Pruning mechanism quickly increased with the sequence length from *0.03 ms* at *48 bp* to *5.99 ms* at *960 bp*. However, the CPU with the pruning mechanism increased is much slower, with the sequence length from *0.03 ms* at *48 bp* to the sequence length of *2.05 ms* at *960 bp*, indicating that the time consummation decreases when the pruning mechanism is applied.

In Figure 4.2, the GPGPU version with the pruning mechanism increased from *0.1 ms* at length *48 bp* to *2.5 ms* at *960 bp*, and the GPGPU version without the pruning mechanism remains the same (increased from *0.1 ms* at length *48 bp* to *2.5 ms* at *960 bp*), indicating that the pruning mechanism does not affect the GPGPU version. CPU with the pruning mechanism and GPGPU with the pruning mechanism both have a similar increasing trend with the sequence length. However, GPGPU with the pruning mechanism has a much higher speed of increase, suggesting that GPGPU with the pruning mechanism would never beat CPU with the pruning mechanism.

Based on data in Figure 4.2 and Table 4.2, it can be concluded that the pruning has no effect on GPU's performance, but improves CPU's performance dramatically. More specifically, before 144 bp, CPU's performance with or without the pruning mechanism remains the same. However, pruning starts to improve the performance at *144 bp*. The cause of this is due to a large number of cells been cut off, and the pruning mechanism starts to show the real effect.

4.2.4 Test 3: The Effect of the Pruning Mechanism towards CPU and GPGPU Versions Alignment Performance with Longer Sequence

After the first and second experiments, we can conclude that the following statement is true. Under the case without the pruning mechanism, GPGPU version time consumption has a linear relation with the alignment length, but it is not affected by the level of sequence similarity. On the other hand, the CPU version's time consumption has a positive correlation with the alignment length, and it further increases with the sequence similarity level. The pruning mechanism does not affect GPU's performance but improves CPU's performance dramatically. However, this experiment may change for longer sequences. Therefore, we are going to test the sequence's length as long as possible. We stop at *7416 bp* as the computer becomes extremely slow.

Table 4.3 CPU and GPGPU versions alignment performance comparisons with or without the pruning mechanism up to *7416 bp* length at parallelization *Level 1*[1].

Length (bp)[2]	CPU version without the Pruning Mechanism (ms)	CPU version with the Pruning Mechanism (ms)	GPU version without the Pruning Mechanism (ms)
16	0	0.01	0.1
616	2.7	1.27	1.6
1816	19.7	3.98	4.9
3016	52.4	6.73	9.3
4216	101.5	9.44	13.8
5416	167.1	12.2	19.2
6616	247.3	15.03	25.5
7416	309	16.55	30.3

[1]Reduced Data Set, Full Set Data is Displayed in Appendix B.3.

[2]Sequence alignments length ranges from 16 to *7416 bp*.

Table 4.3 describes the time consumption of CPU and GPGPU versions with or without the pruning mechanism for sequence up to 7416 bp, which was generated using the code from Listing 4.3. Since the GPGPU version remains the same with or without the pruning mechanism, only the GPGPU with the pruning mechanism is tested.

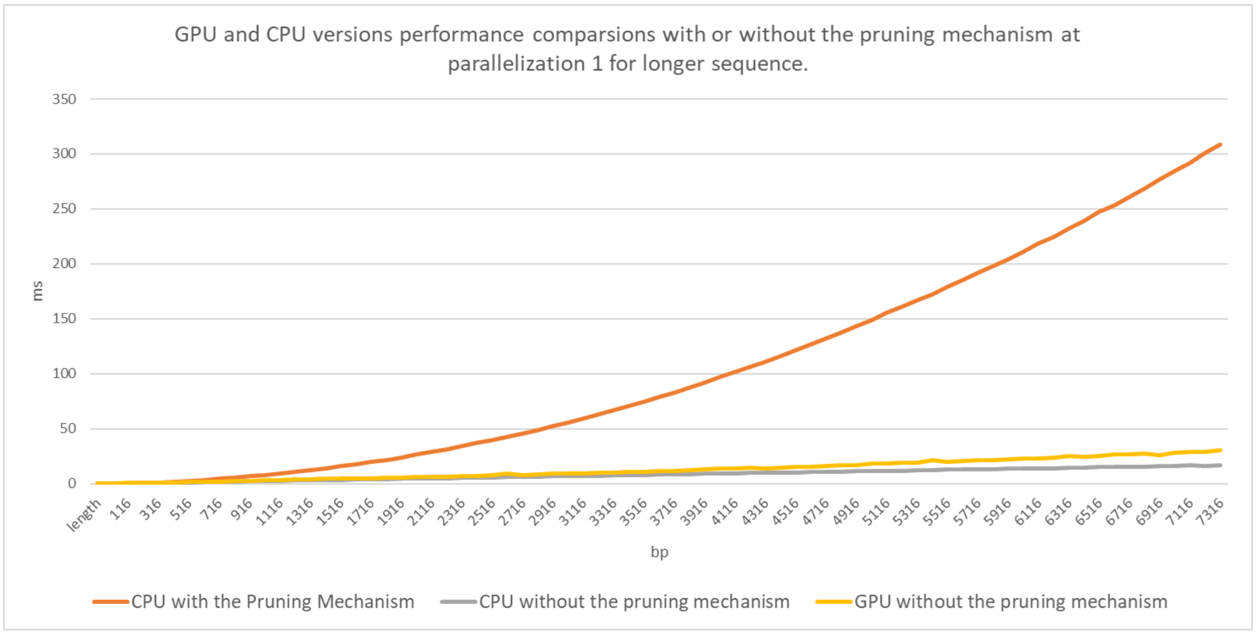


Figure 4.3 GPU and CPU versions performance comparisons with or without the pruning mechanism at parallelization *Level 1*. The y-axis represents the time consumption in ms, and the x-axis represents sequence length from *16* to *7416 bp*.

As shown in Figure 4.3, the time for CPU with the pruning mechanism increased from *0.01 ms* at *16 bp* to *16.55 ms* at *7416 bp*. The GPU's time with the pruning mechanism increased slightly faster than the CPU with the pruning mechanism from *0.1 ms* at *16 bp* to *30.3 ms* at *7416 bp*. However, the CPU time without pruning version quickly increased from *0 ms* at *16 bp* to *309 ms* at *7416 bp*, much quicker than the GPGPU version without the pruning mechanism. The length of the sequences are being tested from *16 bp* to *7116 bp*. After *7116 bp*, there is not enough RAM for the GPGPU to compute it at the same time. The limitation is that it depends on how big the RAM is.

4.2.5 Summary

At this level, all tests was carried out on a intel i7-6700k CPU, NVIDIA GeForce GTX 1080, 32GB RAM, and 1TB solid state hard drive desktop, which previously referred to as PC 1.

The main goal of parallelization Level 1 test 1 is aimed to understand the effect of different sequence similarity levels. Tests was performed at similarity Level 92%, 94%, 96%, 98% with the length range of 16 bp to 992 bp. The main goal of test 2 is to understand the effect of pruning mechanism. Tests was performed on both CPU and GPGPU versions with or without the pruning mechanism at similarity level of 90% and a length range of 16 bp to 992 bp. Pruning mechanism eliminates cells that cannot generate a higher score than the exiting maximum score. The main goal for Level 1 test 3 is to check out if the outcome from test 2 remains the same for longer sequences. The tests was performed with a similarity Level of 90% and a length range of 16 bp to 992 bp. At this level, the same data set is used for the same similarity level and length, and the mutations are generated randomly at random locations on the sequence. Under the case without the pruning mechanism, GPGPU version's time consumption increases with the alignment's length in a linear relation, and is not affected by the level of sequence similarity. CPU version's time consumption has shown a positive correlation with the alignment length, and further increases with the sequence similarity level. Under the case without the pruning mechanism, the time consumption of the GPGPU version is always higher than the CPU version.

4.3 Parallelization Level 2 and Level 3

In the tests with real read data[5], *BWA-MEM* has shown that there are only a few extensions on average at the chain level, referred to as *Level 2* parallelization. In *Level 3* parallelization, multiple chains are combined for the maximum number of chains that can be collected. In the tests using data mentioned from section 4.1, the length of the fragments are rarely longer than *32 bp* in seed extension. Therefore, the sequences' length in the tests are shorter than 32 bp, including *8 bp*, *16 bp*, and *32 bp* length tests. The time consumption on *GPU with time-saving* and memory-saving versions and CPU versions are measured in each test, and

[5]Section 4.1.

these three versions are described below.

1. *GPU with memory-saving Version*: the program is designed under the assumption that saving memory was much more important than saving time.

2. *GPU with time-saving Version*: The program is designed to assume that saving time is much more important than saving memory.

3. *CPU Version*: the original version of *ksw_extend2*.

4.3.1 How the Sequence Data has been Generated and Used in Parallelization Level 2 and Level 3

Listing 4.4 describes how the sequences are generated and used in the test. First, all the necessary matrices and variables were initialized from Line 43 to Line 48. Each alignment is processed and initialized using function sw_ext_int (Line 60). The sequence generation is performed by function ref_seq_gen at Line 2, similar to the previous result from the *Level 1* test.

```
1  /*this method pass in a pointer address and generate query and target test
       references*/
2  void ref_seq_gen(uint8_t **query, uint8_t **target, int query_size, int
      target_size, int chance_percent){
3
4     (*query) = (uint8_t*)calloc(query_size,sizeof(uint8_t));
5      (*target) = (uint8_t*)calloc(target_size,sizeof(uint8_t));
6      uint8_t *tmp_query = (*query);
7      uint8_t *tmp_target = (*target);
8      bool has_longer_query = query_size > target_size;
9      int min_size;
10     int max_size;
11     int idx;
12     if(has_longer_query){
13         min_size = target_size;
14         max_size = query_size;
15     }else{
16         min_size = query_size;
17         max_size = target_size;
18     }
19
20     for(idx = 0; idx < min_size; idx++){
21         //generate a chance digit
22         int chance = rand()%100;
23         //generate a random nucleotide
24         *tmp_query = rand()%4;
```

```
        if(chance > chance_percent) *tmp_target = rand()%4;
        else *tmp_target = *tmp_query;
        tmp_query++;
        tmp_target++;
    }

    uint8_t *tmp_pt = has_longer_query? tmp_query : tmp_target;
    for(; idx < max_size; idx++){
        *tmp_pt = rand()%4;
        tmp_pt++;
    }
    return;
}


/*the following code is for~\textit{Level 2} and~\textit{Level 3} testing
    */
void lvl_2_3_testing(int test_size){
    //BWA\-MEM options
    mem_opt_t *opt;
    //option initialization
    opt = mem_opt_init();
    bwa_fill_scmat(opt->a, opt->b, opt->mat);
    //memory allocation for Smith-Waterman Extension structure
    struct sw_ext *h_swext = (sw_ext*)calloc(test_size,sizeof(sw_ext));

    //adding up testing information, assuming we are testing for 32 bp}
    length
    int g_qlen = 32;
    int g_tlen = 32;
    //assuming that we are generating sequences with mutation rate 80
    int chance = 80;
    int score_h0 = 100;
    //generate random number with random seed
    srand(RD_SEED);
    struct sw_ext *tmp_swext = h_swext;
    for(int i = 0; i < test_size; i++){
        sw_ext_int(&tmp_swext, score_h0, g_qlen, g_tlen, chance);
        tmp_swext++;
    }
    //generating shared information, getting ready for GPGPU computation
    int *h_shared;
    sw_state_init(&h_shared, test_size,1, opt);
    //initialization is over
    //testing starts here:
    LocalMatrix *result = gpu_sw_seed_extend(h_swext, h_shared, opt->mat);
    LocalMatrix *result2 = gpu_sw_seed_extend_v2(h_swext, h_shared, opt->
    mat);
    //computes the result from GPU, this function is an altered original
    code so it can compute the result from gpU
    for (int i = 0;i< test_size; i++) {
        computeMax((h_swext+i), h_shared, opt->mat, (result+i**(h_shared +
     block_x_len) * *(h_shared + block_y_len)), 5);
    }
    \\we compute CPU version
    for (int i = 0;i< *(h_shared + dims3); i++) {
        (h_swext+i)->score = ksw_extend2((h_swext+i)->qlen, (h_swext+i)->
    query, (h_swext+i)->tlen, (h_swext+i)->target, 5, opt->mat, opt->o_del,
     opt->e_del, opt->o_ins, opt->e_ins, (h_swext+i)->aw[1], opt->pen_clip3
    , opt->zdrop, (h_swext+i)->sc0, &((h_swext+i)->qle), &(h_swext+i)->tle,
     &(h_swext+i)->gtle, &(h_swext+i)->gscore, &(h_swext+i)->max_off[1]);
    }
```

```
}
```

Listing 4.4 *Level 2* and *Level 3* parallelization test data set generation

4.3.2 Test 1: Performance Comparisons Among Different Implementations at Parallelization Level 2 and Level 3 of ksw_extend2 with Alignments length at 8 bp

The first test performs a different number of alignments simultaneously, where all fragment pairs have the same length of *8 bp*. From the test we performed using data from Section 4.1, it is common to have short sequences[6]. If most sequences are short, it is possible to have all the sequences shorter than *8 bp*. When all the sequences are shorter than 8 bp, the GPGPU version will lengthen all of them to *8 bp*. Therefore, for this test, assuming that the length is equal to *8 bp* for all sequences.

Table 4.4 Performance comparisons among different implementations at parallelization *Level 2* and 3 of *ksw_ extend2* with alignments length at 8 bp[1].

Number of Alignments[2]	*GPU with memory-saving* Version(ms)	*GPU with time-saving* Version (ms)	CPU Version (ms)
30	0	0	0.035
630	0.2	0.1	0.687
1230	0.5	0.2	1.327
1830	0.7	0.2	1.994
2430	0.9	0.3	2.631
2550	0.9	0.3	2.746

[1]Reduced data set, full data set in AppendixB.4.

[2]The number of alignments performed in a single test.

Table 4.4 describes the time consumption of *CPU version*, *GPU with time-saving version* and *GPU with memory-saving version* with the sequence length of *8 bp*, which was generated

[6]The short sequences we are referring here are those around the seeds for seed extension inside each read.

using the code from Listing 4.4.

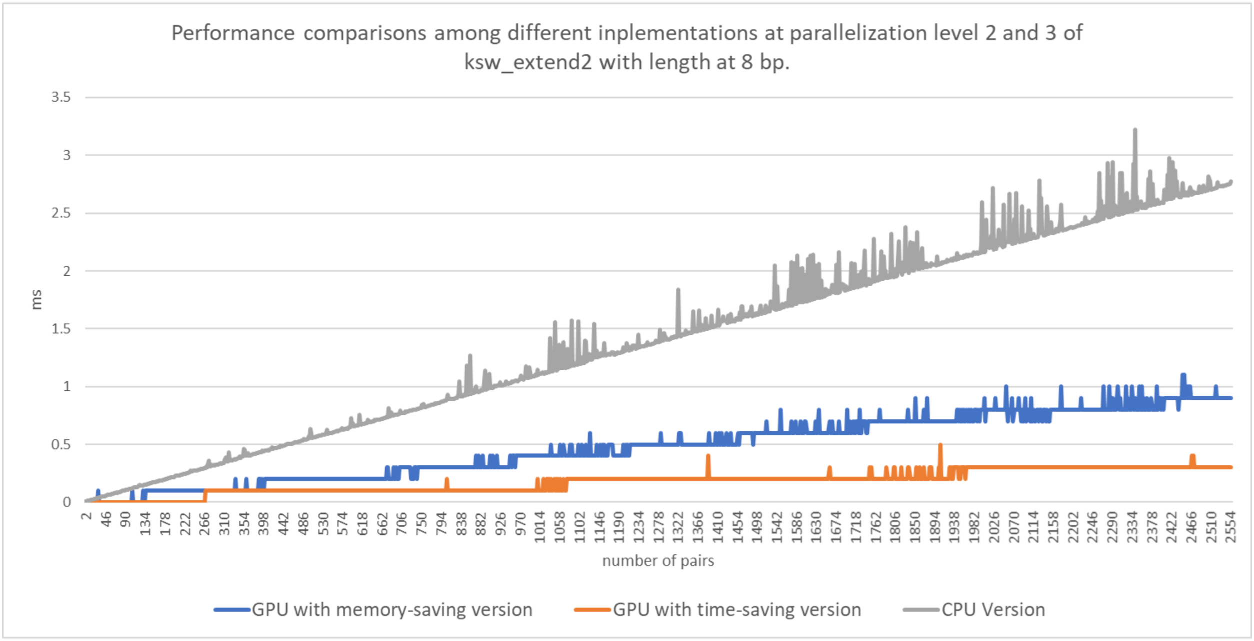


Figure 4.4 Performance comparisons among different implementations at parallelization *Level 2* and *Level 3* of *ksw_ extend2* with alignments length at *8 bp*. The y-axis represents the time consumption in ms, and the x-axis represents number of pairs from 2 to 2558.

Figure 4.4 shows the test result for the alignments at *8 bp*. The data showed that the *GPU with time-saving* version is much better than the other two versions. The time for the CPU version increased quickly with 30 alignments from *0.035 ms* at *30 bp* to *2.746 ms* with *2550* alignments, while both GPGPU versions show very little increase with 30 alignments[7] to *2550* alignments[8]. The *GPU with time-saving* version shows an excellent performance in comparison to the CPU version. However, it has much worse performance than the *GPU with time-saving* version. The conclusion is that the *GPU with time-saving* version was roughly nine times faster than the CPU version. The *GPU with memory-saving* version was roughly three times slower than the CPU version[9].

[7]Both at *0 ms*.

[8]Time-saving version at *0.3 ms*, memory-saving version at 0.9 ms.

[9]With the number of alignments at 2550, *GPU with memory-saving* version finished alignments in *0.9*

4.3.3 Test 2: Performance Comparisons Among Different Implementations at Parallelization Level 2 and Level 3 of ksw_extend2 with Alignments length at 16 bp

After the first test with a sequence length of 8 bp, all alignments' length may be longer than *8 bp* but shorter than *16 bp*. Therefore, the second test performs a different number of alignments simultaneously, where all alignments have a length of *16 bp*. When all the sequences are longer than *8 bp* and shorter than *16 bp*, the GPGPU version will convert all shorter than *16 bp* to *16 bp*. Therefore, for this test, assuming that the length is equal to *16 bp* for all sequences.

Table 4.5 Performance comparisons among different implementations at parallelization *Level 2* and *Level 3* of *ksw_extend2* with alignments length at *16 bp*[1].

Number of Alignments[2]	*GPU with memory-saving* Version(ms)	*GPU with time-saving* Version (ms)	CPU Version (ms)
30	0.1	0.1	0.114
630	1.1	0.3	2.338
1230	2.1	0.5	4.576
1830	2.4	0.8	6.902
2430	2.7	1	9.166
2550	2.9	1	9.612

[1]reduced data set, full data set in AppendixB.5.

[2]The number of alignments performed in a single test.

Table 4.5 shows the time consumption of CPU and GPGPU with the pruning mechanism with the sequence length of *16 bp*, which is generated using the code from Listing 4.4.

ms, *GPU with time-saving* version in 0.3 ms, and CPU version in *2.7 ms*.

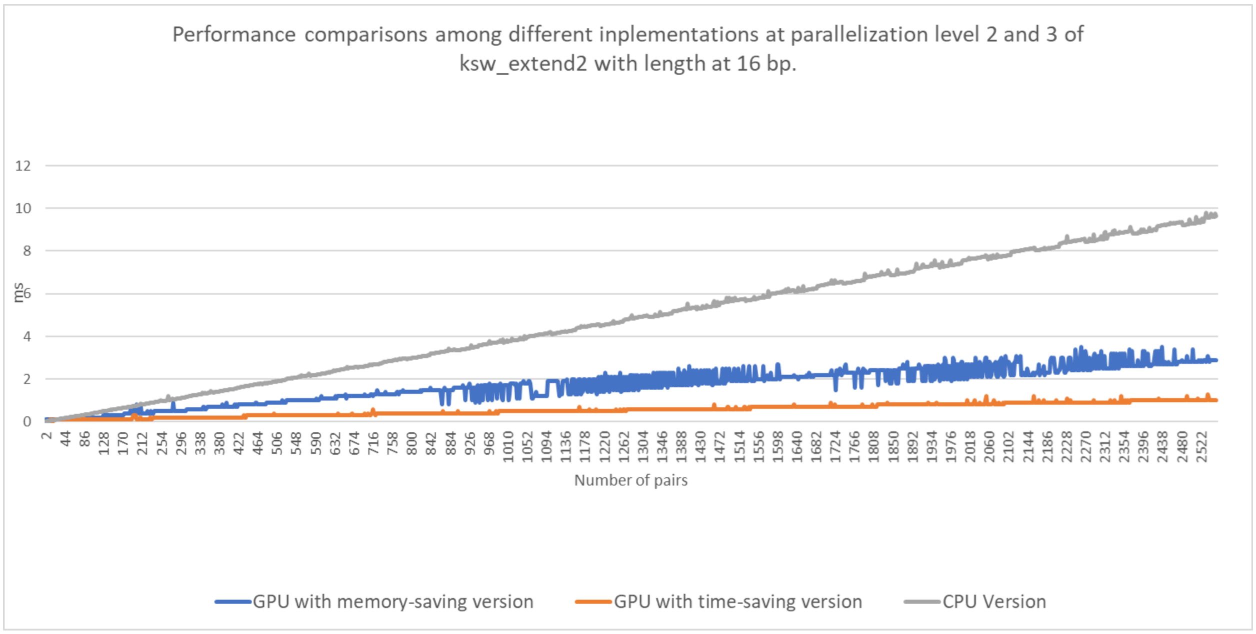


Figure 4.5 Performance comparisons among different implementations at parallelization *Level 2* and *Level 3* of *ksw_ extend2* with alignments length at *16 bp.* The *x-axis* represents the time consumption in *ms*, and the *y-axis* represents number of pairs from *2* to *2558*.

Figure 4.5 shows the test result for alignments at the length of *16 bp* among different versions. As shown in the figure, the time for the CPU version increased quickly with *30* alignments with *0.114 ms* at *30 bp* to *9.612 ms* with 2550 alignment, while both GPGPU versions of the code show very little increase with 30 alignments (reduced time *0.1 ms*, reduced memory *0.1 ms*) to *2550* alignments[10]. Like the previous test, the *GPU with time-saving* version shows excellent performance over the CPU version. However, it has much worse performance than the *GPU with time-saving* version. Therefore, the conclusion remains the same that the *GPU with time-saving* version is roughly nine times faster than the CPU version. The *GPU with memory-saving* version is roughly three times slower than the CPU version[11].

[10]Reduced time 1 ms, reduced memory 2.9 ms.

[11]With the number of alignments at *2550 GPU with memory-saving* version finished alignments in *2.9 ms*, *GPU with memory-saving* version in 1 ms, and CPU version in 9.612 ms.

4.3.4 Test 3: Performance Comparisons Among Different Implementations at Parallelization Level 2 and Level 3 of ksw_extend2 with Alignments length at 32 bp

After the first test and the second test with a sequence length of *8 bp* and *16 bp*, we perform another test with *32 bp* sequences.

Table 4.6 Performance comparisons among different implementations at parallelization *Level 2* and *Level 3* of *ksw_extend2* with alignments length at 32 bp[1].

Number of Alignments[2]	*GPU with memory-saving* Version(ms)	*GPU with time-saving* Version (ms)	CPU Version (ms)
30	0.3	0.1	0.426
150	1.2	0.3	2.046
270	1.4	0.5	3.692
390	2.3	0.7	5.325
510	3.1	0.9	7.62
630	3.3	1.3	8.724
750	3.5	1.2	10.21
810	4.1	1.4	11.18

[1]reduced data set, full data set in AppendixB.6.

[2]The number of alignments performed in a single test.

Table 4.6 describes the time consumption of CPU and GPGPU with the pruning mechanism with sequences length of 32 bp, which is generated using the code from Listing 4.4.

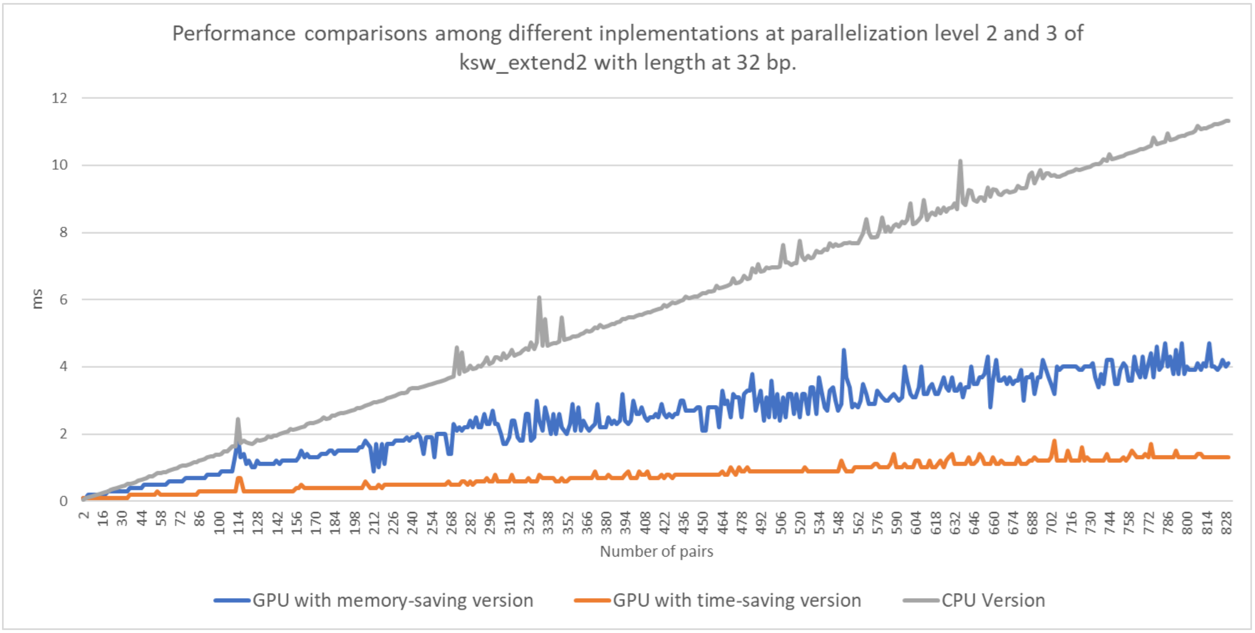


Figure 4.6 Performance comparisons among different implementations at parallelization *Level 2* and *Level 3* of *ksw_ extend2* with alignments length at *32 bp*. The y-axis represents the time consumption in *ms*, and the *x-axis* represents number of pairs from *2 to 832*.

Figure 4.6 shows the test result of alignments at the length of *32 bp* among different versions. As shown in the figure, the CPU version's time increased quickly with 30 alignments with *0.426 ms* at *30 bp* to *11.18 ms* with 819 alignments. In comparison, both GPGPU versions of the code show very little increase with 30 alignments[12] to 810 alignments[13]. Like the previous test, the *GPU with time-saving* version shows good performance over the CPU version. However, it has much worse performance than the *GPU with time-saving* version. Therefore, the conclusion remains the same that the *GPU with time-saving* version is roughly nine times faster than the CPU version. The *GPU with memory-saving* version is roughly three times slower than the CPU version[14].

[12]Reduced time *0.1 ms*, reduced memory *0.3 ms*.

[13]Reduced time *1.4 ms*, reduced memory *4.1 ms*.

[14]With the number of alignments *810*, *GPU with memory-saving* version finished alignments in *4.1 ms*, *GPU with memory-saving* version in *1.4 ms*, and CPU version in *11.18 ms*.

4.3.5 Summary

At this level, all tests was carried out on a *intel i7-6700k CPU*, *NVIDIA GeForce GTX 1080*, *32GB RAM, and 1TB solid state hard drive* desktop, which previously referred to as *PC 1.* In the test with real data[15], *BWA-MEM* has shown that on average, there are only few extensions at parallelization *Level 2*[16]. Therefore, before *Level 3* parallelization computation start, we need to collect as much fragment pairs as possible. During the test, it has shown that the length of the fragments are rarely longer than *32 bp* in seed extension. Therefore, tests at parallelization *Level 2* and *Level 3* are aimed to find out the performance of aligning multiple pairs at the same time. In the tests, there are two different GPGPU *ksw_ extend2* versions being created. GPGPU with *time-saving version* is aimed at reducing the time consumption, under the assumption that reducing time usage is more important than saving GPGPU memory. GPGPU with memory-saving version is aimed at reducing the GPGPU memory usage, under the assumption that GPGPU has a very limited number of RAM, and this is done through performing multiple data transfer between GPGPU and the main memory.

The test results remains the same for tests of alignment length at *8 bp, 16 bp and 32 bp.* and they have shown that CPU version's time consumption increases very fast. GPGPU with *memory-saving* version is roughly *3x* faster than the CPU version, and the GPGPU with *time-saving* version is roughly *3x* faster than the GPGPU with *memory-saving* version. The reason why GPGPU with memory-saving version is slower is because data transfer between GPGPU and main memory has a huge overhead.

[15]Section 4.1.
[16]At Chain Level.

4.4 GPGPU distributed BWA-MEM

As the new stable release version of Spark[17] recognizes GPGPU as a resource, it gives us an opportunity to build a Spark standalone GPGPU version of *BWA-MEM* on *SharcNET*. In this section, we will go through Big Data frameworks, Hadoop and Spark, and then presents the idea of Spark Standalone GPGPU version of *BWA-MEM* under parallelization Level 1 of ksw_extend2.

4.4.1 GPGPU Distributed Framework

GPGPU distributed version is similar to CPU distributed version as they both execute the program on multiple nodes in the cluster. *GPGPU distributed* can be implemented as one of the *Spark + CUDA*, *Hadoop + CUDA* and *OpenMPI + CUDA*. In comparison to other *GPGPU distributed* frameworks, Spark standalone GPGPU framework stands out for been version agnostic and easy to set-up.

4.4.2 Big Data

In data analytics, big data software are used to analyze and extract information from large data sets that are too sophisticated for the traditional data-processing applications. *Yahoo! Inc.* claimed they launched the most massive *Hadoop* production environment in *2008*, in which a *Linux cluster* with more than *10,000* cores was used for *Yahoo!*'s search query [78]. As of *2013*, more than half of the *Fortune 50* have claimed the use of *Hadoop* [79]. *Spark* is considered another popular big data software package. *Spark* and *Resilient Distributed Dataset* (RDD) were developed in *2012* in response to the *MapReduce* framework's limitations.

[17]Released on September 2, 2020.

4.4.2.1 Hadoop

Apache *Hadoop* contains multiple open-source software utilities, which facilitates a network of computers to resolve problems involving large amounts of data and computation resources. Using the *MapReduce* programming model and distributed storage, *Hadoop* can process large amounts of data and automatically handle hardware failures. Both *Hadoop* Distributed File System (HDFS) [80], and *MapReduce* are the core for Apache Hadoop, where HDFS is a file storage system, and *MapReduce* is a framework for processing.

Large files are first separated into large data blocks with HDFS, and *Hadoop* distributes these data blocks across the cluster. By allowing data to be stored directly in the same node where the execution happens, the dataset can be processed faster and more effectively than the conventional super-computer architecture, which mainly relies on high-speed networking for data transferring. HDFS is a distributed, scalable, and portable file system responsible for separating large files into multiple blocks. HDFS has five services: name node, secondary name node, job tracker, data node and task tracker.

4.4.2.2 Apache Spark

Apache Spark's Resilient Distributed Dataset (RDD) is considered a distributed multi-set of read-only data across multiple nodes with the feature of fault-tolerant. *MapReduce* framework has four stages: reading input data from disk, reducing the results, and storing the reduction results. Apache *Spark* depends on a cluster manager and a distributed storage system in the system, and its core provides distributed task dispatching, scheduling, and basic *I/O* functionalities with the RDD abstraction.

The popular cluster manager for *Spark* includes standalone, *Hadoop* YARN and Apache Mesos. For supported distributed storage, the most popular one is HDFS. As the most popular combination is *Hadoop* Yarn and Spark, *Spark* offers a pre-build Spark-Hadoop package. *Spark* also provides standalone mode, which uses Spark's built-in cluster manager, although it still requires a distributed storage system. Both *Hadoop* YARN and Apache Mesos offer the capability to include GPGPU resources. *Spark version 3.0.1*, which is published in *September 2020*, offers the capacity to include GPGPU resources in the standalone version. After successfully starting the master node, the worker is started with a GPGPU discovery code, which allows *Spark* to locate GPGPU resources using the configuration file. The program can be submitted using *spark-submit* with GPGPU resources scheduling configuration.

```
./spark-submit --class com.github.sparkbwa.SparkBWA --master spark://
    DESKTOP-A8LGS28.cogeco.local:7077 --driver-memory 2G --conf spark.
    executor.memory=4G --conf spark.executor.cores=1 --conf spark.task.cpus
    =1 --conf spark.task.resource.gpu.amount=0.3 --conf spark.executor.
    resource.gpu.amount=1 file:/opt/SparkBWA/target/SparkBWA-0.2.jar -m -r
    -p --index /Data/HumanBase/hg38 -n 32 -w "-R @RG\tID:foo\tLB:bar\tPL:
    illumina\tPU:illumina\tSM:ERR000589" file:/data/E1.FASTQ file:/data/E2.
    FASTQ file:/data/Output_ERR000589123b
```

Listing 4.5 Software Execution in *Spark* Standalone mode with GPGPU resources

Listing 4.5 describes the code for software execution in *Spark* Standalone mode with GPGPU resources, which requests a worker with one GPGPU to execute the program.

4.4.2.3 SparkBWA Analyzation

In *MapReduce*'s structure, *GPGPU distributed BWA-MEM* has three main stages: RDD creation, map, and reduction. A Resilient Distributed Dataset (RDD) is an immutable, partitioned collection of elements operated on in parallel. In the RDD creation, the sequence data is fed into the map phase, where the FASTQ file is used as input and stores into HDFS. Since HDFS is a distributed file system, the FASTQ file is separated into data blocks and distributed across the cluster. The FASTQ file is converted into JavaPairRDD, where they would appear as $< read\_id, read\_content >$. The read content contains all information corresponding to the sequence information, with $read\_id$ as the sequence identifier. However, *BWA-MEM* supports paired-end read, where two FASTQ files are used as input. In the case of the paired-end read method, the two tuples are joined using the key with the format of $< read\_id, Tuple < read\_content1, read\_content2 >>$. In this way, two reads from the same pair being stored onto a different node are avoided.

Once the RDD creation is completed, the mappers would call for *BWA-MEM* using the Java Native Interface (JNI), which would allow the combination of Java code and C/C++ code. The parallelized GPGPU *BWA-MEM* (C/C++ code) is used to process all the alignments parallel in the map stage. All outputs from execution nodes are combined in the reduce phase, and only one file is produced. As mentioned previously, the *GPGPU distributed BWA-MEM* is version agnostic towards the GPGPU version of parallelized GPGPU *BWA-MEM*. However, due to Open Sourced GPGPU *BWA-MEM*'s unavailability, we have to work on parallelizing *BWA-MEM*.

4.4.2.4 Summary

GPU distributed is considered the fastest framework among *CPU multi-threaded, CPU distributed, GPGPU multi-threaded, and GPGPU distributed.* However, the setup is the most expensive: set-up a cluster with minimum 3 GPGPU equipped roughly costs 3,000 dollars on commercial cluster.

Member of *Brock University* can take advantage of *SharcNET* without the support of Hadoop manager. Both Hadoop and Spark[18] needs Hadoop manager to run on *SharcNET*. However, the Spark stable release standalone version[19] changed this situation, where standalone version now supports GPGPU resource on the cluster.

4.4.3 Implementation

Apache *Spark* is the *de facto* standard framework for distributed scale-out data processing. With Spark, large amounts of data are analyzed quickly using a farm of servers to analyze data or generate business insights. Many data processing tasks are considered *embarrassingly parallel*, and it is very natural for GPGPU to be leveraged for *Spark* data processing queries.

[18]Before September 2, 2020.

[19]Released on September 2, 2020.

NVIDIA CUDA is a parallel computing architecture designed explicitly for NVIDIA GPGPU architecture, aiming to accelerate computational operations. The combination of *Spark* and NVIDIA CUDA accelerates data processing while substantially lowering the infrastructure costs.

The stable *3.0* version of Spark, which was published on *September 2nd of 2020*, now offers the capability to recognize GPUs as *first-class* resources along with CPU and system memory. The newest version allows *Spark 3.0* to directly place GPU-accelerated workloads on servers with GPGPU resources as GPGPU resources are required to complete a task. *Spark 3.0* offers GPGPU resources in *Spark standalone, YARN, and Kubernetes* clusters.

This thesis aims to combine *Spark* and CUDA parallel computing architecture with *BWA-MEM* to improve its performance. Due to the large amount of work involved in parallelizing *BWA* with CUDA parallel computing architecture, the parallelization *Level 3* of seed extension was not completed on the CPU side. The parallelization of *BWA-MEM*'s seed-extend function has three levels. In *Level 1*, function *ksw_ extend2* is parallelized, which performs seed extension on the single side of the seed. At *Level 2*, function *chain2aln* is parallelized, which performs seed extension on both side of the seeds within the same chain. in *Level 3*, function *mem_ align1_ core* is parallelized, which performs seed extension on all seeds in all chains. However, *BWA* has successfully combined with parallelization Level 1.

As mentioned previously, *Brock University* is a member of Compute Canada, which offers a giant amount of computation resources. The older version of *Spark* only offers NVIDIA CUDA as a resource with other resources managers. As Compute Canada has already installed a resource manager, installing another resource manager can cause conflict in resource management. The only option is to setup a cluster with *Spark* standalone mode, which does not require an additional resource manager.

As the newer stable release version of *Spark* can recognize GPGPU as a resource, the possibility of combining GPGPU versions of *BWA-MEM* and *Spark* was tested. As any third party does not offer a GPGPU version, the parallelized *Level 1* GPU *BWA-MEM* is used for this purpose. The Section here describes how they have been combined.

SparkBWA is a combination of *Spark* and BWA, which has provided a basis for Spark-GPU-BWA's current work. The benefit of SparkBWA is that it is version agnostic as the original *BWA-ALN* is not being touched. The original version of *BWA-MEM* is replaced with *BWA-MEM*-GPU as the input data format for *BWA-MEM* and *BWA-MEM*-GPU. After modification towards the makefile and Maven settings, Spark-GPU-BWA-MEM has been successfully compiled. Spark-GPU-BWA-MEM has successfully run in the standalone mode by setting up the newest versions of *Spark* on the system.

As *Spark* Standalone mode configuration for GPGPU resources is quite new, there are only few examples for Spark-GPU framework. The best guide that can be found online is RAPIDS, which is a software package that uses both *Spark* and GPU. After successful installation of *Spark* software, the discovery script's location is needed to be referenced by *Spark* configuration. By running the launching scripts, *Spark* can automatically discover GPGPU resources on the worker node.

```
spark-submit --class com.github.sparkbwa.SparkBWA --master spark://DESKTOP
    -A8LGS28.cogeco.local:7077 --driver-memory 1500m --executor-memory 6g
    --executor-cores 1 --verbose --num-executors 1 file:/home/cli/Documents
    /SparkBWA1/target/SparkBWA-0.2.jar -m -k -n 1 -s --index file:/home/cli
    /Documents/40p10000ltest/10000index.fa file:/home/cli/Documents/40
    p10000ltest/40pairs.FASTQ file:/home/cli/Documents/40p10000ltest
```

Listing 4.6 Spark-GPU Lunch Script

Listing 4.6 describes the launch script for launching Spark-GPU-BWA-MEM for a single read alignment. Resources have to be requested when submitting applications. For Spark-GPU applications, GPGPU resources have to be requested. BWA-MEM is used as a dynamic library in SparkBWA. The modification towards any newer version of *BWA-MEM* for SparkBWA is to add *-fPIC*. Therefore, an attempt to attach the flag *-fPIC* to the NVIDIA CUDA compiler is successful, and the Spark-GPU-BWA-MEM has produced the same result as *BWA-MEM*. The result of successfully combining three different technologies has proven that the idea of Spark-GPU-BWA-MEM can work.

4.5 Summary

Both *GPU with time-saving* and *GPU with memory-saving* versions outran the CPU version when multiple alignments were performed simultaneously. The *GPU with time-saving* version is roughly *3x* faster than the *GPU with memory-saving* version, and the *GPU with memory-saving* version is roughly *3x* faster than the CPU version.

There are limitations to the experiment. First, CUDA only allows time measurement done through a specific part of the CUDA code, which only allows us to measure the GPGPU part of the computation. Therefore, only the GPGPU part of the GPGPU version is computed. However, all parts of the CPU version are measured. Second, we did not implement the whole program in *Level 2* and three due to time constraints. As a result, at *Level 2* and *Level 3*, with correct implementation, the GPGPU version of *BWA-MEM* could outrun the CPU version of *BWA-MEM* in all testing cases. CUDA only allows time measurement through a specific part of the CUDA code, which means only the GPGPU part of the computation is measured. The host's code was not implemented in *Level 2* and *Level 3* due to time constraints. The testing data set was created using a random number generator. We also proofed that Spark-GPGPU-BWA-MEM is possible on *SharcNET*.

5 DISCUSSION

In this chapter, we will go through rationale and objective, results, theoretical running time, and future work.

5.1 Rationale and Objective

Developing a new algorithm from scratch within 2 years of the time frame is hard. As there are a lot of advances been made towards algorithms, developing one or improving one is hard. As newer framework and hardware come out every single day, improving an existing algorithm with new hardware or framework is easier. This thesis's main goal is to improve an existing algorithm with Big Data frameworks and GPGPU. We decided to accelerate *BWA* as it has been a popular alignment package since 2009, and there is already some research done. The early work for parallelizing *BWA* is mostly related to BWA-ALN[1]. The early parallelization work for *BWA-ALN* is pBWA [9]. Other versions include BarraCUDA [10], SparkBWA [16], BigBWA [15]. As *BWA-ALN* has both BarraCUDA[2] and SparkBWA[3], and other existing *GPGPU distributed* programs have already shown promising result, the *GPGPU distributed* version of *BWA-ALN* has a foreseeable performance improvement.

[1] *BWA-ALN* was developed with the first version of *BWA* package.

[2] The GPGPU version of BWA-ALN.

[3] The big data version of BWA-ALN.

BWA-MEM[4] was first developed in *2012*, and is now much popular than *BWA-ALN* when dealing with newer *NGS platforms*. Therefore, the decision was made to parallelize *BWA-MEM* instead of BWA-ALN. However, the structure of *GPGPU distributed* can still have a good performance improvement. As we do not have an existing open-source GPGPU version of *BWA-MEM*, the first step is to produce GPGPU-BWA-MEM[5]. After a hot spot analysis [21], we have determined that seed extension has a higher usage than other parts of the program. Due to time limitations, we have to focus on the part that would have the best chance to improve our program.

In *Level 2* and *Level 3* parallelization, we have *GPU with time-saving* and *GPU with memory-saving* versions. In parallelization *Level 2* and *Level 3* test, GPU with time-saving and *GPU with memory-saving* versions outran the CPU version as multiple alignments were performed simultaneously. The GPGPU with the time-saving version is roughly *9x* faster than the CPU version, and the GPGPU with the memory-saving version is roughly *3x* faster than the CPU version. The smith-Waterman algorithm is a famous approximate algorithm, and its operation is costly. To change this situation, *ksw_ extend2* implemented the pruning mechanism, which reduced the time complexity to a linear relation. On the other hand, no matter with or without the pruning mechanism, the GPGPU version of the *Smith-Waterman algorithm* is always linear, and there is no time saved at all.

In recent years, there has been a boom in big data because of the growth of social mobile, cloud, and multi-media computing. Understanding the data is far more interesting than the data itself, and is up to organizations to extract useful, actionable insights. However, the traditional system cannot store, process, and analyze massive amounts of unstructured data. That is where cloud computing and big data comes in. In compassion to other methods, big data programs are highly fault-tolerant and easy to setup. Both *Spark* and *Hadoop* are the big data Software packages, and they are commonly used to analyze large amounts of

[4]Another tool within the *BWA* package.

[5]The GPGPU *BWA-ALN* is BarraCUDA [10].

data. The successful compilation of the current work shows the *GPGPU distributed BWA-MEM* works with *Spark* Standalone mode, which indicates it is possible to run *Spark* on the Compute Canada-*SharcNET* Computing systems. Future work can focus on merging the GPGPU version of the seed extension with *BWA-MEM*.

As mentioned previously, *Hadoop* has a conflict with the existing scheduling system on SharcNET, and is not possible to setup *Hadoop* on Compute Canada. We still wanted to compare the differences between *Hadoop* and *Spark* for future references. The major difference between *Hadoop* and *Spark* is how the data is being stored on the hardware. For *I/O* intensive programs, *Hadoop* on the computer hard drive's *I/O* for its performance. *Spark* uses fast in-memory performance with reduced disk reading and writing operations. Sequence alignment programs are *I/O* intensive programs as both read and reference sequences are quite large. Storing them into RAM can help to reduce the time spent on *I/O*. *Hadoop* is a highly fault-tolerant system that replicates the data across the nodes and uses them in case of an issue. *Spark* keeps track RDD block creation process, and then it can rebuild a dataset when a partition fails. *Spark* standalone version supports built-in tools for resource allocation, scheduling and monitoring.

The benefit of running *BWA-MEM* on a *Spark* network is obvious. First, we can use the *Spark* network as a huge RAM storage. Another obvious reason to run on a *Spark* network is that we want to use a distributed network. The use of *Spark* with the GPGPU version of *BWA* allows the GPGPU version of *BWA* to be updated. It is perfect as project can easily be separated into two. As long as *MapReduce*'s output can be used as input for the GPGPU version of the *BWA-MEM*.

5.2 Result

In *BWA-MEM*, we first collect as many extension alignments as possible before the GPGPU version of ksw_extend2's computation. All the alignments then resume afterwards. To collect as many alignments as possible, the modification of the existing code structure is mandatory. After modification, multiple alignments can be computed simultaneously. the parallelization level is low at ksw_extend2 parallelization *Level 1* and *Level 2*. This is because the parallelization is only done for a single alignment at parallelization Level 1, where no more than $length_{query} + length_{sequence}$ cells can be calculated at the same time. At *Level 2* parallelization, on average, there are two to three alignments been performed, which does not provide enough room for parallelization. To have a significant speedup, the parallelization should be higher than *Level 3*. The higher level of parallelization involves deeper GPGPU parallelization in the *BWA-MEM*. The stable *3.0* version of Spark published on *September* *2nd of 2020* offers the capability to recognize GPUs as first-class resources along with CPU and system memory. This means that we will be able to use Compute Canada's computation resources. We can alter SparkBWA's library to form the Spark-GPU-BWA-MEM.

Two different versions of GPU *ksw_extend2* were tested, where one is aimed at reducing GPGPU memory, and the other is aimed to reduce the time usage. At the start of this thesis, GPU's assumption with memory-saving version may not affect the execution time. However, to reduce memory use, extra work is needed for transferring data in and out of the GPU, which is the most time-consuming part of the GPGPU program even though the smaller memory version is still roughly three times faster than the CPU version. The GPGPU version of *ksw_extend2* with time-saving usages performs quite well, roughly nine times faster than the CPU version of *ksw_extend2*. The GPGPU reduced memory version of *ksw_extend2* has also shown an excellent performance versus the CPU version, which is three times faster. Since reducing the time is far more important than reducing the memory usage, the GPGPU reduced time version is considered a better option than the GPGPU reduced memory version. It was expected that the GPGPU version of *ksw_extend2* would outperform the CPU version of *ksw_extend2* when processing multiple alignments at the same time. However, it does not seem to be the case when it comes to a single alignment. During the parallel *Level 1* testing, the GPGPU version could not outperform the CPU version in all testing cases, especially when the pruning mechanism was used. Therefore, the performance improvement is gained when the number of alignments increases, which seems to be independent of the alignment length.

C limits how much memory can be *malloc*ed, which has a max size of 16 Megabytes. Therefore, if people want to use more memory, they have to create more than one pointer for arrays to store more than 16 Megabytes of information. Because of the time limitations, the only implemented part is the GPGPU version of *ksw_ extend2*, which means further work is needed to convert the CPU part of the code to fit what we need.

5.3 Theoretical Running Time

For a typical Smith-Waterman algorithm showed in Algorithm 3, assuming that the two strings S_0, and S_1, have the length of $|S_0| = L_0$, and $|S_1| = L_1$, which means the dimension of solving matrix is $L_0 \cdot L_1$. As Algorithm 3 computes one cell at a time, the time complexity of a typical Smith-Waterman algorithm is $O(L_0 \cdot L_1)$. Since the Smith-Waterman algorithm needs to record the scoring matrix, the space complexity is $O(L_0 \cdot L_1)$.

For *ksw_ extend2* algorithm shown in Algorithm 8 and Algorithm 7, the initialization initializes the first column and first row of the scoring matrix, which takes $O(L_0 + L_1)$. In the case of without the pruning mechanism, each cell has to be computed separately, which is similar to Algorithm 3, where the time complexity is $O(L_0 \cdot L_1)$. However, as the *ksw_ extend2* was implemented with dynamic programming, only previously computed H matrix's row (Figure 3.8), E matrix's column (Figure 3.10), and F matrix's row (Figure 3.11) are kept. However, the whole S matrix is pre-calculated. Therefore, it at least has a space complexity of $O(L_0 \cdot L_1)$.

When Algorithm 9 is applied to *ksw_ extend2* for pruning mechanism, only the cells between *beg* and *end* are computed, which has a total length of $2w + 1$. The time complexity has been reduced to $O(wL_1)$. If L_1 and L_0 are both equal to n, the time complexity has been reduced from $O(n^2)$ to $O(wn)$. When parallelized algorithm is applied, the number of steps taken is $L_0 + L_1$ (as shown in Algorithm 6), which has a time complexity of $O(n)$.

When k alignments are being computed together, the time complexity of parallelized *ksw_ extend2* remains at $O(n)$ (under the case that GPGPU has enough computation power). However, the CPU version of *ksw_ extend2* has a time complexity of $O(kwn)$. GPGPU parallelized code would spend more time transferring data when the number of alignments increased, which means the time complexity would not remain at $O(n)$, but it would be much lesser than $O(kw)$. As shown in Figure 4.6, clearly both CPU and GPGPU versions have a linear time complexity, which proves the computed time complexity being correct. Also, time consumption for CPU the version increased much faster than the GPGPU version (observed from Figure 4.4, Figure 4.5, and Figure 4.6).

5.4 Future Work

The thesis objective is to work towards a *GPGPU distributed BWA-MEM*. The program has two main components, which are GPGPU *BWA-MEM* and big data. At the beginning of the thesis, we find out no open-source version of *BWA-MEM* is available. Therefore the first step is to modify *BWA-MEM*'s code for GPGPU programming. Because of the amount of work involved in parallelizing the *BWA-MEM*, we only finished parallelizing the *BWA-MEM* seed extension part. Also, we did not integrate the GPGPU version of the seed extension into the *BWA-MEM*. However, we successfully combined *Spark* with *BWA-MEM* and the GPGPU version of the seed extension at parallelization *Level 1*. It does not make sense to compare the *GPGPU distributed BWA-MEM* as the current parallelization level is low.

To have a better understanding of how seed-extension works in *BWA-MEM*, we have tested the seed-extension program using the data mentioned in section 4.1. *BWA-MEM* works by finding seeds in each read, chaining the close seeds into chains, and then performs seed-extension on each chain. These seeds are short fragments from the read that exactly matches the reference sequence. The extension is performed on both ends of the seed, and the approximate string matching is done through the Smith-Waterman algorithm. The maximum extension length is smaller than the length of the corresponding exact match.

We ran *BWA-MEM* with these data step by step to learn exactly how *ksw_ extend2* works, especially its input and output. Basically, *ksw_ extend2* takes in the exact matching score and a pair of possible extension sequence fragments and returns a valid extension result. As both ends of the seed are extended using the same function, the left side extension is reversed before extension.

These sequence fragments, in each pair, all have a certain similarity. *ksw_ extend2* also uses a pruning mechanism to reduce time complexity. Therefore, the parallelization *Level 1* test was designed to determine the relations between similarity and sequence length and the pruning mechanism's effect. The result has shown that the time consumption for both CPU and GPGPU versions shows a positive correlation with the length of the alignments under the case without the pruning mechanism. The GPGPU version without the pruning mechanism is not affected by the level of sequence similarity. However, with the CPU version, the time consumption further increases with the sequence similarity level. The same thing applies to both versions with the pruning mechanism.

In parallelization *Level 1*, we have already tested everything up to *7416 bp* length. However, in real situation, the length of the extension rarely goes over *32 bp*. Therefore, in Parallelization *Level 2* test, we only test up to *32 bp*. The test result has also shown that the pruning has no effect on the GPU's performance but dramatically improves the CPU's performance. Under the pruning mechanism, both program versions have a linear relation with time consumption and length. However, the GPGPU version can never beat the CPU version in this case. The main reason behind the program is caused by not utilizing all the GPGPU resources. As we know, Smith-Waterman's parallelization is done through the Wave-Front method and the current anti-diagonal relays on the previous anti-diagonal in the matrix. However, for the first anti-diagonal, there is only one cell computed. So why not with start

10, or 100, or even thousands of extensions at the same time to reduce the time consumption? This is why we continued to *Level 2* and *Level 3* parallelization for *ksw_ extend2*.

6 APPENDIX EXPLANATION

The modified version of *ksw_ extend2* is provided from Page 149 to Page 186. Moreover, a detailed version of the test output is produced from Page 138 to Page 149.

REFERENCES

[1] F. S. Collins, M. Morgan, and A. Patrinos, “The human genome project: Lessons from large-scale biology,” *Science*, vol. 300, no. 5617, pp. 286–290, 2003.

[2] S. Marsh, “Pyrosequencing,” in *Molecular Diagnostics*, Elsevier, 2010, pp. 107–116.

[3] R. N. Bharagava, D. Purchase, G. Saxena, and S. I. Mulla, “Applications of metagenomics in microbial bioremediation of pollutants: From genomics to environmental cleanup,” in *Microbial diversity in the genomic era*, Elsevier, 2019, pp. 459–477.

[4] A. K. Gupta and U. Gupta, “Next generation sequencing and its applications,” in *Animal Biotechnology*, Elsevier, 2014, pp. 345–367.

[5] S. Schbath, V. é. r. Martin, M. Zytnicki, J. Fayolle, V. Loux, and J.-F. ç. o. Gibrat, “Mapping reads on a genomic sequence: An algorithmic overview and a practical comparative analysis,” *Journal of Computational Biology*, vol. 19, no. 6, pp. 796–813, 2012.

[6] 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.

[7] N. E. Morton, “Parameters of the human genome,” *Proceedings of the National Academy of Sciences*, vol. 88, no. 17, pp. 7474–7476, 1991.

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

[9] D. Peters, X. Luo, K. Qiu, and P. Liang, “Speeding up large-scale next generation sequencing data analysis with pbwa.,” *Journal of Applied Bioinformatics & Computational Biology*, vol. 6, p. 2, 2012.

[10] W. B. Langdon and B. Y. H. Lam, “Genetically improved barracuda,” *BioData Mining*, vol. 10, no. 1, p. 28, 2017.

[11] C.-M. Liu, T. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, *et al.*, “Soap3: Ultra-fast gpu-based parallel alignment tool for short reads,” *Bioinformatics*, vol. 28, no. 6, pp. 878–879, 2012.

[12] 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, 2013.

[13] 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, 2012.

[14] I. Merelli, H. P é rez-S á nchez, S. Gesing, and D. D'Agostino, *High-performance computing and big data in omics-based medicine*, 2014. DOI: 10.1155/2014/825649.

[15] J. é. M. Abu ı n, J. C. Pichel, T. á. s. F. Pena, and J. Amigo, "Bigbwa: Approaching the burrows–wheeler aligner to big data technologies," *Bioinformatics*, vol. 31, no. 24, pp. 4003–4005, 2015.

[16] ——, "Sparkbwa: Speeding up the alignment of high-throughput dna sequencing data," *PloS one*, vol. 11, no. 5, e0155461, 2016.

[17] A. Amir, "Implementation of bio-informatics applications on various gpu platforms," 2013.

[18] K. Deb and K. Deb, "Multi-objective Optimization," in *Search Methodologies : Introductory Tutorials in Optimization and Decision Support Techniques*, Boston, MA: Springer US, 2014, pp. 403–449, ISBN: 978-1-4614-6940-7. DOI: 10.1007/978-1-4614-6940-7_15.

[19] N. Khare, A. Khare, and F. Khan, "Hcudablast: An implementation of blast on hadoop and cuda," *Journal of Big Data*, vol. 4, no. 1, pp. 1–8, 2017.

[20] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with bwa-mem," *arXiv preprint arXiv:1303.3997*, 2013.

[21] L. Hai, J. Liu, Z. Hai, Y. Liu, J. Yang, and Z. Liu, "Analysis and accelerating of bwa sequence comparison algorithm based on GPU," *China Academic Journal Electronic Publishing House*, vol. 1009-2552, no. 3, pp. 67–78, 2018. DOI: 10.13274/j.cnki.hdzj.2018.03.014.

[22] J. M. Heather and B. Chain, "The sequence of sequencers: The history of sequencing dna," *Genomics*, vol. 107, no. 1, pp. 1–8, 2016.

[23] F. Sanger and A. R. Coulson, "A rapid method for determining sequences in dna by primed synthesis with dna polymerase," *Journal of molecular biology*, vol. 94, no. 3, pp. 441–448, 1975.

[24] A. Gomes and B. Korf, "Chapter 5-genetic testing techniques," *Pediatric Cancer Genetics. Amsterdam: Elsevier*, pp. 47–64, 2018.

[25] J. Wages Jr, "Polymerase chain reaction," *Encyclopedia of Analytical Science*, p. 243, 2005.

[26] B. M. Paegel, R. G. Blazej, and R. A. Mathies, "Microfluidic devices for dna sequencing: Sample preparation and electrophoretic analysis," *Current opinion in biotechnology*, vol. 14, no. 1, pp. 42–50, 2003.

[27] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice, "The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants," *Nucleic acids research*, vol. 38, no. 6, pp. 1767–1771, 2010.

[28] J. Eid, A. Fehr, J. Gray, K. Luong, J. Lyle, G. Otto, P. Peluso, D. Rank, P. Baybayan, B. Bettman, *et al.*, "Real-time dna sequencing from single polymerase molecules," *Science*, vol. 323, no. 5910, pp. 133–138, 2009.

[29] M. J. Levene, J. Korlach, S. W. Turner, M. Foquet, H. G. Craighead, and W. W. Webb, "Zero-mode waveguides for single-molecule analysis at high concentrations," *science*, vol. 299, no. 5607, pp. 682–686, 2003.

[30] R. Bowden, R. W. Davies, A. Heger, A. T. Pagnamenta, M. de Cesare, L. E. Oikkonen, D. Parkes, C. Freeman, F. Dhalla, S. Y. Patel, *et al.*, "Sequencing of human genomes with nanopore technology," *Nature communications*, vol. 10, no. 1, pp. 1–9, 2019.

[31] B. Ma, K. Zhang, C. Hendrie, C. Liang, M. Li, A. Doherty-Kirby, and G. Lajoie, "Peaks: Powerful software for peptide de novo sequencing by tandem mass spectrometry," *Rapid communications in mass spectrometry*, vol. 17, no. 20, pp. 2337–2342, 2003.

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

[33] T. K. Vintsyuk, "Speech discrimination by dynamic programming," *Cybernetics*, vol. 4, no. 1, pp. 52–57, 1968.

[34] 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.

[35] D. Sankoff, "Matching sequences under deletion/insertion constraints," *Proceedings of the National Academy of Sciences*, vol. 69, no. 1, pp. 4–6, 1972.

[36] P. H. Sellers, "On the theory and computation of evolutionary distances," *SIAM Journal on Applied Mathematics*, vol. 26, no. 4, pp. 787–793, 1974.

[37] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM (JACM)*, vol. 21, no. 1, pp. 168–173, 1974.

[38] R. A. Wagner and R. Lowrance, "An extension of the string-to-string correction problem," *Journal of the ACM (JACM)*, vol. 22, no. 2, pp. 177–183, 1975.

[39] J. R. Ullmann, “A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words,” *The Computer Journal*, vol. 20, no. 2, pp. 141–147, 1977.

[40] D. Sankoff and J. Kruskal, *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison (1983).* Addison-Wesley, Reading, MA, 1983.

[41] K. Kukich, “Techniques for automatically correcting words in text,” *Acm Computing Surveys (CSUR)*, vol. 24, no. 4, pp. 377–439, 1992.

[42] P. H. Sellers, “The theory and computation of evolutionary distances: Pattern recognition,” *Journal of algorithms*, vol. 1, no. 4, pp. 359–373, 1980.

[43] G. Navarro, “A guided tour to approximate string matching,” *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.

[44] R. M. Karp and M. O. Rabin, “Efficient randomized pattern-matching algorithms,” *IBM journal of research and development*, vol. 31, no. 2, pp. 249–260, 1987.

[45] R. Baeza-Yates, *Efficient text searching.* University of Waterloo, 1989.

[46] R. Baeza-Yates, “Some new results on approximate string matching,” in *Workshop on Data Structures*, 1991.

[47] R. A. Baeza-Yates, “Text-retrieval: Theory and practice.,” in *IFIP Congress (1)*, vol. 12, 1992, pp. 465–476.

[48] S. Wu and U. Manber, “Fast text searching: Allowing errors,” *Communications of the ACM*, vol. 35, no. 10, pp. 83–91, 1992.

[49] R. Baeza-Yates, “A unified view to string matching algorithms,” in *International Conference on Current Trends in Theory and Practice of Computer Science*, Springer, 1996, pp. 1–15.

[50] R. Baeza-Yates and G. Navarro, “Faster approximate string matching,” *Algorithmica*, vol. 23, no. 2, pp. 127–158, 1999.

[51] A. H. Wright, “Approximate string matching using withinword parallelism,” *Software: Practice and Experience*, vol. 24, no. 4, pp. 337–362, 1994.

[52] G. Myers, “A fast bit-vector algorithm for approximate string matching based on dynamic programming,” *Journal of the ACM (JACM)*, vol. 46, no. 3, pp. 395–415, 1999.

[53] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio, “Indexing methods for approximate string matching,” en, *IEEE Data Engineering Bulletin*, vol. 24, no. 4, pp. 19–27, 2001.

[54] E. Ukkonen, “Finding approximate patterns in strings,” en, *Journal of Algorithms*, vol. 6, pp. 132–137, 1985, ISSN: 01966774. DOI: 10.1016/0196-6774(85)90023-9.

[55] G. Navarro and R. Baeza-Yates, "A hybrid indexing method for approximate string matching," en, *Journal of Discrete Algorithms*, vol. 1, no. 1, pp. 205–239, 2000.

[56] N. Ahmed, K. Bertels, and Z. Al-Ars, "A comparison of seed-and-extend techniques in modern dna read alignment algorithms," in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, IEEE, 2016, pp. 1421–1428.

[57] 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.

[58] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.-T. Chen, and J. Seiferas, "The smallest automation recognizing the subwords of a text," *Theoretical computer science*, vol. 40, pp. 31–55, 1985.

[59] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," 1994.

[60] W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu, "A space and time efficient algorithm for constructing compressed suffix arrays," *Algorithmica*, vol. 48, no. 1, pp. 23–36, 2007.

[61] T. W. Lam, R. Li, A. Tam, S. Wong, E. Wu, and S.-M. Yiu, "High throughput short read alignment via bi-directional bwt," in *2009 IEEE International Conference on Bioinformatics and Biomedicine*, IEEE, 2009, pp. 31–36.

[62] Z.-G. Wei, S.-W. Zhang, and F. Liu, "Smsmap: Mapping single molecule sequencing reads by locating the alignment starting positions," *BMC bioinformatics*, vol. 21, no. 1, pp. 1–15, 2020.

[63] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[64] M. J. Flynn and K. W. Rudd, "Parallel architectures," *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, pp. 67–70, 1996.

[65] T. Ungerer, B. Robi č, and J. Š ilc, "Multithreaded processors," *The Computer Journal*, vol. 45, no. 3, pp. 320–348, 2002.

[66] T. Sterling, M. Anderson, and M. Brodowicz, "Chapter 8 - the essential mpi," in *High Performance Computing*, Morgan Kaufmann, 2018, pp. 249–284, ISBN: 978-0-12-420158-3.

[67] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[68] "Cuda memory," in *CUDA Application Design and Development*, Morgan Kaufmann, 2011, pp. 109–131, ISBN: 978-0-12-388426-8. DOI: 10.1016/B978-0-12-388426-8.00005-7.

[69] C.-L. Hung and G.-J. Hua, “Local alignment tool based on hadoop framework and gpu architecture,” *BioMed research international*, vol. 2014, 2014.

[70] S. Mehrotra and A. Grade, *Apache Spark Quick Start Guide: Quickly learn the art of writing efficient big data applications with Apache Spark*. Packt Publishing Ltd, 2019.

[71] S. Mccarthy, *Through the university of western ontario, researchers are using high-tech bait - a powerful new computing network - to reel in the "great whites" of the research world*, 2005. [Online]. Available: https://www.innovation.ca/story/sharc-bait (visited on 11/27/2020).

[72] *Sharcnet: Cluster copper.sharcnet.ca*. [Online]. Available: https://www.sharcnet.ca/my/systems/show/108.

[73] *Sharcnet: Visualization vdi-centos6*. [Online]. Available: https://www.sharcnet.ca/my/systems/show/104 (visited on 08/12/2020).

[74] *Sharcnet: Cluster mosaic.sharcnet.ca*. [Online]. Available: https://www.sharcnet.ca/my/systems/show/106 (visited on 08/12/2020).

[75] *Sharcnet: Cluster graham.sharcnet.ca*. [Online]. Available: https://www.sharcnet.ca/my/systems/show/114 (visited on 08/12/2020).

[76] 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.

[77] S. Warris, F. Yalcin, K. J. Jackson, and J. P. Nap, “Flexible, fast and accurate sequence alignment profiling on gpgpu with paswas,” *PloS one*, vol. 10, no. 4, e0122524, 2015.

[78] C. Metz, *How yahoo spawned hadoop, the future of big data*, 2011.

[79] A. Inc, *Altior’s altrastar - hadoop storage accelerator and optimizer now certified on CDH 4 (cloudera’s distribution including apache hadoop version 4)*. [Online]. Available: https://www.prnewswire.com/news-releases/altiors-altrastar---hadoop-storage-accelerator-and-optimizer-now-certified-on-cdh4-clouderas-distribution-including-apache-hadoop-version-4-183906141.html (visited on 08/13/2020).

[80] *Hdfs architecture guide*. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html (visited on 08/12/2020).

APPENDIX

A Appendix Explanation

The modified version of ksw_extend2 is provided from Page 149 to Page 186. And a detailed version of the test output is produced from Page 138 to Page 149.

B List of Test Output

B.1 Parallelization Level 1 Test 1

Table B.1 CPU performance in alignment time without pruning mechanism Using different similarities at Level 1.

Length (bp)	**CPU 90%**	**GPU 90%**	**CPU 92%**	**GPU 92%**	**CPU 94%**	**GPU 94%**	**CPU 96%**	**GPU 96%**	**CPU 98%**	**GPU 98%**
16	0	0.1	0	0	0	0	0.01	0	0.01	0
32	0.01	0.1	0.01	0.1	0.01	0.1	0.01	0.1	0.01	0.1
48	0.03	0.1	0.03	0.1	0.03	0.1	0.03	0.1	0.03	0.1
64	0.05	0.2	0.05	0.2	0.05	0.2	0.04	0.2	0.04	0.2
80	0.08	0.2	0.07	0.2	0.07	0.2	0.08	0.2	0.08	0.2
96	0.14	0.3	0.11	0.3	0.11	0.3	0.1	0.3	0.1	0.3
112	0.17	0.4	0.15	0.3	0.14	0.3	0.15	0.3	0.15	0.3
128	0.27	0.3	0.18	0.3	0.19	0.3	0.18	0.3	0.18	0.3
144	0.22	0.4	0.23	0.4	0.22	0.4	0.22	0.4	0.22	0.4
160	0.27	0.4	0.27	0.4	0.27	0.4	0.27	0.4	0.26	0.4
176	0.31	0.5	0.31	0.5	0.31	0.5	0.31	0.5	0.32	0.5
192	0.36	0.5	0.35	0.5	0.35	0.5	0.35	0.5	0.36	0.8
208	0.4	0.5	0.4	0.5	0.39	0.5	0.4	0.5	0.41	0.9
224	0.45	0.6	0.45	0.6	0.45	0.6	0.46	0.6	0.48	0.6
240	0.5	0.6	0.51	0.6	0.49	0.7	0.5	0.6	0.52	0.6
256	0.56	0.7	0.56	0.7	0.54	0.7	0.56	0.7	0.59	0.7
272	0.63	0.7	0.61	0.7	0.6	0.7	0.62	0.7	0.64	0.7
288	0.67	0.7	0.67	0.7	0.66	0.7	0.68	0.7	0.71	0.7
304	0.74	0.8	0.73	0.8	0.73	0.8	0.74	0.8	0.78	0.8
320	0.81	0.8	0.81	0.8	0.8	0.8	0.81	1.3	0.85	0.8
336	0.87	0.9	0.86	0.9	0.86	0.9	0.88	0.9	0.92	1.3
352	0.94	0.9	0.93	1.4	0.93	0.9	0.95	0.9	1.01	0.9
368	1.01	1	1.01	1	1	1	1.03	1	1.08	1
384	1.08	1	1.1	1	1.08	1	1.12	1	1.17	1
400	1.16	1	1.16	1	1.17	1	1.19	1	1.25	1
416	1.24	1.1	1.23	1.1	1.24	1.1	1.27	1.1	1.34	1.1
432	1.32	1.1	1.32	1.1	1.33	1.1	1.35	1.1	1.43	1.1
448	1.4	1.2	1.39	1.2	1.42	1.2	1.44	1.2	1.54	1.2

Continued on next page

Table B.1 – continued from previous page

Length (bp)	**CPU 90%**	**GPU 90%**	**CPU 92%**	**GPU 92%**	**CPU 94%**	**GPU 94%**	**CPU 96%**	**GPU 96%**	**CPU 98%**	**GPU 98%**
464	1.48	1.2	1.48	1.2	1.51	1.7	1.55	1.2	1.63	1.6
480	1.56	1.2	1.56	1.2	1.6	1.7	1.62	1.2	1.73	1.2
496	1.66	1.3	1.65	1.3	1.7	1.3	1.75	1.3	1.82	1.3
512	1.73	1.3	1.74	1.3	1.79	1.3	1.82	1.3	1.93	1.3
528	1.82	1.4	1.83	1.4	1.9	1.8	1.93	1.4	2.04	1.4
544	1.92	1.4	1.93	1.9	2.02	1.4	2.03	1.4	2.15	1.4
560	2.02	1.5	2.02	1.4	2.1	1.9	2.14	1.4	2.27	1.4
576	2.12	1.5	2.14	1.5	2.21	1.5	2.25	1.5	2.38	1.5
592	2.23	1.5	2.21	2	2.32	1.5	2.35	1.5	2.52	1.5
608	2.33	1.6	2.32	1.6	2.45	2	2.47	2	2.62	1.6
624	2.42	1.6	2.43	1.6	2.54	1.6	2.59	1.6	2.75	1.6
640	2.54	1.6	2.54	1.6	2.66	1.6	2.71	2.1	2.87	1.7
656	2.65	1.7	2.65	1.7	2.78	1.7	2.83	1.7	3.01	1.7
672	2.78	1.7	2.77	2.2	2.94	1.7	2.96	1.7	3.15	1.7
688	2.89	1.8	2.89	1.8	3.03	2.2	3.09	1.8	3.28	1.8
704	3.02	1.8	3.01	1.8	3.17	1.8	3.22	1.8	3.42	1.8
720	3.13	1.9	3.13	2.3	3.29	1.9	3.35	1.9	3.58	1.9
736	3.24	2.4	3.26	1.9	3.43	2.4	3.49	1.9	3.71	1.9
752	3.36	1.9	3.38	1.9	3.6	1.9	3.63	2	3.85	1.9
768	3.49	2	3.51	2	3.75	2.6	3.77	2.4	4	2
784	3.63	2.5	3.65	2	3.88	2.6	3.92	2.5	4.16	2
800	3.74	2.5	3.78	2.1	4.04	2.5	4.07	2.5	4.34	2.1
816	3.88	2.6	3.92	2.1	4.19	2.2	4.22	2.1	4.46	2.1
832	4.02	2.2	4.05	2.2	4.53	2.2	4.37	2.2	4.62	2.2
848	4.15	2.2	4.2	2.2	4.42	2.6	4.52	2.2	4.82	2.2
864	4.29	2.2	4.35	2.7	4.58	2.7	4.68	2.2	4.94	2.2
880	4.44	2.3	4.5	2.3	4.73	2.3	4.85	2.3	5.11	2.3
896	4.58	2.3	4.64	2.8	4.89	2.3	5.01	2.3	5.28	2.3
912	4.73	2.3	4.8	2.4	5.09	2.3	5.18	2.4	5.46	2.4
928	4.88	2.4	4.95	2.4	5.22	2.4	5.34	2.4	5.63	2.4
944	5.02	2.4	5.11	2.4	5.42	2.4	5.51	2.4	5.81	2.4
960	5.21	2.5	5.26	2.5	5.55	2.5	5.69	2.5	5.99	2.6
976	5.35	3	5.43	2.5	5.75	3	5.86	2.5	6.18	2.5
992	5.49	2.6	5.63	2.6	5.89	2.6	6.05	2.6	6.41	3.2

B.2 Parallelization Level 1 Test 2

Table B.2 Alignment performance test comparison with or without The pruning mechanism at Level 1 up to 992 bp Length.

length (bp)	CPU with the pruning mechanism (ms)	CPU without the pruning mechanism (ms)	GPU without the pruning mechanism (ms)	GPU with the pruning mechanism(ms)
16	0	0.01	0.1	0.1
32	0.01	0.01	0.1	0.1
48	0.03	0.03	0.1	0.1
64	0.05	0.04	0.2	0.2
80	0.07	0.08	0.2	0.2
96	0.11	0.1	0.3	0.3
112	0.14	0.15	0.3	0.3
128	0.18	0.18	0.4	0.4
144	0.22	0.22	0.4	0.4
160	0.26	0.26	0.4	0.4
176	0.29	0.32	0.5	0.5
192	0.33	0.36	0.5	0.5
208	0.35	0.41	0.6	1.1
224	0.4	0.48	0.6	0.6
240	0.42	0.52	0.6	0.6
256	0.46	0.59	0.7	0.7
272	0.5	0.64	0.7	0.7
288	0.54	0.71	0.8	0.8
304	0.57	0.78	0.8	0.8
320	0.61	0.85	0.8	0.8
336	0.65	0.92	0.9	0.9
352	0.69	1.01	0.9	0.9
368	0.72	1.08	1	1
384	0.75	1.17	1	1
400	0.79	1.25	1	1
416	0.82	1.34	1.5	1.1
432	0.86	1.43	1.1	1.1
448	0.89	1.54	1.2	1.2
464	0.94	1.63	1.2	1.2
480	0.97	1.73	1.7	1.3
496	1	1.82	1.3	1.3
512	1.04	1.93	1.3	1.3
528	1.08	2.04	1.4	1.8
544	1.11	2.15	1.4	1.4
560	1.15	2.27	1.5	1.5
576	1.18	2.38	1.9	1.5
592	1.21	2.52	1.5	1.5
608	1.25	2.62	1.6	1.6

Continued on next page

Table B.2 – continued from previous page

length (bp)	**CPU with the pruning mechanism (ms)**	**CPU without the pruning mechanism (ms)**	**GPU without the pruning mechanism (ms)**	**GPU with the pruning mechanism(ms)**
624	1.29	2.75	1.6	1.6
640	1.33	2.87	1.7	2.1
656	1.36	3.01	1.7	1.7
672	1.4	3.15	1.8	1.8
688	1.45	3.28	1.8	1.8
704	1.48	3.42	1.8	1.8
720	1.51	3.58	1.9	1.9
736	1.54	3.71	1.9	1.9
752	1.59	3.85	2	2
768	1.62	4	2	2
784	1.65	4.16	2	2
800	1.69	4.34	2.1	2.1
816	1.72	4.46	2.1	2.1
832	1.77	4.62	2.2	2.2
848	1.8	4.82	2.2	2.2
864	1.83	4.94	2.3	2.3
880	1.87	5.11	2.3	2.3
896	1.9	5.28	2.3	2.3
912	1.94	5.46	2.4	2.4
928	1.97	5.63	2.4	2.4
944	2.01	5.81	2.5	2.5
960	2.05	5.99	2.5	2.5
976	2.09	6.18	3	3
992	2.11	6.41	2.6	2.6

B.3 Parallelization Level 1 Test 3

Table B.3 Alignment performance test comparison with or without The pruning mechanism at Level 1 up to 7416 bp Length.

Length (bp)	**CPU without pruning mechanism (ms)**	**CPU With the pruning mechanism (ms)**	**GPU Without the pruning mechanism (ms)**
16	0	0.01	0.1
116	0.2	0.15	0.3
216	0.4	0.37	1
316	0.9	0.6	0.8
416	1.3	0.83	1.1
516	1.9	1.05	1.3
616	2.7	1.27	1.6
716	3.5	1.5	1.9
816	4.5	1.72	2.1
916	5.5	1.95	2.4
1016	6.7	2.17	2.7
1116	7.9	2.4	2.9
1216	9.2	2.63	3.2
1316	10.7	2.88	4
1416	12.3	3.08	3.8
1516	14	3.31	4.5
1616	15.8	3.52	4.8
1716	17.7	3.77	4.7
1816	19.7	3.98	4.9
1916	21.8	4.21	5.2
2016	24	4.42	5.5
2116	26.4	4.67	6.3
2216	28.9	4.87	6.2
2316	31.4	5.1	6.5
2416	34	5.33	6.8
2516	37	5.55	7.2
2616	39.8	5.78	7.5
2716	42.8	6.03	9.1
2816	45.8	6.42	8.1
2916	49	6.5	8.5
3016	52.4	6.73	9.3
3116	55.7	6.95	9.1
3216	59.3	7.15	9.5
3316	63.2	7.34	10.3
3416	67	7.57	10.3
3516	70.9	7.79	11.1
3616	74.7	8.03	11
3716	79	8.29	11.8
3816	83.2	8.56	11.7
3916	87.7	8.72	12.2

Continued on next page

Table B.3 – continued from previous page

Length (bp)	CPU without the pruning mechanism (ms)	CPU With the pruning mechanism (ms)	GPU Without the pruning mechanism (ms)
4016	92	9.25	13
4116	96.9	9.16	13.5
4216	101.5	9.44	13.8
4316	106.4	9.93	14.3
4416	111.1	9.96	14.2
4516	116.3	10.1	15
4616	121.8	10.31	15.4
4716	126.7	10.48	15.5
4816	131.9	10.72	16.3
4916	137.7	10.95	16.8
5016	143.1	11.23	17.2
5116	148.9	11.65	18.1
5216	155.2	11.87	18.1
5316	160.7	11.91	18.8
5416	167.1	12.2	19.2
5516	172.5	12.38	21.4
5616	178.8	12.74	20.3
5716	184.9	13.17	20.8
5816	191.6	13.28	21.3
5916	197.8	13.3	21.7
6016	204.2	13.63	22.4
6116	211.2	13.8	22.9
6216	218.2	14.03	23.3
6316	224.7	14.19	23.9
6416	231.9	14.37	25.6
6516	239.2	14.68	24.1
6616	247.3	15.03	25.5
6716	253.7	15.2	26.5
6816	261.2	15.36	26.5
6916	268.5	15.63	27.7
7016	277.1	16.11	25.9
7116	284.5	15.89	28.5
7216	292.1	16.53	29.2
7316	300.9	16.36	28.9
7416	309	16.55	30.3

B.4 Parallelization Level 2 and Level 3 Test 1

Table B.4 Performance comparison for different number of alignments at 8 bp.

Number of Alignments	**GPU with memory-saving version (ms)**	**GPU with time-saving version (ms)**	**CPU Version (ms)**
30	0	0	0.035
60	0	0	0.067
90	0	0	0.1
120	0	0	0.132
150	0.1	0	0.164
180	0.1	0	0.196
210	0.1	0	0.229
240	0.1	0	0.263
270	0.1	0	0.296
300	0.1	0.1	0.327
330	0.1	0.1	0.361
360	0.1	0.1	0.404
390	0.2	0.1	0.429
420	0.2	0.1	0.473
450	0.2	0.1	0.497
480	0.2	0.1	0.522
510	0.2	0.1	0.558
540	0.2	0.1	0.599
570	0.2	0.1	0.619
600	0.2	0.1	0.683
630	0.2	0.1	0.687
660	0.2	0.1	0.713
690	0.3	0.1	0.751
720	0.3	0.1	0.782
750	0.3	0.1	0.815
780	0.3	0.1	0.846
810	0.3	0.2	0.892
840	0.3	0.1	0.914
870	0.3	0.1	0.97
900	0.3	0.1	0.982
930	0.3	0.1	1.008
960	0.4	0.1	1.046
990	0.4	0.1	1.076
1020	0.4	0.1	1.11
1050	0.4	0.1	1.558
1080	0.4	0.2	1.168
1110	0.5	0.2	1.255
1140	0.4	0.2	1.255
1170	0.5	0.2	1.279
1200	0.4	0.2	1.298
1230	0.5	0.2	1.327

Continued on next page

Table B.4 – continued from previous page

Number of Alignments	GPU with memory-saving version (ms)	GPU with time-saving version (ms)	CPU version (ms)
1260	0.5	0.2	1.364
1290	0.5	0.2	1.396
1320	0.5	0.2	1.427
1350	0.5	0.2	1.467
1380	0.5	0.2	1.492
1410	0.5	0.2	1.533
1440	0.5	0.2	1.555
1470	0.6	0.2	1.601
1500	0.6	0.2	1.633
1530	0.6	0.2	1.652
1560	0.6	0.2	1.686
1590	0.6	0.2	2.136
1620	0.6	0.2	2.133
1650	0.6	0.2	1.809
1680	0.6	0.2	1.84
1710	0.6	0.2	1.845
1740	0.7	0.2	2.178
1770	0.7	0.2	1.918
1800	0.7	0.2	2.32
1830	0.7	0.2	1.994
1860	0.7	0.2	2.018
1890	0.7	0.3	2.067
1920	0.7	0.2	2.079
1950	0.7	0.2	2.101
1980	0.7	0.3	2.136
2010	0.7	0.3	2.168
2040	0.8	0.3	2.357
2070	0.9	0.3	2.236
2100	0.7	0.3	2.263
2130	0.8	0.3	2.784
2160	0.8	0.3	2.329
2190	0.8	0.3	2.358
2220	0.8	0.3	2.392
2250	0.8	0.3	2.441
2280	0.8	0.3	2.476
2310	0.9	0.3	2.844
2340	0.8	0.3	2.924
2370	0.9	0.3	2.58
2400	0.9	0.3	2.588
2430	0.9	0.3	2.631
2460	0.9	0.3	2.68
2490	0.9	0.3	2.727
2520	0.9	0.3	2.712
2550	0.9	0.3	2.746

B.5 Parallelization Level 2 and Level 3 Test 2

Table B.5 Performance comparison for different number of alignments at 16 bp.

Number of Alignments	**GPU with Saving RAM Version (ms)**	**GPU with Saving Time Version (ms)**	**CPU Version (ms)**
30	0.1	0.1	0.114
60	0.1	0.1	0.223
90	0.2	0.1	0.339
120	0.2	0.1	0.458
150	0.3	0.1	0.57
180	0.4	0.1	0.673
210	0.5	0.2	0.809
240	0.5	0.2	0.911
270	0.5	0.2	1.006
300	0.5	0.2	1.115
330	0.6	0.2	1.248
360	0.7	0.2	1.36
390	0.8	0.2	1.45
420	0.8	0.2	1.565
450	0.8	0.3	1.704
480	0.8	0.3	1.784
510	0.9	0.3	1.899
540	1	0.3	2.049
570	1	0.3	2.15
600	1	0.3	2.228
630	1.1	0.3	2.338
660	1.1	0.3	2.49
690	1.2	0.3	2.563
720	1.3	0.4	2.673
750	1.3	0.4	2.829
780	1.4	0.4	2.942
810	1.4	0.4	3.007
840	1.5	0.4	3.115
870	1.5	0.5	3.283
900	1.6	0.4	3.348
930	1.6	0.4	3.469
960	1.7	0.4	3.635
990	1.7	0.4	3.731
1020	1.8	0.5	3.792
1050	1.8	0.5	3.896
1080	1.2	0.5	4.078
1110	1.9	0.5	4.115
1140	1.9	0.5	4.231
1170	2	0.5	4.416
1200	2	0.5	4.524
1230	2.1	0.5	4.576
1260	1.4	0.5	4.671
1290	1.6	0.6	4.864

Continued on next page

Table B.5 – continued from previous page

Number of Alignments	**GPU with memory-saving version (ms)**	**GPU with time-saving version (ms)**	**CPU Version (ms)**
1320	2.2	0.6	4.9
1350	2.3	0.6	5.001
1380	1.7	0.6	5.201
1410	1.8	0.6	5.314
1440	2.4	0.6	5.347
1470	1.9	0.6	5.612
1500	1.9	0.6	5.647
1530	1.9	0.7	5.775
1560	2	0.7	5.797
1590	2	0.7	5.996
1620	2.1	0.7	6.097
1650	2.1	0.7	6.109
1680	2.2	0.7	6.224
1710	2.2	0.7	6.447
1740	2.2	0.7	6.559
1770	1.6	0.7	6.56
1800	2.3	0.7	6.783
1830	2.4	0.8	6.902
1860	1.8	0.8	6.907
1890	1.9	0.8	7.019
1920	1.9	0.8	7.256
1950	2.7	0.8	7.35
1980	1.9	0.8	7.335
2010	2.7	0.8	7.57
2040	2.7	0.8	7.694
2070	2.7	0.9	7.815
2100	2.8	0.9	7.775
2130	3.1	0.9	8.026
2160	2.3	1.2	8.151
2190	2.3	0.9	8.11
2220	3	0.9	8.362
2250	3.4	0.9	8.479
2280	2.5	0.9	8.457
2310	2.5	0.9	8.57
2340	3.2	0.9	8.824
2370	2.6	1	9.119
2400	2.6	1	9.019
2430	2.7	1	9.166
2460	2.7	1	9.273
2490	2.8	1	9.223
2520	2.9	1	9.454
2550	2.9	1	9.612

B.6 Parallelization Level 2 and Level 3 Test 3

Table B.6 Performance comparison for different number of alignments at 32 bp.

Number of Alignments	**GPU with memory-saving version (ms)**	**GPU with time-saving version (ms)**	**CPU Version (ms)**
30	0.3	0.1	0.426
60	0.5	0.2	0.847
90	0.7	0.3	1.232
120	1.4	0.3	1.813
150	1.2	0.3	2.046
180	1.4	0.4	2.455
210	1.7	0.5	2.878
240	1.8	0.5	3.331
270	1.4	0.5	3.692
300	2.7	0.6	4.129
330	1.9	0.6	4.526
360	2.1	0.7	4.905
390	2.3	0.7	5.325
420	2.5	0.7	5.716
450	2.8	0.8	6.141
480	2.5	0.9	6.556
510	3.1	0.9	7.62
540	2.8	0.9	7.518
570	3.2	1	8.39
600	3.6	1	8.363
630	3.3	1.3	8.724
660	2.8	1.1	9.069
690	3.8	1.2	9.796
720	4	1.2	9.84
750	3.5	1.2	10.21
780	4.6	1.3	10.634
810	4.1	1.4	11.18

C ksw_extend2 GPU Version

C.1 definitions.h

```
#ifndef DEFINITIONS_H_
#define DEFINITIONS_H_


/**
 * Warris's adaptation of the Smith-Waterman algorithm (WASWA).
 *
 * Requires a NVidia Geforce CUDA 2.1 with at least 1.3 compute capability
   .
 *
 * @author Sven Warris
 * @version 1.1
 */

/** maximum X per block (used in dimensions for blocks and amount of
   shared memory */
#define SHARED_X 8
/** maximum Y per block (used in dimensions for blocks and amount of
   shared memory */
#define SHARED_Y 8
#define SW_VERBOSE 4

#define FILL_SCORE INT_MIN
#define FILL_CHARACTER 5

#endif /*DEFINITIONS_H_*/
```

Listing C.1 definitions.h

C.2 typedefs.h

```c
#ifndef TYPEDEFS_H_
#define TYPEDEFS_H_

#include "definitions.h"
#include <stdint.h>
typedef struct {
  int64_t rbeg;
  int32_t qbeg, len;
  int score;
} mem_seed_t; // unaligned memory

typedef struct {
  int a, b;               // match score and mismatch penalty
  int o_del, e_del;
  int o_ins, e_ins;
  int pen_unpaired;       // phred-scaled penalty for unpaired reads
  int pen_clip5, pen_clip3; // clipping penalty. This score is not deducted from the DP score.
  int w;                  // band width
  int zdrop;              // Z-dropoff

  uint64_t max_mem_intv;

  int T;                  // output score threshold; only affecting output
  int flag;               // see MEM_F_* macros
  int min_seed_len;       // minimum seed length
  int min_chain_weight;
  int max_chain_extend;
  float split_factor; // split into a seed if MEM is longer than min_seed_len*split_factor
  int split_width; // split into a seed if its occurence is smaller than this value
  int max_occ;       // skip a seed if its occurence is larger than this value
  int max_chain_gap; // do not chain seed if it is max_chain_gap-bp away from the closest seed
  int n_threads;          // number of threads
  int chunk_size;         // process chunk_size-bp sequences in a batch
  float mask_level; // regard a hit as redundant if the overlap with another better hit is over mask_level times the min length of the two hits
  float drop_ratio; // drop a chain if its seed coverage is below drop_ratio times the seed coverage of a better chain overlapping with the small chain
  float XA_drop_ratio; // when counting hits for the XA tag, ignore alignments with score < XA_drop_ratio * max_score; only effective for the XA tag
  float mask_level_redun;
  float mapQ_coef_len;
  int mapQ_coef_fac;
  int max_ins; // when estimating insert size distribution, skip pairs with insert longer than this value
  int max_matesw; // perform maximally max_matesw rounds of mate-SW for each end
  int max_XA_hits, max_XA_hits_alt; // if there are max_hits or fewer, output them all
  int8_t mat[25];         // scoring matrix; mat[0] == 0 if unset
} mem_opt_t;


/* Scorings matrix for each thread block */
typedef struct {
  int value[SHARED_X+3][SHARED_Y+3];
}  LocalMatrix;


/* Scorings matrix for each sequence alignment */
typedef struct {
  LocalMatrix matrix[2][2];
} ScoringsMatrix;


#endif /* TYPEDEFS_H_ */
```

Listing C.2 typedefs.h

C.3 gpuAlign.h

```
void bwa_fill_scmat(int a, int b, int8_t mat[25]) {
  int i, j, k;
  for (i = k = 0; i < 4; ++i) {
    for (j = 0; j < 4; ++j)
      mat[k++] = i == j ? a : -b;
    mat[k++] = -1; // ambiguous base, which is our n.
  }
  for (j = 0; j < 5; ++j)
    mat[k++] = -1;
}


typedef struct {
  int a, b;                // match score and mismatch penalty
  int o_del, e_del;
  int o_ins, e_ins;
  int pen_unpaired;        // phred-scaled penalty for unpaired reads
  int pen_clip5, pen_clip3; // clipping penalty. This score is not
    deducted from the DP score.
  int w;                   // band width
  int zdrop;               // Z-dropoff

  uint64_t max_mem_intv;

  int T;                   // output score threshold; only affecting output
  int flag;                // see MEM_F_* macros
  int min_seed_len;        // minimum seed length
  int min_chain_weight;
  int max_chain_extend;
  float split_factor; // split into a seed if MEM is longer than
    min_seed_len*split_factor
  int split_width; // split into a seed if its occurence is smaller than
    this value
  int max_occ;       // skip a seed if its occurence is larger than this
    value
  int max_chain_gap; // do not chain seed if it is max_chain_gap-bp away
    from the closest seed
  int n_threads;           // number of threads
  int chunk_size;          // process chunk_size-bp sequences in a batch
  float mask_level; // regard a hit as redundant if the overlap with
    another better hit is over mask_level times the min length of the two
    hits
  float drop_ratio; // drop a chain if its seed coverage is below
    drop_ratio times the seed coverage of a better chain overlapping with
    the small chain
  float XA_drop_ratio; // when counting hits for the XA tag, ignore
    alignments with score < XA_drop_ratio * max_score; only effective for
    the XA tag
  float mask_level_redun;
  float mapQ_coef_len;
  int mapQ_coef_fac;
  int max_ins; // when estimating insert size distribution, skip pairs
    with insert longer than this value
  int max_matesw; // perform maximally max_matesw rounds of mate-SW for
    each end
  int max_XA_hits, max_XA_hits_alt; // if there are max_hits or fewer,
    output them all
  int8_t mat[25];          // scoring matrix; mat[0] == 0 if unset
} mem_opt_t;


mem_opt_t *mem_opt_init() {
  mem_opt_t *o;
  o = (mem_opt_t*)calloc(1, sizeof(mem_opt_t));
  o->flag = 0;
  o->a = 1;
  o->b = 4; //instertion penalty, deletion paenlty
  o->o_del = o->o_ins = 6;
  o->e_del = o->e_ins = 1;
  //o->o_del = o->o_ins = 2;
  //o->e_del = o->e_ins = 2;
  o->w = 100;
  o->T = 30;
  o->zdrop = 100;
  o->pen_unpaired = 17;
  o->pen_clip5 = o->pen_clip3 = 5;

  o->max_mem_intv = 20;

  o->min_seed_len = 19;
  o->split_width = 10;
  o->max_occ = 500;
  o->max_chain_gap = 10000;
  o->max_ins = 10000;
  o->mask_level = 0.50;
  o->drop_ratio = 0.50;
  o->XA_drop_ratio = 0.80;
  o->split_factor = 1.5;
  o->chunk_size = 10000000;
  o->n_threads = 1;
  o->max_XA_hits = 5;
  o->max_XA_hits_alt = 200;
  o->max_matesw = 50;
  o->mask_level_redun = 0.95;
  o->min_chain_weight = 0;
  o->max_chain_extend = 1 << 30;
  o->mapQ_coef_len = 50;
  o->mapQ_coef_fac = log(o->mapQ_coef_len);
  bwa_fill_scmat(o->a, o->b, o->mat);
  return o;
}



/*below is the method for queue testing*/
void queue_testing(){
  queue *q;
  q = (queue*)malloc(sizeof(queue*));
  queue_initialization(q);

  struct sw_ext *kk = (sw_ext*)malloc(sizeof(sw_ext*));
  kk->qlen = 20;

  struct sw_ext *gg = (sw_ext*)malloc(sizeof(sw_ext*));
  gg->qlen = 30;
//  struct sw_ext gg = {
//      .qlen = 30,
//  };

```

```
  queue_enqueue(q, kk);
  queue_enqueue(q,gg);
  printf("Queue before dequeue\n");
  queue_display(q->front);
//  struct sw_ext result[q->count];
//  to_array_list(q, &result);
//  for(int i = 0; i < 2; i++){
//    printf("%d\n", result[i].qlen+=5);
//  }
//
//
//  array *a;
//  a = queue_to_array(q, a);
}
```

Listing C.3 gpuAlign.h

C.4 gpuAlign.cu

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <inttypes.h>
#include <stdlib.h>
#include <stdint.h>
#include <ctype.h>
#include <sys/time.h>
#include "smithwaterman.h"
#include "gpuAlign.h"
#include "ksw.h"
#define RD_SEED 6




/*this method pass in a pointer address and then generate query and target
     test references*/
void ref_seq_gen(uint8_t **query, uint8_t **target, int query_size, int
    target_size, int chance_percent){

  (*query) = (uint8_t*)calloc(query_size,sizeof(uint8_t));
  (*target) = (uint8_t*)calloc(target_size,sizeof(uint8_t));
  uint8_t *tmp_query = (*query);
  uint8_t *tmp_target = (*target);
  bool has_longer_query = query_size > target_size;
  int min_size;
  int max_size;
  int idx;
  if(has_longer_query){
    min_size = target_size;
    max_size = query_size;
  }else{
    min_size = query_size;
    max_size = target_size;
  }

  for(idx = 0; idx < min_size; idx++){
    int chance = rand()%100;
    *tmp_query = rand()%4;
    if(chance > chance_percent) *tmp_target = rand()%4;
    else *tmp_target = *tmp_query;
    tmp_query++;
    tmp_target++;
  }

  uint8_t *tmp_pt = has_longer_query? tmp_query : tmp_target;
  for(; idx < max_size; idx++){
    *tmp_pt = rand()%4;
    tmp_pt++;
  }
  return;
}

/*
*single extention information, everything is about current single
    extension
```

```c
*/
void sw_ext_int(sw_ext **in, int in_h0, int query_size, int target_size,
   int chance_percent){
  //int* h_col;//this should be a shared information
  uint8_t *t_qu;
  uint8_t *t_ta;
  ref_seq_gen(&t_qu,&t_ta, query_size,target_size,chance_percent);
  (*in)->sc0 = in_h0;
  (*in)->query = t_qu;
  (*in)->target = t_ta;
  (*in)->qlen = query_size;
  (*in)->tlen = target_size;
  (*in)->gtle = 0;
  (*in)->aw[0] =100;
  (*in)->aw[1] = 100;

  (*in)->rmax[0] = 0;
  (*in)->rmax[1] = 0;


  (*in)->max_off[0] = 0;
  (*in)->max_off[1] = 0;

  (*in)->idx = 0;
  (*in)->seed_idx=0;
  (*in)->left_right=0;

}

void sw_state_init(int **in, unsigned int in_dims3, unsigned int in_dims4,
    mem_opt_t *opt){
  (*in) = (int*)calloc(21,sizeof(unsigned int));
  int *tmp_p = (*in);
  *(tmp_p + dims3) = in_dims3;
  *(tmp_p + dims4) = in_dims4;
  *(tmp_p + o_del) = opt->o_del;
  *(tmp_p + e_del) = opt->e_del;
  *(tmp_p + o_ins) = opt->o_ins;
  *(tmp_p + e_ins) = opt->e_ins;
  *(tmp_p + pen_clip3) = opt->pen_clip3;
  *(tmp_p + pen_clip5) = opt->pen_clip5;
  *(tmp_p + zdrop) = opt->zdrop;
  tmp_p = (*in);

}

void print_int_array(int* pt, int row, int col){
  //int* tmp = pt;
  printf("2D array[%d][%d]\n", row, col);
  for(int r = 0; r < row; r++){
    for(int c = 0; c < col; c++){
      printf("[%d]", *(pt+r*8+c));
      //tmp++;
    }
    printf("\n");
  }
}

void print_input(int test_size, struct sw_ext* h_swext) {
  for (int i = 0; i < test_size; i++) {
    for (int j = 0; j < (h_swext + i)->qlen; j++) {
      const uint8_t* tmp_q = (h_swext + i)->query;
      printf("%c", "ACGTN"[*(tmp_q + j)]);
    }
    printf("\n");
    for (int j = 0; j < (h_swext + i)->qlen; j++) {
      const uint8_t* tmp_q = (h_swext + i)->query;
      printf("%d,", *(tmp_q + j));
    }
    printf("\n");
    for (int j = 0; j < (h_swext + i)->tlen; j++) {
      const uint8_t* tmp_t = (h_swext + i)->target;
      printf("%c", "ACGTN"[*(tmp_t + j)]);
    }
    printf("\n");
    for (int j = 0; j < (h_swext + i)->tlen; j++) {
      const uint8_t* tmp_t = (h_swext + i)->target;
      printf("%d,", *(tmp_t + j));
    }
    printf("\n");
  }
}

void ptr_h_value(int* h_shared, LocalMatrix* result) {
  for (int i = 0;i< *(h_shared + block_x_len) * *(h_shared + block_y_len)*
     *(h_shared + dims3); i++) {
    printf("currID: %d  ", i);
    print_int_array(&(result + i)->h_value[0][0], SHARED_X, SHARED_Y);
  }
}

void computeMax(sw_ext* curr_ext, int* h_shared, const int8_t* mat,
    LocalMatrix* h_scoringMatrix, int m)
{ int w = curr_ext->aw[curr_ext->left_right];

  int qlen = curr_ext->qlen;
  int tlen = curr_ext->tlen;
  int end_bonus = curr_ext->left_right? *(h_shared + pen_clip3) : *(
    h_shared + pen_clip5) ;
  int line_y = 0;//thread y
  int max_ie = -1;
  int gscore = -1;
  int MaxScore = curr_ext->sc0;
  int MaxIdx_X = -1;
  int MaxIdx_Y = -1;
  int max_off = 0;
  int SW_VERBOSE = 0;

  int* h_col = curr_ext->h_col;

  {
    int max_ins, max_del, beg, end;
    //int lbk_lCol = (qlen-1)%SHARED_X;
    int max, i;
    for (i = 0, max = 0; i < m * m; ++i) max = max > *(mat + i) ? max : *(
    mat + i);
    //qlen, assume is 10. the max score in the mat is 1. the end bonus
    assume is 2. so it would be (qlen*1+5-6)/1+1
    max_ins = (int) (((double) ((qlen * max + end_bonus - *(h_shared +
    o_ins))) / *(h_shared + e_ins) + 1.)); //get max insertion score
    max_ins = max_ins > 1 ? max_ins : 1;
    w = w < max_ins ? w : max_ins; //calculate max band width? which is
    defined as the max_ins in current situation.
    max_del = (int) (((double) ((qlen * max + end_bonus - *(h_shared +
    o_del))) / *(h_shared + e_del) + 1.)); //get max deletion score
    max_del = max_del > 1 ? max_del : 1;
    w = w < max_del ? w : max_del; // TODO: is this necessary?//ok, I think the max bandwidth is the max insertion/deletion score you can get.
    // DP loop
    beg = 0;
    end = qlen;
    //printf("[w: %2d]", w);
    for (int bk_y = 0; bk_y < *(h_shared +block_y_len); bk_y++) {
      for (int th_y = 0; LIKELY(th_y < SHARED_Y) && LIKELY(line_y < tlen);
          th_y++) {
        if (beg < line_y - w)beg = line_y - w; //>=o, make sure it is not over the max scores, or band width
        if (end > line_y + w + 1) end = line_y + w + 1;         //same thing
        if (end > qlen) end = qlen;
        {         //for calculating h1 and max ie
          int h1;
          if (end == qlen) {
            h1 =(h_scoringMatrix + bk_y * *(h_shared +block_x_len)  + *(h_shared +block_x_len) - 1)->h_value[th_y][(end - 1) % SHARED_X];
            max_ie = gscore > h1 ? max_ie : line_y;
            gscore = gscore > h1 ? gscore : h1;
          }
          if(SW_VERBOSE > 3) printf("!!!end[%d, %d][h1: %d, max_ie: %d, gscore: %d]", line_y, end, h1, max_ie, gscore);

        }
        //printf("maxie: %d", max_ie);

        int CurrM = (h_scoringMatrix + bk_y * *(h_shared +block_x_len))->s_value[th_y]; //current max at current line
        int mj = (h_scoringMatrix + bk_y * *(h_shared +block_x_len))->x_value[th_y]; //current max xloc
        for (int bk_x = 1; bk_x < *(h_shared +block_x_len); bk_x++) { // for each block
          int h = (h_scoringMatrix + bk_y * *(h_shared +block_x_len) + bk_x)->s_value[th_y];
          if (SW_VERBOSE > 3) printf("!!%d!!", h);
          //compare each block's max
          if (h > CurrM) {
            CurrM = h;
            mj = bk_x * SHARED_X + (h_scoringMatrix + bk_y * *(h_shared +block_x_len) + bk_x)->x_value[th_y];
          }
        }
        if (CurrM == 0) break;

        if (UNLIKELY(CurrM > MaxScore)) {
          max_off = max_off > abs(mj - line_y) ?  max_off : abs(mj - line_y);
          MaxScore = CurrM;
          MaxIdx_X = mj;
          MaxIdx_Y = line_y;
        }

        if (SW_VERBOSE>3)
          printf( "001 [CurrM: %d, max: %d][max_off: %d, mj: %d, line_y: %d, maxoff_diff>?: %d]\n",CurrM, MaxScore, max_off, mj, line_y,max_off > abs(mj - line_y));
          if (SW_VERBOSE > 3) printf("[beg: %d, end: %d]\n", beg, end);
```

```
      {        //calculate end, and begin

        int *curr_eh = (int*) malloc(sizeof(int) * (qlen + 1));
        //int *curr_eh2 = (int*) malloc(sizeof(int) * (qlen + 1));
        //curr_h[0] = h_col[line_y + 1];
        { //this is for printing out f value
          int next_h = h_col[line_y + 1];
          int line_x = 0;
          for (int bk_x = 0; bk_x < *(h_shared +block_x_len); bk_x++) {
            for (int th_x = 0;LIKELY(th_x <SHARED_X)&& LIKELY(line_x < qlen); th_x++) {
              curr_eh[line_x]= (h_scoringMatrix + bk_y * *(h_shared + block_x_len)+ bk_x)->f_value[th_y][th_x] + next_h;
              next_h = (h_scoringMatrix + bk_y * *(h_shared +block_x_len) + bk_x)->h_value[th_y][th_x];
              line_x++;

            }
          }
          //curr_f[qlen] = 0;
          curr_eh[qlen] = next_h;

        }

        int j;
        for (j = beg;LIKELY(j < end) && curr_eh[j] == 0;++j);
        beg = j;
        for (j = end;LIKELY(j >= beg) && curr_eh[j] == 0; --j);
        end = j + 2 < qlen ? j + 2 : qlen;
      }


      line_y++;

    }
  }
}

curr_ext->qle = MaxIdx_X + 1;
curr_ext->tle = MaxIdx_Y + 1;
curr_ext->gtle = max_ie + 1;
curr_ext->gscore = gscore;
curr_ext->max_off[curr_ext->left_right] = max_off;
curr_ext->score= MaxScore;
if (SW_VERBOSE) printf(">>>>>>result report: 001 [qle: %d, tle: %d, gtle: %d, gscore: %d, max_off: %d, maxScore: %d]\n", curr_ext->qle, curr_ext->tle, curr_ext->gtle, curr_ext->gscore, curr_ext->max_off[curr_ext->left_right], curr_ext->score);

}






void testing2(int test_size){

  mem_opt_t *opt;
  opt = mem_opt_init();
  bwa_fill_scmat(opt->a, opt->b, opt->mat);

  struct sw_ext *h_swext = (sw_ext*)calloc(test_size,sizeof(sw_ext));

  //adding up testing information

  int g_qlen = 32;
  int g_tlen = 32;
  int chance = 80;
  int score_h0 = 100;

  srand(RD_SEED);
  struct sw_ext *tmp_swext = h_swext;
  for(int i = 0; i < test_size; i++){
    sw_ext_int(&tmp_swext, score_h0, g_qlen, g_tlen, chance);
    tmp_swext++;
  }
  uint8_t tmp_q[] = {1,1,1,3,3,1,0,0,0,2,2,3,1,3,3,1};
  uint8_t tmp_t[] = {0,0,0,1,1,1,3,3,1,0,0,0,2,2,3,1};
  h_swext->query = &tmp_q[0];
  h_swext->target= &tmp_t[0];
  int *h_shared;
  sw_state_init(&h_shared, test_size,1, opt);
  //initialization is over
  //testing starts here:
  //print_input(test_size, h_swext);
  LocalMatrix *result = gpu_sw_seed_extend(h_swext, h_shared, opt->mat);
  //ptr_h_value(h_shared, result);


  LocalMatrix *result2 = gpu_sw_seed_extend_v2(h_swext, h_shared, opt->mat
    );
  //ptr_h_value(h_shared, result2);

  for (int i = 0;i< test_size; i++) {
    computeMax((h_swext+i), h_shared, opt->mat, (result+i**(h_shared +
   block_x_len) * *(h_shared + block_y_len)), 5);
    }




//   clock_t t;
//   t = clock();
  struct timeval stop, start;
  gettimeofday(&start, NULL);

  for (int i = 0;i< *(h_shared + dims3); i++) {
    (h_swext+i)->score = ksw_extend2((h_swext+i)->qlen, (h_swext+i)->query
   , (h_swext+i)->tlen, (h_swext+i)->target, 5, opt->mat, opt->o_del, opt
   ->e_del, opt->o_ins, opt->e_ins,
        (h_swext+i)->aw[1],
        opt->pen_clip3, opt->zdrop,
        (h_swext+i)->sc0,
        &((h_swext+i)->qle),
        &(h_swext+i)->tle,
        &(h_swext+i)->gtle,
        &(h_swext+i)->gscore,
        &(h_swext+i)->max_off[1]);
  }

  gettimeofday(&stop, NULL);
  printf("%lu\n", (stop.tv_sec - start.tv_sec) * 1000000 + stop.tv_usec -
   start.tv_usec);
  //t = clock()-t;
//  double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
//  printf("%f", time_taken);
}




int main(void) {
  //queue_testing();

for(int i = 2; i < 2560; i+=2){
  printf("%d,", i);
  testing2(i);

}

return 0;

}
```

Listing C.4 gpuAlign.cu

C.5 smithwaterman.h

```
/*
*Connor Li @ Feb. 11th, 2019 V3.2
*Brock University Computer Science
*this modified smithwaterman algorithm is to be used in GPU version of
   KSW_extend2 computation
*we consider this is computing alignment on 3 dimention
*/

/** maximum X per block (used in dimensions for blocks and amount of
   shared memory */
#define SHARED_X 8
/** maximum Y per block (used in dimensions for blocks and amount of
   shared memory */
#define SHARED_Y 8
/*wether we need to test input output or not*/

/*fill score, we define int's min-value as our fill score*/
#define FILL_SCORE INT_MIN
/*our fill charactor is defined as 5, as a, t, c, g, n is 0 to 4*/
#define FILL_CHARACTER 5
#define EPT_SCORE 0


#ifdef __GNUC__
#define LIKELY(x) __builtin_expect((x),1)
#define UNLIKELY(x) __builtin_expect((x),0)
#else
#define LIKELY(x) (x)
#define UNLIKELY(x) (x)
#endif



/* Scorings matrix for each thread block
*we have the following plan:
*int h_value -> to store h value
*int f_value -> to temperately store f value during matrix calculation
*int e_value -> to temperately store e value during matrix calculation
*int s_value -> for future calculation references, it stores the max value
    of each line
*int x_value -> for future calculation references, it stores the location
   of the max value ineach line
*/
typedef struct {
  //h_value is needed to store the computed value
  int h_value[SHARED_Y][SHARED_X];
  int f_value[SHARED_Y][SHARED_X];
  int e_value[SHARED_Y];
  int s_value[SHARED_Y];
  int x_value[SHARED_Y];
  //f_value is needed to store the f_value for post computation
}  LocalMatrix;

//matrix for score computation
__constant__ int8_t d_mat[25];
//sw_shared infomation, this is never changed
__constant__ int d_shared[21];
```

```
enum sw_state {
  //in all alignment, the longest sequence and reference
  max_qlen,//the longest read / sequence1
  max_tlen,//the longest reference
  //consider everything together
  max_x,//max_x, consider each block is 8 * 8, how many cell we need on x
    axis?
  max_y,//max_y, consider each block is 8 * 8, how many cell we need on y
    axis?
  //consicer block as cells, what we have? <on the 3dim>
  block_x_len,//consider block as a cell, how many block on the x axis?
  block_y_len,//consider block as a cell, how many block on the y axis?
  block_diagnal_len,//max(block_x_len, block_y_len)
  //consider block as cell, how many block do we have on the diagnal in a
    single alignment?
  //7
  alignment_x,//the total number of cell on x as we compute all x axis
  alignment_y,//the total number of cell on y as we compute all y axis
  alignment_diagnal_len,//block_diagnal_len * dim3, consider block as a
    cell, how many block on the diagnal on all of the localMatrix?
  //10

  dims3,//3ed dimension, we define as our number of sequence
  dims4,//fourth dimension, which is left for next level of
    parallelization
  //12
  o_del,//opening del
  e_del,//extension del
  o_ins,//opening insertion
  e_ins,//extension insertion
  oe_del,//open extention opening deletion
  oe_ins, //open extention opening insertion
  //18
  pen_clip3,
  pen_clip5,
  zdrop
  //21
};

/*
*single extention information, everything is about current single
   extension
*/
struct sw_ext{
  //int* h_col;//this should be a shared information
  int qlen;
  const uint8_t *query;
  int tlen;
  const uint8_t *target;
  int sc0;
  int* h_col;


  int aw[2];
  int rmax[2];
  int max_off[2];

  int qle;
  int tle;
  int gtle;
  int gscore;
  int score;
  int idx;
  //mem_alnreg_t *a;
  //mem_seed_t *s;
  int seed_idx;
  int left_right;//0 is left and 1 is right

};



// A linked list node
struct node{
    struct sw_ext *data;
    struct node *next;
};


/*this is our struct linked list*/
typedef struct
{
    int count;
    struct node *front;
    struct node *rear;
}queue;


typedef struct{
    int count;
    struct sw_ext *data;

}array;


/*queue initialization*/
void queue_initialization(queue *q);

/*check if queue is ompty or not*/
int queue_isempty(queue *q);

/*add an item to the queue*/
void queue_enqueue(queue *q, struct sw_ext *value);

/*this will delete the oldest item in the list*/
struct sw_ext* queue_dequeue(queue *q);

/*this method displays queue item*/
void queue_display(node *head);


void queue_to_array_list (queue *q, void *data);

array *queue_to_array(queue *q, array *a);

extern "C" LocalMatrix* gpu_sw_seed_extend(sw_ext *swext, int *h_shared,
    int8_t *mat);

```

```
extern "C" LocalMatrix *gpu_sw_seed_extend_v2(sw_ext *swext, int *h_shared
    , int8_t *mat);
```

Listing C.5 smithwaterman.h

C.6 smithwaterman.cu

```
/*
*Connor Li @ Feb. 20th, 2019 V3.3
*Brock University Computer Science
*this modified smithwaterman algorithm is to be used in GPU version of
   KSW_extend2 computation
*we consider this is computing alignment on 3 dimention
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <builtin_types.h>
#include <time.h>
#include <helper_cuda.h>
#include <stdint.h>
#include "smithwaterman.h"


/*queue initialization*/
void queue_initialization(queue *q)
{
    q->count = 0;
    q->front = NULL;
    q->rear = NULL;
}

/*check if queue is ompty or not*/
int queue_isempty(queue *q)
{
    return (q->rear == NULL);
}

/*add an item to the queue*/
void queue_enqueue(queue *q, struct sw_ext *value)
{
    node *tmp;
    tmp = (node*)malloc(sizeof(node));
    tmp->data = value;
    tmp->next = NULL;
    if(!queue_isempty(q))
    {
        q->rear->next = tmp;
        q->rear = tmp;
    }
    else
    {
        q->front = q->rear = tmp;
    }
    q->count++;
}

/*this will delete the oldest item in the list*/
struct sw_ext* queue_dequeue(queue *q)
{
    node *tmp;
    struct sw_ext *n = q->front->data;
    tmp = q->front;
    q->front = q->front->next;
    q->count--;
    free(tmp);
    return(n);
}

/*this method displays queue item*/
void queue_display(node *head)
{
    if(head == NULL)
    {
        printf("NULL\n");
    }
    else
    {
    /*prints out desired element*/
        printf("%d\n", head->data->qlen);
        queue_display(head->next);
    }
}

void queue_to_array_list (queue *q, void *data){
    //struct sw_ext result[]){
  struct sw_ext *result = reinterpret_cast<struct sw_ext*>(data);
    node* curr = q->front;
    int index = 0;
    while(curr!=NULL){
        result[index++] = *curr->data;
        curr = curr->next;
    }
}

array *queue_to_array(queue *q, array *a) {
    a = (array*)malloc(sizeof(array));
    //array_initialize(&a, q->count);
    a->count = q->count;
    struct sw_ext data[q->count];
    queue_to_array_list(q, &data);
    a->data = data;
    return a;
}

//                      [0]    [1]          [2]
//check if a is bigger than b, and assign bigger value to &c
__device__ bool getHigher(int a, int b, int *c){
  bool result = a > b;
  *c = result? a : b;
  return result;
}


/*we need to access two lines, we call: the diagnalLine has been
   computated before, we call it as diagnalLine
*the line that is even more before: we call it as diagnalLinePre
*there are two LocalMatrix array. here is what they do:
*our calculation is seperated into two diagnals, (which both is needed for
    calculation)
*1. d_diagnalLine
*and the current one is with the previous information
*/
__global__ void calculateScore(
    LocalMatrix *d_diagnalLine, LocalMatrix *d_diagnalLinePre,
    int *d_row, int *d_col,
    unsigned int x, unsigned int y,
    unsigned int numOfBlocks,
    uint8_t* d_sequences, uint8_t* d_references){
  unsigned int DIAGONAL = SHARED_Y + SHARED_X;
  int innerScore;
  /**
   * shared memory block for calculations. It requires
   * extra (+1 in both directions) space to hold
   * Neighboring cells
   */
  __shared__ int h_matrix[SHARED_Y+1][SHARED_X+1];
  __shared__ int e_matrix[SHARED_Y+1][SHARED_X+1];
  __shared__ int f_matrix[SHARED_Y+1][SHARED_X+1];
  //__shared__ int m_matrix[SHARED_Y][SHARED_X];
  __shared__ int s_maxima[SHARED_Y];
  __shared__ int x_maxloc[SHARED_Y];


  //if there are only one item, block y is the number of block we needed
  //x, y is our axises
  unsigned int currAlign = blockIdx.y%d_shared[dims3];//which alignment we
     are looking at
  unsigned int currBlockOrder = blockIdx.y/d_shared[dims3];//compute
    diagnally, the current MingCi of the block
  unsigned int blockx = x - currBlockOrder;//the block pos of x
  unsigned int blocky = y + currBlockOrder;//the block pos of y
  unsigned int tIDx = threadIdx.x;//current thread id of x
  unsigned int tIDy = threadIdx.y;//current thread id of y
  //unsigned int bIDx = blockIdx.x;//sequence id, 4th dim, always 1
  //unsigned int bIDy = currAlign;//bidy is considered as our alignemnt
    nubmer

  // indices of the current characters in both sequen
  int seqIdx = tIDx + currAlign * d_shared[max_x] + blockx * SHARED_X;//
    shorter read
  int refIdx = tIDy + currAlign * d_shared[max_y] + blocky * SHARED_Y;//
    longer ref

//initialization to the EPT_score
  int tmp_1 = (SHARED_Y+1)*(SHARED_X+1);

  memset(&h_matrix[0][0], EPT_SCORE, tmp_1*sizeof(int));
  memset(&e_matrix[0][0], EPT_SCORE, tmp_1*sizeof(int));
  memset(&f_matrix[0][0], EPT_SCORE, tmp_1*sizeof(int));
  memset(&s_maxima[0], EPT_SCORE, SHARED_Y*sizeof(int));
  memset(&x_maxloc[0], EPT_SCORE, SHARED_Y*sizeof(int));
  __syncthreads();

  int idx = 0;
//first block row first row
//we have multiple blocks, therefore, we have to be very carefull
  if(!blocky && !tIDy) {//when tIDy is 0, which would be the first line
    h_matrix[0][tIDx] = d_row[seqIdx];
  }
  if(!blockx && !tIDx){//tIDx is 0, left column
    h_matrix[tIDy][0] = d_col[refIdx];
    //if(!tIDx&&!tIDy) h_matrix[0][SHARED_X] = d_col[refIdx]
  }
  //surrounded line that we have to copy them from computed d_row and
    d_col
  //blocky is > 0
  if (blocky && !tIDy){
    //(x, y-1)
    idx = !y? blockIdx.y - d_shared[dims3] : blockIdx.y;
    if(tIDx) h_matrix[0][tIDx] = d_diagnalLine[idx].h_value[SHARED_Y-1][
   tIDx-1];
     f_matrix[0][tIDx] = d_diagnalLine[idx].f_value[SHARED_Y-1][tIDx];//
   for restoring previous h

  }
  else if(blockx && !tIDx && tIDy){
    //(x-1, y)
    idx = !y? blockIdx.y : blockIdx.y + d_shared[dims3];
    h_matrix[tIDy][0] = d_diagnalLine[idx].h_value[tIDy-1][SHARED_X-1];

  }

  if(blockx&& ! tIDx){
    //(x-1, y)
    idx = !y? blockIdx.y : blockIdx.y + d_shared[dims3];
     e_matrix[tIDy][0] = d_diagnalLine[idx].e_value[tIDy];//for restoring
   previous e

  }
  if (blockx && blocky && ! tIDx && !tIDy){
    //(x-1,y-1)
    int idx = !y? blockIdx.y - d_shared[dims3] : y == 1? blockIdx.y :
   blockIdx.y + d_shared[dims3];
    h_matrix[0][0] = d_diagnalLinePre[idx].h_value[SHARED_Y-1][SHARED_X
   -1];

  }


  /**
   * tXM1 and tYM1 are to store the current value of the thread Index.
   tIDx and tIDy are
   * both increased with 1 later on.
   */
  unsigned int tXM1 = tIDx;
  unsigned int tYM1 = tIDy;
  // shared location for the parts of the 2 sequences, for faster
   retrieval later on:
  __shared__ uint8_t s_seq[SHARED_X];
  __shared__ uint8_t s_ref[SHARED_Y];

  // copy sequence data to shared memory (shared is much faster than
   global)
  if (!tIDy){
    s_seq[tIDx] = d_sequences[seqIdx];
  }
  if (!tIDx){
    s_ref[tIDy] = d_references[refIdx];

  }
  __syncthreads();
  // set inner score (aka sequence match/mismatch score):
  uint8_t charSeq = s_seq[tIDx];
  uint8_t charRef = s_ref[tIDy];


  innerScore = charSeq == FILL_CHARACTER || charRef == FILL_CHARACTER ?
```

```
    FILL_SCORE : d_mat[charSeq+charRef*5];
  // transpose the index
  ++tIDx;
  ++tIDy;
  // set shared matrix to zero (starting point!)
  // wait until all elements have been copied to the shared memory block
  /**** sync barrier ****/

  __syncthreads();

    for (int i=0; i < DIAGONAL; ++i) {
    if(innerScore!=FILL_SCORE){
      if (i == tXM1+ tYM1) {
//          // calculate only when there are two valid characters
//          // this is necessary when the two sequences are not of equal
    length
//          // this is the SW-scoring of the cell:
//          // At the beginning of the loop: eh[j] = { H(i-1,j-1), E(i,j) },
     f = F(i,j) and h1 = H(i,j-1)
//          // Similar to SSE2-SW, cells are computed in the following order
    :
//          //   H(i,j)   = max{H(i-1,j-1)+S(i,j), E(i,j), F(i,j)}
//          //   E(i+1,j) = max{H(i,j)-gapo, E(i,j)} - gape
//          //   F(i,j+1) = max{H(i,j)-gapo, F(i,j)} - gape
        int M = h_matrix[tYM1][tXM1]? h_matrix[tYM1][tXM1] + innerScore:
    0;
        //m_matrix[tYM1][tXM1] = M;
        h_matrix[tIDy][tIDx] = max(max(M, e_matrix[tYM1][tXM1]), f_matrix[
    tYM1][tXM1]);
        e_matrix[tYM1][tIDx] = max(max(M-d_shared[oe_ins], e_matrix[tYM1][
    tXM1]-d_shared[e_ins]), 0);
        f_matrix[tIDy][tXM1] = max(max(M-d_shared[oe_del], f_matrix[tYM1][
    tXM1]-d_shared[e_del]), 0);

      }

    }

    if(i-1 == tXM1 + tYM1){
      if(!tXM1){
        s_maxima[tYM1] = h_matrix[tIDy][1];
        x_maxloc[tYM1] = tXM1;
      }
      else if(getHigher(h_matrix[tIDy][tIDx], s_maxima[tYM1], &s_maxima[
    tYM1]))
          x_maxloc[tYM1] = tXM1;

    }
    // wait until all threads have calculated their new score
      /**** sync barrier ****/
    __syncthreads();
  }



  //pass on the information to the next block
  //here we modify for our diagnalLine
  //int idx = get1DIdx(blockx, blocky, XdivSHARED_X);
  idx = blockIdx.y;
  d_diagnalLinePre[idx].h_value[tYM1][tXM1] = h_matrix[tIDy][tIDx];
  d_diagnalLinePre[idx].f_value[tYM1][tXM1] = f_matrix[tIDy][tXM1];

  //stored for next time computation
  if(!tXM1){
    d_diagnalLinePre[idx].e_value[tYM1]=e_matrix[tYM1][SHARED_X];
    d_diagnalLinePre[idx].s_value[tYM1]=s_maxima[tYM1];
    d_diagnalLinePre[idx].x_value[tYM1]=x_maxloc[tYM1];

  }
  __syncthreads();

}


/*
*sw_ext *sw_ext: this is our array of extension information
*int num_seq: number if sequence
*int num_target: number of targets
*this is our converter, we basically convert everything and get ready
    thing to be ready
*/
/*this method pass out a pointer of 2D array type*/
extern "C" LocalMatrix *gpu_sw_seed_extend(sw_ext *swext, int *h_shared,
    int8_t *mat){
  int *d_col = 0, *d_row = 0;
  uint8_t *d_sequences = 0, *d_references = 0;
  LocalMatrix* d_diagnalLine;//this is our diagnaline on the computed
    matrix
  LocalMatrix* d_diagnalLinePre;
  *(h_shared +oe_del) = *(h_shared +o_del) + *(h_shared +e_del);
  *(h_shared +oe_ins) = *(h_shared +o_ins) + *(h_shared +e_ins);
  //[1]search for max_qlen and max_tlen for all element
  //we are looking for the max qlen and max tlen
  *(h_shared +max_qlen) = swext->qlen;
  *(h_shared +max_tlen) = swext->tlen;
  //starts from 1
  for(int i = 1; i < *(h_shared + dims3); i++){//skip the first one
    if((swext+i)->qlen > *(h_shared +max_qlen)) *(h_shared +max_qlen) = (
    swext+i)->qlen;
    if((swext+i)->tlen > *(h_shared +max_tlen)) *(h_shared +max_tlen) = (
    swext+i)->tlen;
  }

  *(h_shared +block_x_len) = (int) ceil((double)*(h_shared +max_qlen)/
    SHARED_X);//how many 8*8 block on x div
  *(h_shared +block_y_len) = (int) ceil((double)*(h_shared +max_tlen)/
    SHARED_Y);//how many 8*8 block on y div
  *(h_shared +max_x) = *(h_shared +block_x_len) * SHARED_X;//for seed
    extension part
  *(h_shared +max_y) = *(h_shared +block_y_len) * SHARED_Y;//reference
    extension part
  *(h_shared +alignment_x) = *(h_shared +max_x) * *(h_shared +dims3);//
    TOTAL LENGTH OF X
  *(h_shared +alignment_y) = *(h_shared +max_y) * *(h_shared +dims3);//
    TOTAL LENGTH OF Y
  *(h_shared +block_diagnal_len) = max(*(h_shared +block_x_len), *(
    h_shared +block_y_len));//TOTAL LENGTH OF Y
  *(h_shared +alignment_diagnal_len) = *(h_shared +block_diagnal_len)**(
    h_shared +dims3);//TOTAL LENGTH OF Y

  int h_row[*(h_shared +alignment_x)];
  int h_col[*(h_shared +alignment_y)];
  uint8_t h_seq[*( h_shared +alignment_x)];
  uint8_t h_ref[*( h_shared +alignment_y)];
  LocalMatrix h_diagnalLine[*(h_shared+alignment_diagnal_len)];
  LocalMatrix *h_scoringMatrix = (LocalMatrix*) calloc(sizeof(LocalMatrix)
   , *(h_shared +dims4) * *(h_shared +dims3) * *(h_shared +block_y_len) *
   *(h_shared +block_x_len));

  //initialization is done here.
  //[2] initialization of top and left row

  //we are initializing for everyone
  //init h_row, h_col
  for(int align_idx = 0; align_idx < *(h_shared +dims3); align_idx++){//go
    though each alignment option here
    int start_x_pos = align_idx * *( h_shared +max_x);//ours 403//starting
    position
    int start_y_pos = align_idx * *( h_shared +max_y);//current alignment
    struct sw_ext *curr_sw_ext  = swext + align_idx;
    int curr_h0 = curr_sw_ext->sc0;


    h_row[start_x_pos+0] = curr_h0;//position is 0
    h_row[start_x_pos+1] = LIKELY(curr_h0 > *(h_shared +oe_ins))? curr_h0
   - *(h_shared +oe_ins) : 0;//404, position as 1
    for(int curr_x_loc = start_x_pos + 2; LIKELY(curr_x_loc < start_x_pos
   + *(h_shared +max_x)); ++curr_x_loc)
      h_row[curr_x_loc] = (curr_x_loc <= (start_x_pos + curr_sw_ext->qlen)
    && h_row[curr_x_loc - 1] > *(h_shared +e_ins))? h_row[curr_x_loc - 1]
   - *(h_shared +e_ins) : 0;

    // adjust $w if it is too large
    // generate the first row
    h_col[start_y_pos+0] = curr_h0; //eh[0].e = highest possible score
    h_col[start_y_pos + 1] = LIKELY(curr_h0 > *(h_shared +oe_del)) ?
   curr_h0 - *(h_shared +oe_del) : 0;
    for (int curr_y_loc =start_y_pos + 2; LIKELY(curr_y_loc < start_y_pos
   + *(h_shared +max_y)); ++curr_y_loc)
      h_col[curr_y_loc] = (curr_y_loc <= (start_y_pos + curr_sw_ext->tlen)
    && h_col[curr_y_loc - 1] > *(h_shared +e_del))? h_col[curr_y_loc - 1]
   - *(h_shared +e_del) : 0;

    for(int curr_x_loc = start_x_pos; LIKELY(curr_x_loc < start_x_pos + *(
    h_shared +max_x)); ++curr_x_loc)
      h_seq[curr_x_loc] = LIKELY(curr_x_loc < start_x_pos + curr_sw_ext->
   qlen)? curr_sw_ext->query[curr_x_loc-start_x_pos] : FILL_CHARACTER;

    for(int curr_y_loc = start_y_pos; LIKELY(curr_y_loc < start_y_pos + *(
    h_shared +max_y)); ++curr_y_loc)
      h_ref[curr_y_loc] = LIKELY(curr_y_loc < start_y_pos + curr_sw_ext->
   tlen)? curr_sw_ext->target[curr_y_loc-start_y_pos] : FILL_CHARACTER;


    curr_sw_ext->h_col = &h_col[start_y_pos];
  }

  checkCudaErrors(cudaMalloc((void **)&d_row, sizeof(h_row)));
  checkCudaErrors(cudaMalloc((void **)&d_col, sizeof(h_col)));
  checkCudaErrors(cudaMemcpy(d_row, &h_row[0], sizeof(h_row),
   cudaMemcpyHostToDevice));
  checkCudaErrors(cudaMemcpy(d_col, &h_col[0], sizeof(h_col),
   cudaMemcpyHostToDevice));

  checkCudaErrors(cudaMalloc((void** ) &d_sequences,sizeof(h_seq)));
  checkCudaErrors(cudaMalloc((void** ) &d_references,sizeof(h_ref)));
```

```
    checkCudaErrors(cudaMemcpy(d_sequences, &h_seq[0], sizeof(h_seq), cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(d_references, &h_ref[0], sizeof(h_ref), cudaMemcpyHostToDevice));

    checkCudaErrors(cudaMalloc((void**) &d_diagnalLine, sizeof(h_diagnalLine)));
    checkCudaErrors(cudaMalloc((void**) &d_diagnalLinePre, sizeof(h_diagnalLine)));

    checkCudaErrors(cudaMemcpyToSymbol(d_mat, mat, 25 * sizeof(int8_t),0, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpyToSymbol(d_shared, h_shared, 21 * sizeof(unsigned int),0, cudaMemcpyHostToDevice));

    unsigned int halfZhouChang =  *(h_shared +block_x_len) + *(h_shared + block_y_len); //half zhouchang of a matrix, width and hight
    unsigned int xiaoBian = halfZhouChang - *(h_shared + block_diagnal_len); //the width, xiaoBian, the smaller one also it is the Max number of blocks
    unsigned int daBian = *(h_shared + block_diagnal_len);//the hight, DaBian, the longer one, also it is the "starting to decrease at
    unsigned int numBlocks = 0, x = 0, y = 0;

    dim3 dimBlock(SHARED_X, SHARED_Y, 1);



    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);
    // adjust $w if it is too large
      //locate memory for d_eh
      for (unsigned int i = 1; LIKELY(i < halfZhouChang); ++i) {
        numBlocks = i <= xiaoBian? i : i >= daBian? halfZhouChang - i : xiaoBian;

        //reserve dim4 for anything beyound chain
        dim3 dimSWGrid(*(h_shared +dims4), *(h_shared +dims3) * numBlocks, 1);//numBlocks

        calculateScore<<<dimSWGrid, dimBlock>>>(
            d_diagnalLine, d_diagnalLinePre,
            d_row, d_col,
            x, y,
            numBlocks,
            d_sequences, d_references);
        cudaDeviceSynchronize();

        checkCudaErrors(cudaMemcpy(&h_diagnalLine[0], d_diagnalLinePre, sizeof(h_diagnalLine), cudaMemcpyDeviceToHost));

        LocalMatrix* temp = &*d_diagnalLinePre;
        d_diagnalLinePre = &*d_diagnalLine;
        d_diagnalLine = temp;


        for(int currBlockOrder = 0; currBlockOrder < numBlocks; currBlockOrder++){
          for(int currAlign = 0; currAlign < *(h_shared +dims3); currAlign++){
```

```
          int blockIdx_y = currBlockOrder * *(h_shared +dims3) + currAlign;
          int calgn = currAlign * *(h_shared +block_y_len) * *(h_shared +block_x_len);
          int blocky = (y + currBlockOrder) * *(h_shared +block_x_len);
          int blockx = (x - currBlockOrder);

          memcpy((h_scoringMatrix + calgn + blocky + blockx), (&h_diagnalLine[0]+blockIdx_y), sizeof(LocalMatrix));
        }
      }

      if (x == *(h_shared +block_x_len) - 1)
        ++y;
      if (x < *(h_shared +block_x_len) - 1)
        ++x;
  }
  cudaEventSynchronize(stop);
  cudaEventRecord(stop);
  cudaEventSynchronize(stop);
  float milliseconds = 0;
  cudaEventElapsedTime(&milliseconds, start, stop);
  printf("%3.1f,", milliseconds);
  cudaEventDestroy(start);
  cudaEventDestroy(stop);
  checkCudaErrors(cudaFree(d_col));
  checkCudaErrors(cudaFree(d_row));
  checkCudaErrors(cudaFree(d_sequences));
  checkCudaErrors(cudaFree(d_references));
  checkCudaErrors(cudaFree(d_diagnalLine));
  checkCudaErrors(cudaFree(d_diagnalLinePre));
  return h_scoringMatrix;

}

//2D to 1D array idx converter https://www.cyotek.com/blog/converting-2d-arrays-to-1d-and-accessing-as-either-2d-or-1d
__device__ int get1DIdx(int curr_chain, int curr_aln, int block_y, int block_x){
  return curr_chain * d_shared[dims3] * d_shared[block_y_len] * d_shared[block_x_len] + curr_aln * d_shared[block_y_len] * d_shared[block_x_len] + block_y * d_shared[block_x_len] + block_x;

}
/*this version is a speed up version
 * we need to access two lines, we call: the diagnalLine has been computated before, we call it as diagnalLine
*the line that is even more before: we call it as diagnalLinePre
*there are two LocalMatrix array. here is what they do:
*our calculation is seperated into two diagnals, (which both is needed for calculation)
*1. d_diagnalLine
*and the current one is with the previous information
*/
__global__ void calculateScore_v2(
    LocalMatrix *d_scoringMatrix,
    int *d_row, int *d_col,
    unsigned int x, unsigned int y,
    unsigned int numOfBlocks,
    uint8_t* d_sequences, uint8_t* d_references){
  unsigned int DIAGONAL = SHARED_Y + SHARED_X;
  int innerScore;
  /**
   * shared memory block for calculations. It requires
   * extra (+1 in both directions) space to hold
   * Neighboring cells
   */
  __shared__ int h_matrix[SHARED_Y+1][SHARED_X+1];
  __shared__ int e_matrix[SHARED_Y+1][SHARED_X+1];
  __shared__ int f_matrix[SHARED_Y+1][SHARED_X+1];
  //__shared__ int m_matrix[SHARED_Y][SHARED_X];
  __shared__ int s_maxima[SHARED_Y];
  __shared__ int x_maxloc[SHARED_Y];


  //if there are only one item, block y is the number of block we needed
  //x, y is our axises
  unsigned int currAlign = blockIdx.y%d_shared[dims3];//which alignment we
     are looking at
  unsigned int currBlockOrder = blockIdx.y/d_shared[dims3];//compute
   diagnally, the current MingCi of the block
  unsigned int blockx = x - currBlockOrder;//the block pos of x
  unsigned int blocky = y + currBlockOrder;//the block pos of y
  unsigned int tIDx = threadIdx.x;//current thread id of x
  unsigned int tIDy = threadIdx.y;//current thread id of y
  //unsigned int bIDx = blockIdx.x;//sequence id, 4th dim, always 1
  //unsigned int bIDy = currAlign;//bidy is considered as our alignemnt
   nubmer

  // indices of the current characters in both sequen
  int seqIdx = tIDx + currAlign * d_shared[max_x] + blockx * SHARED_X;//
   shorter read
  int refIdx = tIDy + currAlign * d_shared[max_y] + blocky * SHARED_Y;//
   longer ref

//initialization to the EPT_score
  int tmp_1 = (SHARED_Y+1)*(SHARED_X+1);

  memset(&h_matrix[0][0], EPT_SCORE, tmp_1*sizeof(int));
  memset(&e_matrix[0][0], EPT_SCORE, tmp_1*sizeof(int));
  memset(&f_matrix[0][0], EPT_SCORE, tmp_1*sizeof(int));
  memset(&s_maxima[0], EPT_SCORE, SHARED_Y*sizeof(int));
  memset(&x_maxloc[0], EPT_SCORE, SHARED_Y*sizeof(int));
  __syncthreads();

//first block row first row
//we have multiple blocks, therefore, we have to be very carefull
  if(!blocky && !tIDy) {//when tIDy is 0, which would be the first line
    h_matrix[0][tIDx] = d_row[seqIdx];
  }
  if(!blockx && !tIDx){//tIDx is 0, left column
    h_matrix[tIDy][0] = d_col[refIdx];
    //if(!tIDx&&!tIDy) h_matrix[0][SHARED_X] = d_col[refIdx]
  }
  //surrounded line that we have to copy them from computed d_row and
   d_col
  //blocky is > 0
  int idx = 0;
  if (blocky && !tIDy){
    //(x, y-1)
    idx = get1DIdx(0, currAlign,blocky-1,blockx);
    if(tIDx) h_matrix[0][tIDx] = d_scoringMatrix[idx].h_value[SHARED_Y-1][
   tIDx-1];
     f_matrix[0][tIDx] = d_scoringMatrix[idx].f_value[SHARED_Y-1][tIDx];//
   for restoring previous h

  }
  else if(blockx && !tIDx && tIDy){
    idx = get1DIdx(0, currAlign,blocky,blockx-1);
    //(x-1, y)
    h_matrix[tIDy][0] = d_scoringMatrix[idx].h_value[tIDy-1][SHARED_X-1];

  }

  if(blockx&& ! tIDx){
    idx = get1DIdx(0, currAlign,blocky,blockx-1);
    //(x-1, y)
     e_matrix[tIDy][0] = d_scoringMatrix[idx].e_value[tIDy];//for
   restoring previous e

  }
  if (blockx && blocky && ! tIDx && !tIDy){
    idx = get1DIdx(0, currAlign,blocky-1,blockx-1);
    //(x-1,y-1)
    h_matrix[0][0] = d_scoringMatrix[idx].h_value[SHARED_Y-1][SHARED_X-1];

  }


  /**
   * tXM1 and tYM1 are to store the current value of the thread Index.
   tIDx and tIDy are
   * both increased with 1 later on.
   */
  unsigned int tXM1 = tIDx;
  unsigned int tYM1 = tIDy;
  // shared location for the parts of the 2 sequences, for faster
   retrieval later on:
  __shared__ uint8_t s_seq[SHARED_X];
  __shared__ uint8_t s_ref[SHARED_Y];

  // copy sequence data to shared memory (shared is much faster than
   global)
  if (!tIDy){
    s_seq[tIDx] = d_sequences[seqIdx];
  }
  if (!tIDx){
    s_ref[tIDy] = d_references[refIdx];

  }
  __syncthreads();
  // set inner score (aka sequence match/mismatch score):
  uint8_t charSeq = s_seq[tIDx];
  uint8_t charRef = s_ref[tIDy];


  innerScore = charSeq == FILL_CHARACTER || charRef == FILL_CHARACTER ?
   FILL_SCORE : d_mat[charSeq+charRef*5];
  // transpose the index
  ++tIDx;
  ++tIDy;
  // set shared matrix to zero (starting point!)
  // wait until all elements have been copied to the shared memory block
  /**** sync barrier ****/

  __syncthreads();
```

```

    for (int i=0; i < DIAGONAL; ++i) {
    if(innerScore!=FILL_SCORE){
      if (i == tXM1+ tYM1) {
//        // calculate only when there are two valid characters
//        // this is necessary when the two sequences are not of equal
   length
//        // this is the SW-scoring of the cell:
//        // At the beginning of the loop: eh[j] = { H(i-1,j-1), E(i,j) },
    f = F(i,j) and h1 = H(i,j-1)
//        // Similar to SSE2-SW, cells are computed in the following order
   :
//        //   H(i,j)   = max{H(i-1,j-1)+S(i,j), E(i,j), F(i,j)}
//        //   E(i+1,j) = max{H(i,j)-gapo, E(i,j)} - gape
//        //   F(i,j+1) = max{H(i,j)-gapo, F(i,j)} - gape
        int M = h_matrix[tYM1][tXM1]? h_matrix[tYM1][tXM1] + innerScore:
   0;
        //m_matrix[tYM1][tXM1] = M;
        h_matrix[tIDy][tIDx] = max(max(M, e_matrix[tYM1][tXM1]), f_matrix[
   tYM1][tXM1]);
        e_matrix[tYM1][tIDx] = max(max(M-d_shared[oe_ins], e_matrix[tYM1][
   tXM1]-d_shared[e_ins]), 0);
        f_matrix[tIDy][tXM1] = max(max(M-d_shared[oe_del], f_matrix[tYM1][
   tXM1]-d_shared[e_del]), 0);

      }

    }

    if(i-1 == tXM1 + tYM1){
      if(!tXM1){
        s_maxima[tYM1] = h_matrix[tIDy][1];
        x_maxloc[tYM1] = tXM1;
      }
      else if(getHigher(h_matrix[tIDy][tIDx], s_maxima[tYM1], &s_maxima[
   tYM1]))
          x_maxloc[tYM1] = tXM1;

    }
    // wait until all threads have calculated their new score
      /**** sync barrier ****/
    __syncthreads();
  }



  //pass on the information to the next block
  //here we modify for our diagnalLine
  //int idx = get1DIdx(blockx, blocky, XdivSHARED_X);
  idx = get1DIdx(0, currAlign,blocky,blockx);
  d_scoringMatrix[idx].h_value[tYM1][tXM1] = h_matrix[tIDy][tIDx];
  d_scoringMatrix[idx].f_value[tYM1][tXM1] = f_matrix[tIDy][tXM1];

  //stored for next time computation
  if(!tXM1){
    d_scoringMatrix[idx].e_value[tYM1]=e_matrix[tYM1][SHARED_X];
    d_scoringMatrix[idx].s_value[tYM1]=s_maxima[tYM1];
    d_scoringMatrix[idx].x_value[tYM1]=x_maxloc[tYM1];

  }
  __syncthreads();

```

```
}

/*
*sw_ext *sw_ext: this is our array of extension information
*int num_seq: number if sequence
*int num_target: number of targets
*this is our converter, we basically convert everything and get ready
    thing to be ready
*/
/*this method pass out a pointer of 2D array type*/
extern "C" LocalMatrix *gpu_sw_seed_extend_v2(sw_ext *swext, int *h_shared
    , int8_t *mat){



  int *d_col = 0, *d_row = 0;
  uint8_t *d_sequences = 0, *d_references = 0;
  *(h_shared +oe_del) = *(h_shared +o_del) + *(h_shared +e_del);
  *(h_shared +oe_ins) = *(h_shared +o_ins) + *(h_shared +e_ins);
  //[1]search for max_qlen and max_tlen for all element
  //we are looking for the max qlen and max tlen
  *(h_shared +max_qlen) = swext->qlen;
  *(h_shared +max_tlen) = swext->tlen;
  //starts from 1
  for(int i = 1; i < *(h_shared + dims3); i++){//skip the first one
     if((swext+i)->qlen > *(h_shared +max_qlen)) *(h_shared +max_qlen) = (
    swext+i)->qlen;
     if((swext+i)->tlen > *(h_shared +max_tlen)) *(h_shared +max_tlen) = (
    swext+i)->tlen;

  }


  *(h_shared +block_x_len) = (int) ceil((double)*(h_shared +max_qlen)/
    SHARED_X);//how many 8*8 block on x div
  *(h_shared +block_y_len) = (int) ceil((double)*(h_shared +max_tlen)/
    SHARED_Y);//how many 8*8 block on y div
  *(h_shared +max_x) = *(h_shared +block_x_len) * SHARED_X;//for seed
    extension part
  *(h_shared +max_y) = *(h_shared +block_y_len) * SHARED_Y;//reference
    extension part
  *(h_shared +alignment_x) = *(h_shared +max_x) * *(h_shared +dims3);//
    TOTAL LENGTH OF X
  *(h_shared +alignment_y) = *(h_shared +max_y) * *(h_shared +dims3);//
    TOTAL LENGTH OF Y
  *(h_shared +block_diagnal_len) = max(*(h_shared +block_x_len), *(
    h_shared +block_y_len));//TOTAL LENGTH OF Y
  *(h_shared +alignment_diagnal_len) = *(h_shared +block_diagnal_len)**(
    h_shared +dims3);//TOTAL LENGTH OF Y

  int h_row[*(h_shared +alignment_x)];
  int h_col[*(h_shared +alignment_y)];
  uint8_t h_seq[*( h_shared +alignment_x)];
  uint8_t h_ref[*( h_shared +alignment_y)];
  LocalMatrix t_sc[*(h_shared +dims4)][*(h_shared +dims3)][*(h_shared +
    block_y_len)][*(h_shared +block_x_len)];
  //LocalMatrix *h_scoringMatrix = (LocalMatrix*) calloc(sizeof(
    LocalMatrix), *(h_shared +dims4) *  *(h_shared +dims3) * *(h_shared +
    block_y_len) * *(h_shared +block_x_len));
  LocalMatrix *h_scoringMatrix = &t_sc[0][0][0][0];
  LocalMatrix *d_scoringMatrix;
  //initialization is done here.
  //[2] initialization of top and left row

  //we are initializing for everyone
  //init h_row, h_col
  for(int align_idx = 0; align_idx < *(h_shared +dims3); align_idx++){//go though each alignment option here
     int start_x_pos = align_idx * *( h_shared +max_x);//ours 403//starting position
     int start_y_pos = align_idx * *( h_shared +max_y);//current alignment
     struct sw_ext *curr_sw_ext  = swext + align_idx;
     int curr_h0 = curr_sw_ext->sc0;


     h_row[start_x_pos+0] = curr_h0;//position is 0
     h_row[start_x_pos+1] = LIKELY(curr_h0 > *(h_shared +oe_ins))? curr_h0 - *(h_shared +oe_ins) : 0;//404, position as 1
     for(int curr_x_loc = start_x_pos + 2; LIKELY(curr_x_loc < start_x_pos + *(h_shared +max_x)); ++curr_x_loc)
       h_row[curr_x_loc] = (curr_x_loc <= (start_x_pos + curr_sw_ext->qlen) && h_row[curr_x_loc - 1] > *(h_shared +e_ins))? h_row[curr_x_loc - 1] - *(h_shared +e_ins) : 0;

     // adjust $w if it is too large
     // generate the first row
     h_col[start_y_pos+0] = curr_h0; //eh[0].e = highest possible score
     h_col[start_y_pos + 1] = LIKELY(curr_h0 > *(h_shared +oe_del)) ? curr_h0 - *(h_shared +oe_del) : 0;
     for (int curr_y_loc =start_y_pos + 2; LIKELY(curr_y_loc < start_y_pos + *(h_shared +max_y)); ++curr_y_loc)
       h_col[curr_y_loc] = (curr_y_loc <= (start_y_pos + curr_sw_ext->tlen) && h_col[curr_y_loc - 1] > *(h_shared +e_del))? h_col[curr_y_loc - 1] - *(h_shared +e_del) : 0;

     for(int curr_x_loc = start_x_pos; LIKELY(curr_x_loc < start_x_pos + *(h_shared +max_x)); ++curr_x_loc)
       h_seq[curr_x_loc] = LIKELY(curr_x_loc < start_x_pos + curr_sw_ext->qlen)? curr_sw_ext->query[curr_x_loc-start_x_pos] : FILL_CHARACTER;

     for(int curr_y_loc = start_y_pos; LIKELY(curr_y_loc < start_y_pos + *(h_shared +max_y)); ++curr_y_loc)
       h_ref[curr_y_loc] = LIKELY(curr_y_loc < start_y_pos + curr_sw_ext->tlen)? curr_sw_ext->target[curr_y_loc-start_y_pos] : FILL_CHARACTER;


     curr_sw_ext->h_col = &h_col[start_y_pos];
  }
  checkCudaErrors(cudaMalloc((void**)&d_scoringMatrix, sizeof(t_sc)));

  checkCudaErrors(cudaMalloc((void **)&d_row, sizeof(h_row)));
  checkCudaErrors(cudaMalloc((void **)&d_col, sizeof(h_col)));
  checkCudaErrors(cudaMemcpy(d_row, &h_row[0], sizeof(h_row), cudaMemcpyHostToDevice));
  checkCudaErrors(cudaMemcpy(d_col, &h_col[0], sizeof(h_col), cudaMemcpyHostToDevice));

  checkCudaErrors(cudaMalloc((void** ) &d_sequences,sizeof(h_seq)));
  checkCudaErrors(cudaMalloc((void** ) &d_references,sizeof(h_ref)));
  checkCudaErrors(cudaMemcpy(d_sequences, &h_seq[0], sizeof(h_seq), cudaMemcpyHostToDevice));
  checkCudaErrors(cudaMemcpy(d_references, &h_ref[0], sizeof(h_ref), cudaMemcpyHostToDevice));

```

```

checkCudaErrors(cudaMemcpyToSymbol(d_mat, mat, 25 * sizeof(int8_t),0,
 cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpyToSymbol(d_shared, h_shared, 21 * sizeof(
 unsigned int),0, cudaMemcpyHostToDevice));

unsigned int halfZhouChang =  *(h_shared +block_x_len) + *(h_shared +
 block_y_len); //half zhouchang of a matrix, width and hight
unsigned int xiaoBian = halfZhouChang - *(h_shared + block_diagnal_len);
 //the width, xiaoBian, the smaller one also it is the Max number of
 blocks
unsigned int daBian = *(h_shared + block_diagnal_len);//the hight,
 DaBian, the longer one, also it is the "starting to decrease at
unsigned int numBlocks = 0, x = 0, y = 0;

dim3 dimBlock(SHARED_X, SHARED_Y, 1);



cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
// adjust $w if it is too large
  //locate memory for d_eh
  for (unsigned int i = 1; LIKELY(i < halfZhouChang); ++i) {
    numBlocks = i <= xiaoBian? i : i >= daBian? halfZhouChang - i :
 xiaoBian;

    //reserve dim4 for anything beyound chain
    dim3 dimSWGrid(*(h_shared +dims4), *(h_shared +dims3) * numBlocks,
 1);//numBlocks

    calculateScore_v2<<<dimSWGrid, dimBlock>>>(
        d_scoringMatrix,
        d_row, d_col,
        x, y,
        numBlocks,
        d_sequences, d_references);

    cudaDeviceSynchronize();

    if (x == *(h_shared +block_x_len) - 1)
      ++y;
    if (x < *(h_shared +block_x_len) - 1)
      ++x;
}
checkCudaErrors(cudaMemcpy(h_scoringMatrix, d_scoringMatrix, sizeof(t_sc
 ), cudaMemcpyDeviceToHost));

cudaEventSynchronize(stop);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
printf("%3.1f,", milliseconds);
cudaEventDestroy(start);
cudaEventDestroy(stop);
checkCudaErrors(cudaFree(d_col));
checkCudaErrors(cudaFree(d_row));
checkCudaErrors(cudaFree(d_sequences));
checkCudaErrors(cudaFree(d_references));
```

```
  checkCudaErrors(cudaFree(d_scoringMatrix));
  return h_scoringMatrix;

}
```

Listing C.6 smithwaterman.cu

C.7 ksw_extend2CPU.c

```
/********************
 *** SW extension ***
 ********************/
/**
 * [0]start pos. of seed on query
 * [1]query seed
 * [2]tmp = rbeg-rmax[0], the length of left ref
 * [3]ref
 * [4]tlen
 * [5]scoring matrix
 * [6]deleting cost, gap open penalties for deletions
 * [7]deleting cost, gap extension penalties.
 * [8]insertion cost, gap open penalties for insertions
 * [9]insertion cost, gap insertion penalties.
 * [10]actuall bandwidth
 *    Band width. Essentially, gaps longer than INT will not be found. Note that the maximum gap length is also affected by the scoring matrix and the hit length, not solely determined by this option.
 * [11]this is end_bonus. not clipping panelly. clipping panelly: Penalty for 5'- and 3'-end clipping. When performing the Smith-Waterman extension of the seed alignments,
 *  BWA-MEM keeps track of the best score reaching the end of the read. If this score is larger than the best SW score minus the clipping penalty,
 *   clipping will not be applied (BWA MEM option -L).
 * [12]drop off, off-diagonal X-dropoff
 * [13]max matching score
 *
 * [14]length of the query in the alignment, the best global alignment in the query
 * [15]length of the target in the alignment, the best global alignemtn in the reference
 * [16]length of the target if query is fully aligned, query's target length after full alignment
 * [17]score of the best end to end alignemtn, query's full alignment score
 * [18]max off diagonal dist, the best score, the query and reference position's difference.
 */

/**
 * how the code is running
 * input: rbeg[739080], len[25], l_query[35], qbeg[1], rmax[1]-739118, rmax[0]-739078
 * input: [del: 1, o_ins: 6, e_ins: 1, match_a: 1, misma_b: 4, zdrop: 100, pen_clip5: 5, pen_clip3 5]
 ****currRef:TTATCCTATTACATTATCAATCCTTGCATTTCAGCTTCTT Length = rmax[1]-rmax[0] = 739118 -739078 = 40
 ****currSed:  ATCCTATTACATTATCAATCCTTGC         Length = s->len = 25
 ****currQue: GATCCTATTACATTATCAATCCTTGCGTTTCAGCT
 ****leftRef:TT
 ****leftQue: G
 ****rihtRef:                              ATTTCAGCTTCTT Length = 13
 ****rihtQue:                              GTTTCAGCT   length = 9
 *then our scoring matrix
 ****[0A], [1C], [2G], [3T], [4N],
 ***A[ 1], [-4], [-4], [-4], [-1],
 ***C[-4], [ 1], [-4], [-4], [-1],
 ***G[-4], [-4], [ 1], [-4], [-1],
 ***T[-4], [-4], [-4], [ 1], [-1],
 ***N[-1], [-1], [-1], [-1], [-1],
 ***
 *then we calculate so called query profile
 ****[ G], [ T], [ T], [ T], [ C], [ A], [ G], [ C], [ T],
 ***A[-4], [-4], [-4], [-4], [-4], [ 1], [-4], [-4], [-4],
 ***C[-4], [-4], [-4], [-4], [ 1], [-4], [-4], [ 1], [-4],
 ***G[ 1], [-4], [-4], [-4], [-4], [-4], [ 1], [-4], [-4],
 ***T[-4], [ 1], [ 1], [ 1], [-4], [-4], [-4], [-4], [ 1],
 ***N[-1], [-1], [-1], [-1], [-1], [-1], [-1], [-1], [-1],
 *
 *Then
 *Then
 */
typedef struct {
  int32_t h, //highest possible score
  e;//
} eh_t;

int ksw_extend2(int qlen, const uint8_t *query, int tlen, const uint8_t *
    target, int m, const int8_t *mat, int o_del, int e_del, int o_ins, int
    e_ins, int w, int end_bonus, int zdrop, int h0, int *_qle, int *_tle,
    int *_gtle, int *_gscore, int *_max_off)
{ int dgl = 0;
  if(qlen==9) dgl = 3;
  if(dgl >= 2) printf("**++KSW_extend++**\n");
  eh_t *eh; // score array
  int8_t *qp; // query profile
  int i, j, k, ssss, oe_del = o_del + e_del,//opening and extending
      oe_ins = o_ins + e_ins,
      beg, end,//beg, end, starting and ending position
      max, max_i, max_j, max_ins, max_del, max_ie,
      gscore, max_off;
  assert(h0 > 0);
  // allocate memory
  qp = malloc(qlen * m); //m is 5, I think, this means a, t, c, g, n
  eh = calloc(qlen + 1, 8);//keep tracking all h and e
  if(dgl >= 2) printf("--> gen query profile: \n");
  // generate the query profile
  for (k = i = 0; k < m; ++k) {
    const int8_t *p = &mat[k * m];
    if(dgl >= 2) printf("%d\n",k);
    for (j = 0; j < qlen; ++j) {
      qp[i++] = p[query[j]];
      if(dgl >= 2) printf("%d[%d],",query[j], p[query[j]]);
    }
    putchar('\n');
  }

  // set up highest possible score for each row
  eh[0].h = h0; //eh[0].h = highest possible score
  eh[1].h = h0 > oe_ins? h0 - oe_ins : 0;

  if(dgl >= 2) printf("**eh :");
  for (j = 2; j <= qlen && eh[j-1].h > e_ins; ++j){
    eh[j].h = eh[j-1].h - e_ins;
    if(dgl >= 2) printf("[%d]", eh[j].h);
  }
  putchar('\n');
  // adjust $w if it is too large
  k = m * m;


  for (i = 0, max = 0; i < k; ++i) // get the max score in mat.
    max = max > mat[i]? max : mat[i];

  //qlen, assume is 10. the max score in the mat is 1. the end bonus assume is 2. so it would be (qlen*1+5-6)/1+1
  max_ins = (int)((double)(qlen * max + end_bonus - o_ins) / e_ins + 1.);//get max insertion score
  max_ins = max_ins > 1? max_ins : 1;
  if(dgl >= 2) printf("***max_ins: %d\n", max_ins);
  w = w < max_ins? w : max_ins;//calculate max band width? which is defined as the max_ins in current situation.
  max_del = (int)((double)(qlen * max + end_bonus - o_del) / e_del + 1.);//get max deletion score
  max_del = max_del > 1? max_del : 1;
  if(dgl >= 2) printf("***max_del: %d\n", max_del);
  w = w < max_del? w : max_del; // TODO: is this necessary?//ok, I think the max bandwidth is the max insertion/deletion score you can get.
  // DP loop
  max = h0, max_i = max_j = -1; max_ie = -1, gscore = -1;
  max_off = 0;
  beg = 0, end = qlen;
  //go though each ref.
  for (i = 0; LIKELY(i < tlen); ++i) {
    if(dgl >= 3) {printf("**[1]eh.e :"); for (ssss = 0; ssss <= qlen; ++ssss){ printf("[%d]", eh[ssss].e);} putchar('\n');}


    if(dgl>=2) printf("-> start calc: [ID: %d, ", i);
    int t, f = 0, h1, m = 0, mj = -1;
    if(dgl>=2) {
      printf(" target: %d, ", target[i]);
      putchar("ACGTN"[(int)query[i]]);
      printf(", ");
    }

    int8_t *q = &qp[target[i] * qlen];//get the score
    // apply the bandwidth and the constraint (if provided)
    if (beg < i - w) beg = i - w;//>=o, make sure it is not over the max scores, or band width
    if (end > i + w + 1) end = i + w + 1;//same thing
    if (end > qlen) end = qlen;

    // compute the first column
    if (beg == 0) {//if we are, say, beg is 0, i-w<0,
      //h0 is the highest possible score. h1 is the highest score with one o_del and i+1's e_del
      h1 = h0 - (o_del + e_del * (i + 1));
      if (h1 < 0) h1 = 0;//if it is smaller than 0, set to zero
    } else h1 = 0;//else we just say they are 0.
    if(dgl>=2) printf("h1: %d, h0: %d, beg: %d, end: %d]\n",h1, h0, beg, end);
    //up we pre calc everything
    for (j = beg; LIKELY(j < end); ++j) {
      if(dgl>=3) printf("--->query's element [ID: %d, h1: %d, mj<mzsclr>: %d, m: %d]",j, h1, mj, m);
      // At the beginning of the loop: eh[j] = { H(i-1,j-1), E(i,j) }, f = F(i,j) and h1 = H(i,j-1)
      // Similar to SSE2-SW, cells are computed in the following order:
      //   H(i,j)   = max{H(i-1,j-1)+S(i,j), E(i,j), F(i,j)}
```

```c
      //   E(i+1,j) = max{H(i,j)-gapo, E(i,j)} - gape
      //   F(i,j+1) = max{H(i,j)-gapo, F(i,j)} - gape
      eh_t *p = &eh[j];
      int h, M = p->h, e = p->e; // get H(i-1,j-1) and E(i-1,j)
      if(dgl>=3) printf("001 [h: %d, M: %d, e: %d]", h, M, e);
      p->h = h1;          // set H(i,j-1) for the next row
      M = M? M + q[j] : 0;// separating H and M to disallow a cigar like "100M3I3D20M"
      h = M > e? M : e;   // e and f are guaranteed to be non-negative, so h>=0 even if M<0
      h = h > f? h : f;
      if(dgl>=3) printf("002 [h: %d, M: %d, e: %d, q[%d]:%d]", h, M, e, j, q[j]);

      h1 = h;             // save H(i,j) to h1 for the next column
      mj = m > h? mj : j; // record the position where max score is achieved
      m = m > h? m : h;   // m is stored at eh[mj+1]
      t = M - oe_del;
      if(dgl>=3) printf("003 [t: %d]", t);
      t = t > 0? t : 0;
      e -= e_del;
      e = e > t? e : t;   // computed E(i+1,j)
      p->e = e;           // save E(i+1,j) for the next row
      t = M - oe_ins;
      t = t > 0? t : 0;
      f -= e_ins;
      f = f > t? f : t;   // computed F(i,j+1)
      if(dgl>=3) printf("004 [h: %d, M: %d, e: %d, t: %d, f: %d]", h, M, e, t, f);
      if(dgl>=3) putchar('\n');
    }
    if(dgl >= 3) {printf("**[.05)eh.e :"); for (ssss = 0; ssss <= qlen; ++ssss){ printf("[%d]", eh[ssss].e);} putchar('\n');}

    eh[end].h = h1; eh[end].e = 0;
    if(dgl>=2) printf("-> back to normal: ");

    if(dgl>=2) printf("001 [max_ie: %d, gscore: %d, h1: %d, i: %d]", max_ie, gscore, h1, i);
    if (j == qlen) {
      max_ie = gscore > h1? max_ie : i;
      gscore = gscore > h1? gscore : h1;
    }
    if(dgl>=2) printf("002 [max_ie: %d, gscore: %d, h1: %d, i: %d]", max_ie, gscore, h1, i);

    if (m == 0) break;
    if (m > max) {
      max = m, max_i = i, max_j = mj;
      max_off = max_off > abs(mj - i)? max_off : abs(mj - i);
    } else if (zdrop > 0) {
      if (i - max_i > mj - max_j) {
        if (max - m - ((i - max_i) - (mj - max_j)) * e_del > zdrop) break;
      } else {
        if (max - m - ((mj - max_j) - (i - max_i)) * e_ins > zdrop) break;
      }
    }
    if(dgl >= 3) {printf("**[2]eh.e :"); for (ssss = 0; ssss <= qlen; ++ssss){ printf("[%d]", eh[ssss].e);} putchar('\n');}

    // update beg and end for the next round
    for (j = beg; LIKELY(j < end) && eh[j].h == 0 && eh[j].e == 0; ++j);
    beg = j;
    for (j = end; LIKELY(j >= beg) && eh[j].h == 0 && eh[j].e == 0; --j);
    end = j + 2 < qlen? j + 2 : qlen;
    if(dgl >= 3) {printf("**[3]eh.e :"); for (ssss = 0; ssss <= qlen; ++
   ssss){ printf("[%d]", eh[ssss].e);} putchar('\n');}

    //beg = 0; end = qlen; // uncomment this line for debugging
    if(dgl>=2) printf("003 [max: %d, max_i: %d, max_j: %d]", max, max_i,
   max_j);

    if(dgl>=2) putchar('\n');

  }
  free(eh); free(qp);
  if (_qle) *_qle = max_j + 1;//max query's position
  if (_tle) *_tle = max_i + 1;//max reference's position
  if (_gtle) *_gtle = max_ie + 1;//max_ie,
  if (_gscore) *_gscore = gscore;//best score
  if (_max_off) *_max_off = max_off;
  if(dgl>=2) printf(">>>>>>result report: 001 [qle: %d, tle: %d, gtle: %d,
     gscore: %d, max_off: %d]", *_qle, *_tle, *_gtle, *_gscore, *_max_off);
  if(dgl>=2) printf("002 [max_j: %d, max_i: %d, max_ie: %d, gscore: %d,
   max_off: %d]", max_i, max_j, max_ie, gscore, max_off);
  if(dgl>=2) putchar('\n');
  return max;
}

int ksw_extend(int qlen, const uint8_t *query, int tlen, const uint8_t *
   target, int m, const int8_t *mat, int gapo, int gape, int w,
    int end_bonus, int zdrop, int h0, int *qle, int *tle, int *gtle, int *
   gscore, int *max_off)
{
  return ksw_extend2(qlen, query, tlen, target, m, mat, gapo, gape, gapo,
   gape, w, end_bonus,
                      zdrop, h0, qle, tle, gtle, gscore, max_off);
}
```

Listing C.7 ksw_extend2CPU.c