Optimising Bisulfite Sequencing Analysis

Author
Thomas Kaplan

Supervisors
James Arram
Prof. Wayne Luk
Peter Rice

16th June, 2015

Abstract

DNA methylation is an epigenetic process that is key to numerous cellular phenomena including embryonic development and disease. With next-generation bisulfite sequencing it is now possible to conduct whole-genome methylation analysis at single base resolution. This has recently been used in developing novel methods for cancer and non-invasive prenatal diagnosis. The challenge faced in this research is that the throughput of next-generation bisulfite sequencing machines has been improving at a faster rate than Moore's law. This poses a significant computational and storage challenge for genomic analysis tools. In this report we optimise key bottlenecks of Methy-Pipe, an integrated bioinformatics pipeline developed for bisulfite sequencing alignment and methylation analysis. The contributions of our work include: FPGA and GPU optimisation of the bisulfite sequencing alignment module based on a novel oversampling method, a high throughput referential compression algorithm using hardware accelerated sequence alignment, software optimisation of the methylation calling module and translation of a downstream analysis script. Results indicate that the runtime of Methy-Pipe's bottlenecks is reduced from 5.5 hours to 22 minutes, which could allow potentially life-saving diagnosis techniques to become routine in healthcare applications.

Acknowledgements

I would like to thank:

- Professor Wayne Luk[†] for providing continual guidance and support during this project, including his decision to send me on a Maxeler training day.
- Peter Rice[†] for his support, and for providing a comprehensive introduction to sequencing, sequence analysis and functional genomics.
- James Arram[†] for his invaluable guidance, feedback, and close collaboration in the optimisation of Methy-Pipe's alignment module. This project would not have been possible without his existing work on the acceleration of short-read alignment, namely the Ramethy runtime-reconfigurable architecture.
- Moritz Pflanzer[†] for his novel comments on the use of hardware-accelerated sequence alignment in referential compression of genetic sequencing data, and his support in collecting referential compression benchmarks.
- Dr Peiyong Jiang[‡] for his insights into Methy-Pipe and his ongoing collaboration with the Custom Computing Research Group at Imperial College London.

[†]Department of Computing
Imperial College London
{wl/jma11/p.rice}@imperial.ac.uk

[‡]Department of Chemical Pathology
The Chinese University of Hong Kong
jiangpeiyong@cuhk.edu.hk

Contents

Chapter 1

Introduction

1.1 Motivation

DNA sequencing involves identifying the order of nucleotide bases, our genetic building blocks, in a DNA molecule. Knowledge of DNA sequences is invaluable for biological research, medical diagnosis, forensic biology, and other applied fields. For example, within medical research and diagnosis, sequencing can be used to shed light on DNA methylation patterns which contribute to an array of human diseases. DNA methylation is a common biochemical process in which a methyl group reacts with a cytosine nucleotide, altering our genetic instructions in a potentially harmful manner. Next-generation sequencing (NGS) machines are able to inexpensively produce DNA sequences (reads) with an incredibly high throughput. We can examine the methylation of cytosine bases at single-base resolution using next-generate whole-genome bisulfite sequencing, in which NGS machines are used with chemically treated DNA. However, NGS machines have been improving at a faster rate than Moore's law, posing a significant computational challenge for tools intending to analyse the sequencing data produced. This is shown in F.1.

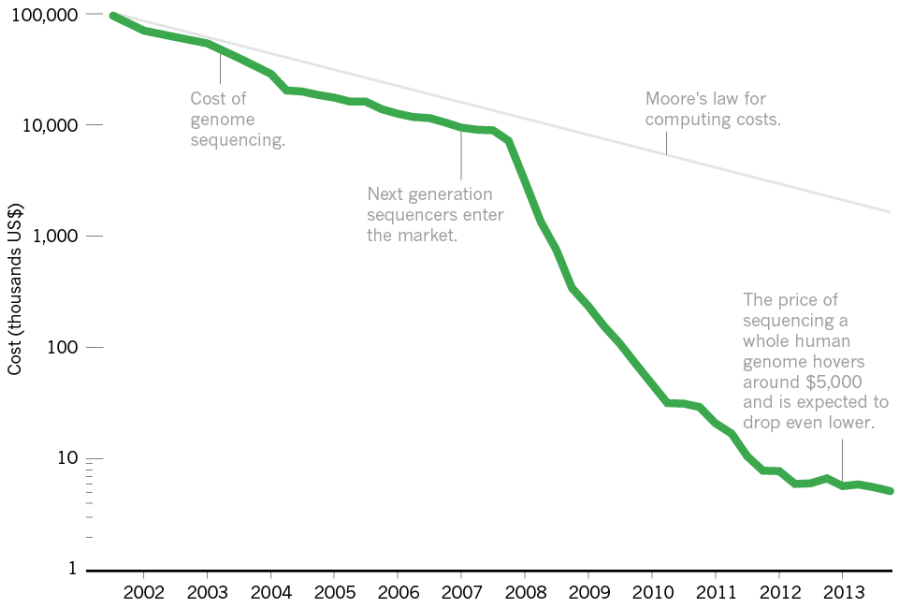


Figure F.1: Sequencing costs relationship to Moore's law[1].

Methy-Pipe[2] is an integrated bioinformatics pipeline for whole-genome methylation analysis. It not only fulfills the core methylation data analysis requirements (of sequence alignment, differential methylation analysis etc.) but also provides numerous tools for methylation data annotation and visualisation. Methy-Pipe contains two modules, BSAligner and BSAnalyzer, which perform sequence alignment and analysis respectively. These are illsutrated in F.2. Due to its significant functionality and ease of use, Methy-Pipe is an invaluable tool for both researchers and clinical scientists. The primary applications of Methy-Pipe are currently non-invasive methylation analysis of the fetus[3] and tumors[4] using plasma DNA bisulfite sequencing.

NGS machines produce short reads from a DNA sample destructively, so the DNA must be reconstructed by determining the location of the short reads in a known reference genome, using sequence alignment. For the alignment of bisulfite sequencing data, un-methylated cytosine bases transform into thymine bases. This means tools such as Methy-Pipe are faced with a

greater computational problem: reads must be aligned to the reverse complement of the reference genome, alongside the cytosine-depleted reference genome. Compared to more general alignment platforms, this effectively doubles the alignment time. It is only following alignment that high-resolution analysis can take place, which is also computationally demanding.

Even with state-of-the-art tools, solely aligning 300 million short bisulfite sequencing reads takes roughly 5 hours, using a system with dual 12-core Intel Xeon processors and 100GB RAM. Subsequent analysis could also take hours if workload is not distributed across many processors. **To shorten diagnosis and response times, in turn increasing patients seen and allowing potentially life-saving diagnosis techniques to become routine in healthcare procedures, Methy-pipe requires acceleration of its alignment and analysis modules**.

(a) BSAligner Module (b) BSAnalyzer Module

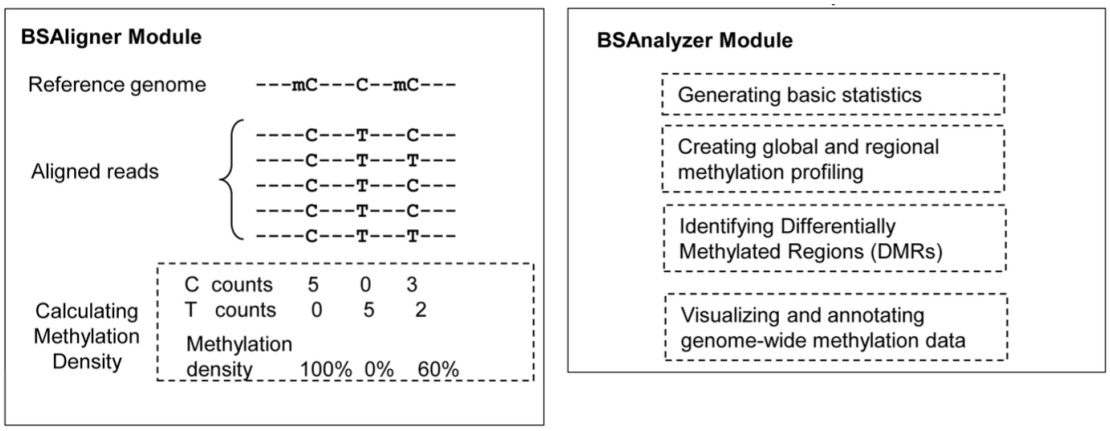


Figure F.2: Methy-Pipe Modules, BSAligner and BSAnalyzer[2].

1.2 Contributions

Methy-Pipe is composed of two modules, BSAligner and BSAnalyzer, which are illustrated in F.2a and F.2b respectively. BSAligner performs sequence alignment following bisulfite sequencing of DNA, whereas BSAnalyzer performs downstream analysis which provides insights into the methylation of the aligned DNA samples. We propose contributions targeting both of these modules:

1. Optimisations targeting BSAligner:

   (a) We optimise a novel runtime-reconfigurable bisulfite sequencing alignment design that uses the FM-index, which has been accelerated using FPGAs. We present optimisations to minimise off-chip accesses to DRAM: (1) novel oversampling of the FM-index, (2) a partial suffix-array interval store located in BRAM and (3) a novel seed and comparison module which limits use of an inefficient inexact alignment module. The design reduces the overall alignment time of BSAlign from 5 hours to roughly 13 minutes, corresponding to a significant speed-up of 22.7 times. The design is highly competitive with other novel FPGA-based alignment designs, aligning 144.2M bases per second.

   (b) We implement the first stage of the same bisulfite sequencing alignment design using GPUs, which involves a novel use of FM-index oversampling to reduce inefficient loads from CUDA global memory. The motivation behind this is to assess the potential contribution of GPU-based alignment to a novel heterogeneous bisulfite sequencing analysis platform using GPUs and FPGAs. The design outperforms state-of-the-art GPU-based sequence aligners SOAP3-dp[5] and BarraCUDA[6], by 2.8 and 6 times respectively. Projections suggest a full implementation could align BSAlign's typical workload in roughly 42 minutes, corresponding to a significant speed-up of 7.1 times.

2. Optimisations targeting BSAnalyzer: we present software optimisations targeting downstream analysis scripts, and the C++ module responsible for methylation calling (the process proceeding all downstream analysis). We have demonstrated how performance optimisations targeting the calling module, including novel parallelisation of file output, could reduce runtime from 30 minutes to 8.8 minutes. This module had a significant memory footprint, so we also reduce the memory usage by almost 52%, from 23.39GB to 11.3GB, resulting in a runtime of 33 minutes. This is achieved using a lightweight hash-associative container, Google's `sparse_hash`[7], alongside genome bit compression. We also translate one of the many Perl scripts used by BSAnalyzer into parallelised C. The module is responsible for reporting the methylation of each chromosome in the genome, a basic regional statistic. We achieve an improvement in run-time by 45 times, corresponding to a fall in run-time from over 3 hours to around 5 minutes.

3. Additionally, we present a novel approach to lossless referential compression of genetic sequencing data. This approach leverages the alignment algorithms from chapters 4 and 5, and we create novel FPGA and GPU-based designs. With a simple alignment algorithm, we achieve a significantly greater compression speed than referential compressor GDC 2[8]. For example, in the case of compressing 10M bisulfite sequencing reads from a human genome, our FPGA-based design runs 401 times faster than GDC 2, at 3.95 seconds. For this data set, our design achieves a compression ratio of 2.20, however GDC 2 with its superior compression algorithm achieved 5.94. Following a few small extensions that could dramatically improve the compression ratio, our approach to compression could consequently be utilised on hosts running Methy-Pipe to counter the increasingly demanding storage requirements of patients' bisulfite sequencing data.

1.3 Report Structure

In chapter 4 we optimise a runtime-reconfigurable bisulfite sequencing alignment design, and explore the feasibility of this design using GPUs in chapter 5. In chapter 6 we apply the previously explored alignment algorithms to hardware-accelerated compression of genetic sequencing data. Finally, in chapter 7 we present software optimisations targeting the methylation calling and downstream analysis modules.

The work presented in chapter 4 is also presented in [9] by J. Arram, T. Kaplan, W. Luk and P. Jiang (Department of Computing, Imperial College London and Department of Chemical Pathology, The Chinese University of Hong Kong). It has been submitted to IEEE Transactions on Computational Biology and Bioinformatics for publication.

Chapter 2

Background

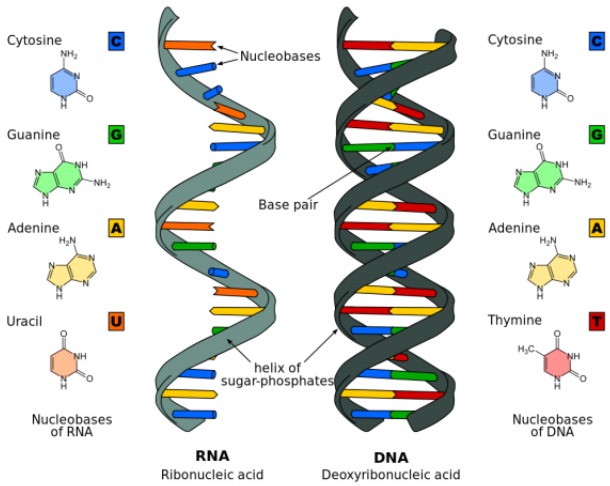
In this chapter we present the reader with an interdisciplinary overview of sequence analysis. We introduce the process of DNA methylation, before discussing the DNA sequencing and sequencing alignment techniques required to perform detailed methylation analysis. It should become evident that genetic information is being collected at an extraordinary pace by next-generation sequencing machines, putting pressure on bisulfite sequencing analysis tools such as Methy-Pipe. This reduces the utility of tools which could otherwise contribute to the research and diagnosis of many human diseases. To hint at a potential solution to this problem, we outline various methods of hardware-acceleration along with their respective merits.

2.1 Genomics

Genomics is the branch of molecular biology that studies the function, structure and other properties of genomes. The genome is an organism's complete set of deoxyribonucleic acid (DNA), and can be found in every cell of the human body excluding blood cells. It encodes our genetic information, which directs the development of proteins throughout our body. The genome is composed of 23 chromosomes, which can be seen as organised units of DNA. The genome is further divided into segments called genes, which are logical groups of information mapping to specific positions within the chromosomes.

As illustrated in F.1, DNA is a two stranded structure in the shape of a double helix. The two strands are often referred to as the 'Watson' and 'Crick'[10] strands in literature. These strands are connected by pairs of monomers, called nucleotides. These monomors are composed of nucleobases and pentose sugar (sugar with 5 carbon atoms). There are four nucleobases in DNA: cytosine, guanine, adenine and thymine. These nucleobases are often referred to as C, G, A and T respectively. The strands are reverse complements of one another, in the sense that one is the reverse of the other, and the bases form pairs of adenine-thymine and cytosine-guanine. Any change in the sequence of these nucleobases is described as a mutation, and can cause cellular disease, death, or alternatively have no effect. These mutations may be inherited.

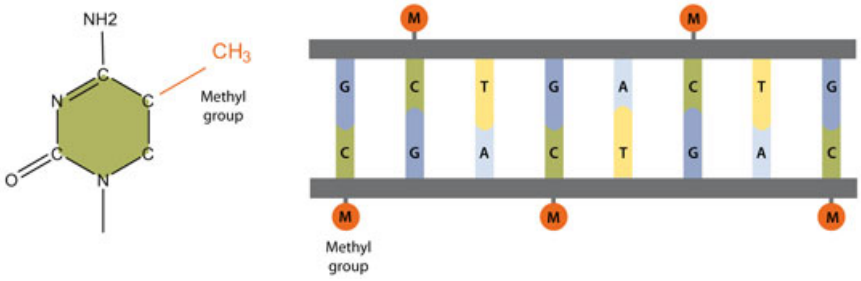
Figure F.1: Nucleic acids and their composition[11].



2.1.1 DNA methylation

DNA methylation is a common biochemical process in which a methyl group (carbon atom bonded to three hydrogen atoms) reacts with a cytosine or adenine nucleotide, more specifically the fifth carbon atom of a cytosine base or the sixth nitrogen atom of an adenine base. This clearly modifies the DNA, altering the genes expressed in cells stably, as they divide and develop into proteins and particular tissues from embryonic stem cells. DNA methylation is an epigenetic process, which means although the DNA is modified, it is a heritable change that does not influence the actual DNA sequence. It can be thought of as a pattern which overlays the DNA sequence. The specific mechanism by which methylation of cytosine nucleobases influences organisms is that by transforming our genetic information, transcription factors (proteins bound to a specific part of DNA) cannot access the elements that regulate them.

Figure F.2: DNA Methylation - the introduction of a methyl group to nucleotides[12].



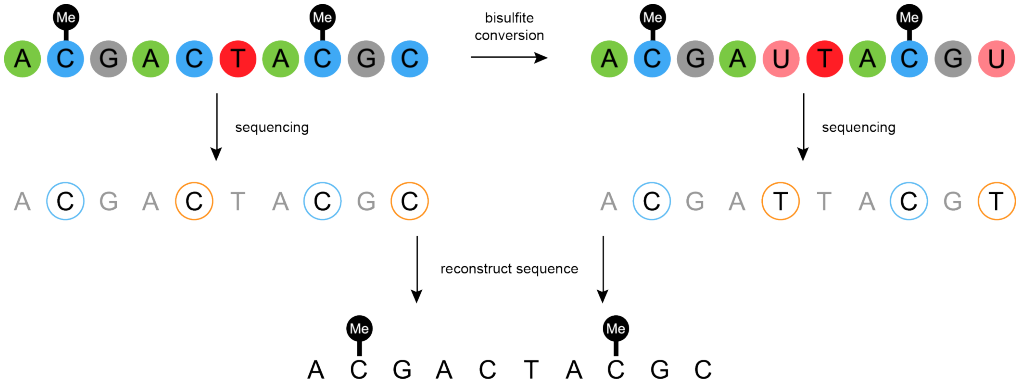
DNA methylation is a vital part of growth and development of living creatures, playing a part in embryonic development, transcription and genomic imprinting. Although methylation often occurs in a familiar pattern, occasionally aberrant DNA methylation patterns are observed[13]. It is in these cases that DNA methylation can have harmful consequences such as the development of carcinogenesis, the creation of cancer. The importance of methylation is ever intensifying with the growing number of human diseases which are known to occur due to epigenetic inconsistencies. The diverse group of associated diseases are presented comprehensively in [14], including cancer, numerous paternally and maternally transmitted disorders, syndromes characterized during infancy and even autoimmune diseases such as lupus. It is worth noting that some of these diseases are related to mutation in our genetic methylation machinery.

Over the last few decades our understanding of the proteins involved in this process, such as methylcytosine-binding proteins, has grown rapidly. This group of proteins includes MECP2 (methyl CpG binding protein 2), a protein deemed crucial for the functioning of nerve cells, which can mutate to cause Rett syndrome[15], a post-natal neurological disorder. By controlling the epigenetic activation of genes, using re-expression, there is increasing interest into pharmacologically reversing abnormalities caused by methylation. This has been trialled as part of cancer therapy in [16].

2.1.2 Bisulfite sequencing

Bisulfite sequencing involves the treatment of DNA with bisulfite ions (HSO_3), in order to analyse methylation patterns. By treating the DNA with bisulfite ions, cytosine nucleotides are converted to uracil, yet methylated cytosines are unaffected. The methylation state of the original DNA can be simply inferred by counting the number of cytosines and thymines at genomic cytosine sites. Various forms of analysis can be performed on the converted DNA in order to get a nucleotide specific resolution of some DNA's methylation status.

Figure F.3: Bisulfite sequencing - sequence conversion and reconstruction[17].



Whole-genome bisulfite sequencing (pre-requisite for Methy-Pipe) involves high throughput analysis of DNA methylation for the entire genome - our genome is composed for roughly 3.2×10^9 nucleotides. Although the process is the same as that previously described, it involves use of next-generation sequencing (NGS) machines in order to produce significant volumes of reads (samples of the genome) in a massively parallel way. The sequences are obtained through a destructive process in which the DNA is cut, requiring the DNA to be re-aligned to the reference genome in order to determine methylation states based on the mismatches through conversion of unmethylated cytosine to uracil.

In particular, CpG sites are examined. These are regions of the DNA where C (cytosine) and G (guanine) nucleotides are found next to one another (in that order), forming dinucleotides where the cytosine can undergo methylation. The regions where CpG sites experience methylation are referred to as differentially methylated regions (DMR), i.e. regions where the methylation patterns will differ to a different sample (in this case a reference genome).

Some of the notable uses of whole-genome bisulfite sequencing with single base resolution include non-invasive detection and monitoring of methylation associated with cancer[4], and non-invasive sequencing of the human fetus[3] that allows pre-natal diagnosis. Both of these cases are deemed non-invasive as long DNA strands are sampled by shotgun sequencing (expanding and quasi-random pattern of sequencing) of plasma, a component of our blood that composes roughly 55% of a human body's total blood volume. In the case of pre-natal diagnosis, maternal plasma can conveniently be used, avoiding the need to pass a needle through a mother's lower abdomen and into the uterus.

2.2 Sequencing Machines

In the previous section we introduced the term next-generation sequence machine, to describe the high-throughput machines used to generate genomic data (sequences) from a given DNA sample. DNA sequencing is the process of determining the order of nucleotides within DNA. Sequencing machines produce nucleotide sub-sequences from a DNA sample, facilitating analysis.

Following the development of Sanger sequencing in 1975[18] and its further automation in 1986[19], The Human Genome Project successfully sequenced the first human genome sequence by 2003. This has led to an increasing demand for inexpensive and highly efficient sequencing methods with greater genetic granularity. The human condition is far more complex than a catalogue of human genes; we need to understand subtle phenomena such as epigenetic patterns which influence the expression of genes during development (i.e. epigenetic processes such as the methylation of cytosine). This contributed to the motivation behind creating a range of second-generation sequencing techniques, also known as next-generation sequencing (NGS).

2.2.1 Next generation sequencing

The major advancement of NGS is the ability to produce a dramatically increased volume of genetic sequencing data, with millions if not billions of reads being produced per instrument run. This allows a whole genome to be sequenced in a day, increasing sequencing data throughput and in turn facilitating research and allowing sequencing techniques to become more routine in healthcare. Compared to the previous Sanger Sequencing technique, which produces long reads of roughly 500-750 base pairs (bp) with a low error rate, NGS techniques produce significantly shorter reads. Paired-end read sequencing can be used to reduce error which could follow from using significantly shorter reads. This involves taking sets of reads from both ends of sequenced DNA, instead of taking single-end reads. Given that the distance between the two reads taken is known, the accuracy for each pair improves - it helps understand the origination of the sub-sequences.

There are numerous NGS platforms, each with a unique approach to sequencing. Factors influencing selection of a sequence platform include size of the genome being studied, the depth of coverage needed and the accuracy needed. For example, Illumina is a company renowned for its numerous sequencers[20], holding 70% of the sequencing market[21] and accounting for over 90% of all DNA data produced[22]. Illumina's sequencers all vary in their outputs, run-times, paired-end reads, maximum read lengths etc - a comparison is shown in T.I. It was claimed by Illumina in 2014 that forty of their highest throughput platform, HiSeq X, would be able to sequence more genomes in a single year than produced by all other sequencers to date[23]. The growth in raw output data generated by NGS platforms such as these have exceeded Moores Law, resulting in more data being produced than can be efficiently analysed and stored[24].

Platform	Run type	Max read len. (bp)	Run-time (hr)	Hg per run*	Cost per M reads (£)
MiSeq	Micro	300	22	0	-
	V3 600 cycle	600	55	0	39
HiSeq	2000 High-output	100/200	264	7	6.12
	2500 Rapid	250/500	60	3	7.05
	2500 High-output	125/250	144	11	5.05
	4000 High-output	150/300	120	17	4.85
	HiSeq X	150/300	72	20	2.00

* Human genome per run at 30x coverage, e.g. 90Gb

Table T.I: Comparison of Illumina MiSeq and HiSeq NGS technologies[25].

2.2.2 Sequencing data storage format

Beyond the generation of sequencing data, the way in which it is stored is important. There are numerous well-known formats for storing sequencing data, allowing it to be shared widely. Two of the most commonly used formats for sharing sequencing read data are the FASTA and FASTQ formats. These are both plain text ASCII formats, for the sake of human readability.

2.2.2.1 FASTA

FASTA form is the simplest of formats for sequencing read data, containing a line of meta-information followed by the read. It was originally developed by Bill Pearson for use with the FASTA suite of tools[26]. The read sequence is represented as ASCII strings of nucleotide acids or proteins. In the case of nucleotides, the alphabet used is {A, C, G, T, U}[1] plus symbols for inaccurate sequencing (typically N) and indels (typically a hyphen). An example is shown in F.4. Large numbers of reads can be formatted with FASTA and stored in an individual file, such that the meta-information acts as a read delimiter.

[1]Although {A, C, G, T, U} is the alphabet used by the majority of reads, additional characters do exist, such as V for not thymine and R for purine.

> SRR014849.1 EIXKN4201CFU84 length=93
GGGGGGGGGGGGGGGGCTTTTTTTGTTTGGAACCGAAAGG
GTTTTGAATTTCAAACCCTTTTCGGTTTCCAACCTTCCAA
AGCAATGCCAATA

Figure F.4: Example of FASTA format for sequencing read data[27].

2.2.2.2 FASTQ

FASTQ form extends FASTA, combining read data with associated per base quality scores. The additional information follows the read itself, using an additional meta-information header, allowing full backwards compatibility with FASTA. An example is shown in F.5. There exist at least three variants of FASTQ; in [27] the original Sanger standard and Solexa/Illumina variants are detailed, alongside conversion between them.

The quality score is the probability that a sequenced produced the incorrect base, and is often parameterised by bioinformatics tools such that reads beyond a threshold inaccuracy are discounted. The quality score, or PHRED quality score, was named after the PHRED software [28][29] which reads DNA sequencing traces, calls bases and assigns each a quality value. The file format created to hold these quality scores is known as the QUAL format, and stores the scores as plain text ASCII characters - this is what follows a FASTA record in the FASTQ form. The PHRED quality score is defined as follows:

Definition - PHRED quality score

Let P_e be the estimated probability of error for a base,

$$Q_{PHRED} = -10 \times log_{10}(P_e) \tag{2.1}$$

@SRR014849.1 EIXKN4201CFU84 length=93
GGGGGGGGGGGGGGGGCTTTTTTTGTTTGGAACCGAAAGG
GTTTTGAATTTCAAACCCTTTTCGGTTTCCAACCTTCCAA
AGCAATGCCAATA
+SRR014849.1 EIXKN4201CFU84 length=93
3+&$#”””””””””””””””7F@71,”;C?,B;?6B;:EA1EA
1EA59B:?:#9EA0D@2EA5: > 5?:%A;A8A;?9B;D@
/= < ?7=9 < 2A8==

Figure F.5: Example of FASTQ format (Sanger variant) for sequencing read data[27].

2.3 Sequence Alignment

Sequence alignment is a way of arranging multiple DNA sequences to identify similar regions, and is required to identify where sequencing reads are located in a full human genome (reference genome). It is consequently pivotal in the identification of differentially methylated regions. Following introductions to the FASTA/FASTQ sequencing data formats, it should be clear that DNA sequences being aligned are expressed as sequences of nucleobases with the alphabet $\Sigma = \{A, C, G, T\}$.

2.3.1 Global and local alignment

When aligning two sequences, you can either do it globally or locally, see F.6. The local alignment is used to find the common regions of the two sequences, and is much more flexible than a global method in which the optimal alignment involves finding the maximum total number of matches (all characters participate in the alignment). Therefore with local alignment, similar regions in different orders can be identified, which can be useful in situations where the sequences are unrelated or differing in length.

```
Global: GTGTACNCCANAN
        G--TAC-CCA-AN
Local: GTGTACNCC-ANAN
       --GTAC-CCAAN--
```

Figure F.6: Global and local sequence alignment for `GTGTACNCCANAN` and `GTACCCAAN` where '-' is an indel.

In the context of whole-genome sequence alignment, local alignment is almost solely used. This is because high throughput sequencing machines produce a tremendous number of reads (discussed in the previous section), which are significantly shorter in length than that of the reference genome - reads range from 75 to a few hundred bases, whereas the human genome is roughly 3.2×10^9 bases. This naturally leads to a high number of reads which overlap with others, and an incredibly high number of candidate alignment locations. An explanation for this is the minimal alphabet of DNA and the possibility for bases to change through mutation or sequencing errors. This poses a great computational challenge similar to that of generic substring searching, so the alignment algorithm used must be chosen carefully.

2.3.2 Alignment algorithms

Algorithms exist to find the optimal alignment between two given sequences. The optimal alignment is determined by a score, in which recursively replacing, inserting or removing an element of a sequence carries an individual score contributing to the total score over the alignment. The algorithms used for alignment can largely be placed into one of two groups: (1) suffix-trie algorithms, such as the FM-index[30], where reads are aligned using an index of the reference genome being used; or (2) dynamic programming algorithms, such as the Smith-Waterman algorithm[31], where optimal alignments are found using a scoring matrix. This section will detail both the FM-index and Smith-Waterman algorithm. When short reads are used with less than 150 bases, and small edit distances, suffix-trie algorithms generally achieve better performance. Whereas when using large reads with hundreds or even thousands of bases, and potentially large edit distances, dynamic algorithms generally achieve better performance.

2.3.2.1 FM-index

FM-index is a compressed full-text substring index based on the Burrows-Wheeler Transformation (BWT)[32], and to some extent the suffix array[33]. It allows substring searching with a linear dependence on alphabet size in time and exponential dependence in space.

The suffix array of a string S is the lexicographically sorted array of its suffixes when a terminal symbol (\$) is appended to it, and each suffix is described by its offset in S. When it comes to identifying substrings of S, we are interested in suffixes with appropriate prefixes. The suffix array interval $(low, high)$ covers the range of indices whereby suffixes share the same prefix, allowing the result of a search operating to be this interval. An interval where low is greater than

$high$ indicates there are no substring hits in S, whereas if low is less than or equal to $high$ there has been at least one hit. An example can be found in F.7 for the string 'BANANA'.

(a) Suffix array and BWT

i	SA	Suffix
0	6	$
1	5	A$
2	3	ANA$
3	1	ANANA$
4	0	BANANA$
5	4	NA$
6	2	NANA$

BWT = ANNB$AA

(b) $Occ(s, i)$ and $C(s)$

i	A	B	N
0	1	0	0
1	1	0	1
2	1	0	2
3	1	1	2
4	1	1	2
5	2	1	2
6	3	1	2

(c) $C(s)$

A	B	N
1	4	5

(d) FM-index (d=4)

BL_0

A	B	N
1	0	0

$BWT_0 = ANNB$

BL_1

A	B	N
1	1	2

$BWT_1 = \$AA-$

Figure F.7: Suffix array and Bruce-Wheeler Transformation of 'BANANA'.

The BWT generates permutations of symbols in a text, where each position of the transformation is computed using a string S and its suffix array SA_S. It can be understood as the final column of a matrix where rows are cyclic shifts of S (with terminal $) in lexicographic order. Through use of two auxiliary functions acting upon a BWT, the FM-index allows substring searching. These two functions are $Occ(s, i)$ and $C(s)$. $C(s)$ returns the number of symbols in the BWT that are lexicographically smaller than the symbol s, and $Occ(s, i)$ returns the number of times s occurs in the BWT from positions in the range [0, i]. The values of these two functions are pre-computed and stored as arrays, forming the FM-index.

To compress the size of the FM-index, the $Occ(s, i)$ array can be sampled to create buckets of size d. This involves storing $Occ(s, i)$ values every d elements as markers, which are interleaved with the corresponding substrings of the BWT. This reduces the $Occ(s, i)$ array size by a factor of d, but means that some occurrences must be calculated directly from the BWT. An example of the BWT and the $Occ(s, i)$ and $C(s)$ functions can be found in F.7.

The FM-index search operation is described in Alg.1. The suffix array intervals are initialised to the boundary indices of the suffix array, and then updated for each symbol in the substring being searched for. This is done backwards. Having iterated through the substring, the number of reference hits are equal to $high - low$. Each exact reference position is found by converting the index in the suffix array of the reference string. An example FM-index search operation can be found in F.8.

Figure F.8: FM-index search for substring 'ANA' in 'BANANA'.

Iteration, i	Symbol	$(low, high)_{(i-1)}$	$(low, high)_i$
0	-	-	(0, 6)
1	A	(0,6)	(1, 3)
2	N	(1,3)	(5, 6)
3	A	(5,6)	(2, 3)

The interval (2,3) gives text positions 1 and 3.

Algorithm 1 FM-index search algorithm

Input: FM-index F, bucket size d, string S, substring R and suffix array SA_S
Output: $positions$ where R occurs in S

```
low ← 1
high ← |S|
for i ← |R| − 1 to 0 do
    low ← C(R[i]) + Occ(R[i], F[low − 1/d], low − 1)
    high ← C(R[i]) + Occ(R[i], F[high/d], high) − 1
end for
for i ← low to high do
    positions ← SA_S[i]
end for

function C(s, i)
    marker ← F.markers[s]
    count ← 0
    for j ← 0 to j < i%d do
        if s = F.bwt[j] then
            count ← count + 1
        end if
    end for
    return marker + count
end function
```

2.3.2.2 Smith-Waterman algorithm

The Smith-Waterman algorithm is well known dynamic algorithm developed in 1981 to find the optimal local alignment of two sequences. It involves constructing a similarity score matrix, in which the character of each sequence is compared with all others, and scored accordingly. For the alignment of two sequences, s_1 and s_2, gap-scoring scheme W_i and similarity function $s(a, b)$, it is defined by the following recurrence relation:

Recurrance Relation - Smith Waterman

$$M(i,0) = 0, 0 \leq i \leq length_{s_1} \tag{2.2}$$

$$M(0,j) = 0, 0 \leq j \leq length_{s_2}$$

$$M(i,j) = max \begin{cases} 0 \\ max_{p \geq 1} \left(M(i-p, j) + W_p \right) \\ max_{p \geq 1} \left(M(i, j-p) + W_p \right) \\ M(i-1, j-1) + s(s_{1i}, s_{2j}) \end{cases}$$

where $1 \leq i \leq length_{s_1}, 1 \leq j \leq length_{s_2}$

Out of the three cases of the recurrence relation, a diagonal step corresponds to a replacement (character match or mismatch), a leftward step refers to a deletion (introduction of a gap in s_1) and an upward step refers to an insertion (introduction of a gap in s_2).

Following construction of the scoring matrix M, the optimal local alignment is found by backtracking from the cell in which the maximum score was achieved. This involves moving towards the highest score neighbor to the above, left or diagonal of a cell, until the head of one of the sequences is reached. On each backtracking step, the corresponding characters for a cell $M(i, j)$ can be prepended to the aligned sequences. This results in a space and time complexity of $\mathcal{O}(mn)$, where m and n are the sequence lengths.

$$H = \begin{pmatrix} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & 2 & 1 & 2 & 1 & 2 & 1 & 0 & 2 \\ G & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ C & 0 & 0 & 3 & 2 & 3 & 2 & 3 & 2 & 1 \\ A & 0 & 2 & 2 & 5 & 4 & 5 & 4 & 3 & 4 \\ C & 0 & 1 & 4 & 4 & 7 & 6 & 7 & 6 & 5 \\ A & 0 & 3 & 3 & 6 & 6 & 9 & 8 & 7 & 8 \\ C & 0 & 2 & 4 & 5 & 8 & 8 & 11 & 10 & 9 \\ A & 0 & 4 & 3 & 6 & 7 & 10 & 10 & 10 & 12 \end{pmatrix} \quad T = \begin{pmatrix} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ G & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \nwarrow & \uparrow \\ C & 0 & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \nwarrow & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ C & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \uparrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ C & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \uparrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \nwarrow \end{pmatrix}$$

Figure F.9: Alignment of sequences $ACACACTA$ and $AGCACACA$ using Smith-Waterman. The results being $A-CACACTA$ and $AGCACAC-A$[34].

2.4 Methy-Pipe

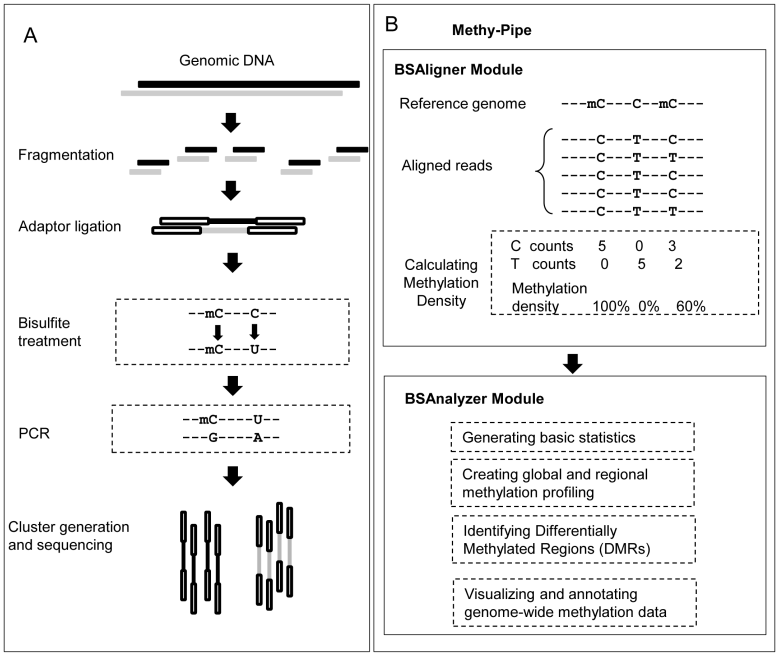
Methy-Pipe facilitates exploration of DNA methylation across the entire genome at single base resolution, using whole-genome bisulfite sequencing. It analyses bisulfite sequencing data produced from MethylC-Seq[35], a library preparation protocol providing genome-wide coverage for CpG sites. MethylC-Seq has been commended for its simplicity, and has been used commercially for numerous studies of whole-genome DNA methylation. However, an integrative computational tool was required to satisfy the numerous requirements of different research focuses centered around methylation data analysis; topics of interest may include methylation-aware alignment, identification of DMRs etc. Many software packages exist that facilitate bisulfite sequencing read alignment[36][37], and for specific-purpose downstream analysis[38][39], but Methy-Pipe was implemented to integrate these functions in a user friendly manner.

2.4.1 Implementation

Methy-Pipe was written using a combination of Perl, R and C++. It has been designed to run on the x86_64 GNU/Linux platform, whereby data analysis performance is improved by distributing multiple samples to nodes across a cluster running a Sun Grid Engine. The workflow of Methy-Pipe is shown in F.10B. There are two modules of Methy-Pipe that are run sequentially: BSAligner and BSAnalyzer. BSAligner is the module responsible for bisulfite sequencing read alignment. It was implemented with influences such as 2BWT[40] and SOAP2[41], both flexible read alignment tools using the BWT and the suffix array. BSAnalyzer is the module responsible for downstream methylation data analysis. It has numerous functions: (1) reporting basic statistics and data sequencing quality; (2) calculating methylation levels of C sites on top of genome-wide methylation levels; (3) identification of DMRs and (4) to annotate and visualise methylation information for further data analysis, mining and interpretation. In this section we will go on to look at (2) and (3).

The input data for BSAligner is high-throughput bisulfite sequencing reads produced from a single or paired-end library and prepared according to the MethylC-Seq protocol F.10A. The reads are presented in the 'FASTQ' format (.fa), whereby a text file stores records of reads and their corresponding quality scores.

Figure F.10: Methy-Pipe workflow[2].



2.4.2 Bisulfite sequencing read alignment

Following the production of bisulfite sequencing reads, BSAligner must prepare them to be aligned back to the reference genome, see F.12. This involves trimming the reads, by disregarding the adapters (product of sequencing process) and bases at the end of reads with quality scores lower than a threshold.

Alignment back to the reference genome is challenging compared to non-bisulfite sequence alignment, as bisulfite conversion dramatically transforms the DNA F.11. Specifically, the DNA changes from a large double-stranded molecule to being single-stranded, as cytosines will have almost completely changed to uracil, in turn removing complementarity. Due to this transformation, Methy-Pipe depletes the cytosines of the reference genome in silico, i.e. converts all cytosines to thymines computationally. This requires creating a separate set of the original bisulfite sequencing reads that are fully cytosine depleted, which is again produced in silico. Indices can then be built for the reference using the BWT algorithm, for alignment.

For alignment, BSAligner first loads the indices into computer memory, before two alignment stages: (1) both the pre-processed and converted reads are then aligned to initial (non-converted) reference genomes, with any reads aligning back to both W and C strands being discarded; and (2) the remaining converted reads are replaced by the original bisulfite sequencing reads, and passed downstream for methylation calling. This involves checking the methylation state of each cytosine in the original bisulfite sequencing reads.

Template (double stranded)

A: 5' -GACCGTTCCAGGTCCAGCAGTGCGCTT 3'
B: 3' -CTGGCAAGGTCCAGGTCGTCACGCGAA- 5'

Bisulfite converted (single stranded)

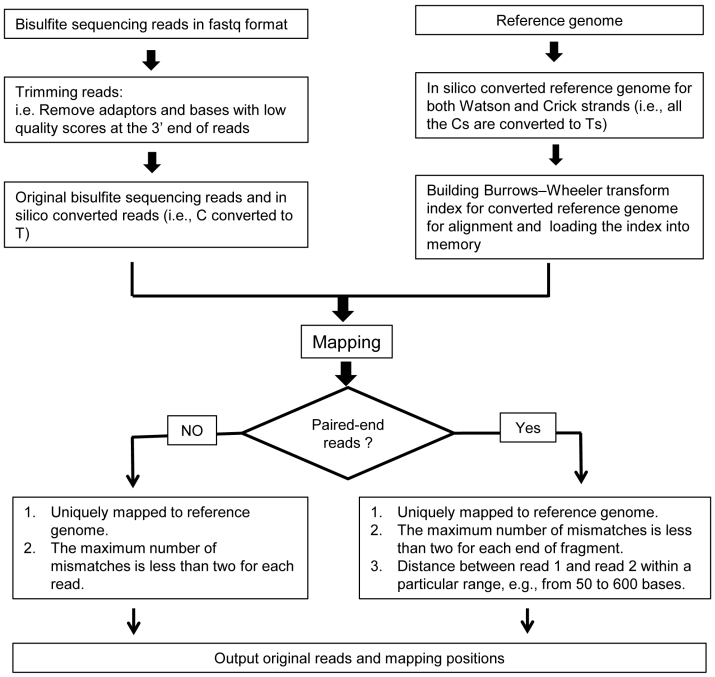
A: 5' -GATCGTTTTAGGTTTAGTAGTGCGTT- 3'
B: 3' -TTGGCAAGGTTTAGGTTGTTATGCGA- 5'

Figure F.11: Bisulfite converted DNA strands[42] - following bisulfite treatment, there is a loss of reverse complementarity between the strands.

Methy-Pipe has two constraints on alignment: (1) a unique mapping must exist for each in silico converted, i.e. reads aligned back to both strands are disregarded and (2) no more than two mismatches can occur in each read. Any read not meeting these criteria is discarded, such that the original bisulfite sequencing reads corresponding to each aligned in silico aligned read can be passed on to downstream methylation data analysis (BSAnalyzer).

The alignments are outputted in a text file containing the chromosome, position, mismatches (up to two), sequencing qualities etc. The exact file format with all headers and example content be found in Appendix B(IXa).

Figure F.12: Methy-Pipe BSAligner - bisulfite sequencing read alignment module[2].



2.4.3 Calculation of the methylation density level

Methylation state of a DNA sample is inferred by examining the number of cytosines and thymines at CpGs; after bisulfite treatment, the methylcytosines are not modified unlike other cytosines which are converted into uracils before PCR further converts them into thymines.

For BSAnalyzer to calculate the methylation density level, the total number of cytosines and thymines overlapping with each CpG site across the whole genome are calculated, and then the following equation is applied:

Definition - Methylation Density

Let n be the resolution of the CpG site in base pairs (bp), $|C_i|$ be the total number cytosines sequenced at the ith position in the reference genome and T_i be the same metric for thymine:

$$\text{MD} = \left(\sum_{1}^{n} C_i \div \sum_{1}^{n} (C_i + T_i) \right) \times 100\% \tag{2.3}$$

In a given CpG site, C_i is the total number of cytosines sequenced at the ith position in the reference genome, and indicates methylation. T_i follows the same logic but instead suggests an unmethylated region. The closer n is to 1, the closer to single-base resolution the methylation density can be calculated.

2.4.4 Identifying differentially methylated regions

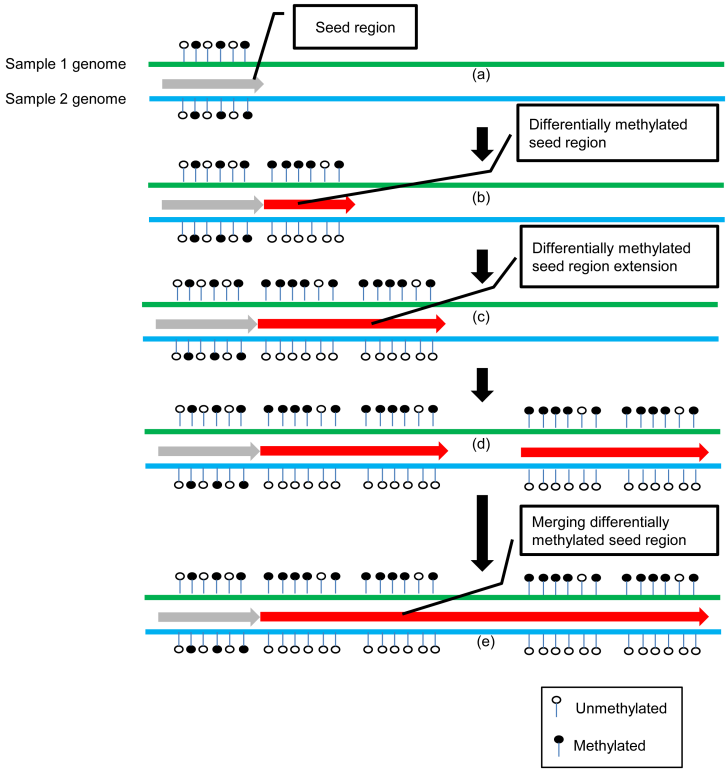
BSAnalyzer allows identification of DMRs genome-wide by comparing two samples, using a sliding window approach shown in F.13. It has four key stages: (1) determining seed regions, (2) identifying whether the seed is a DMR, (3) DMR extension and (4) merging adjacent seed regions if they are differentially methylated.

To determine the seed locations, a sliding window of length w (where w is specified in terms of base pairs, bp) is applied from one end of the chromosomes of the two samples. Between slides, the window will be defined as a seed region if it contains at least m valid CpG sites and each valid CpG site is covered by at least n bisulfite sequencing reads. If this criteria isn't met, the downstream slide will be by an s-base increment. For example, m and n may be set to 5, s to 100 and w to 500bp.

To test if a seed is a DMR, the methylation density of each valid CpG site is calculated and Mann-Whitney U tests are used to test for statistical significance between the two samples. If this test holds for a given p-value, say 0.01, then the seed region is deemed a DMR. The region can then be extended, using this approach again on successive extensions and merging if the neighboring region is also a DMR. This extension will terminate if a new region does not classify as a DMR, or if the overall length of the extended seed exceeds a specific length k (e.g. k=1000). Extended DMR regions are merged if they are within 1000 bases of one another and share a similar methylation pattern.

Finally, chi-squared tests are applied to all CpG sites within merged DMRs, to examine whether the proportion of methylated cytosines to total sequenced is statistically different between the samples. If a merged DMR is not significantly different, then it is not deemed a putative DMR.

Figure F.13: Methy-Pipe BSAnalyser - differentially methylated region detection module[2].



2.4.5 Performance and bottlenecks

Methy-Pipe has a typical workload of 300M short bisulfite sequencing reads (75bp). The total pipeline (including BSAnalyzer's downstream analysis) takes roughly 30 minutes for only 10M 75bp reads with a peak memory usage of 25GB when run on a single 20-core Intel Xeon X5675 processor. At a $\frac{1}{30}$th of the typical workload, it is clear that the overall runtime could exceed 10 hours on a meager hardware set-up. This means without acceleration or a significant hardware backbone, Methy-Pipe is not easy to use routinely within a clinical environment.

With a full work-load of 300M short reads of 75bp, BSAligner alone takes roughly 5 hours when running on a system with dual 12-core Intel Xeon processors. This is not fast enough to meet the throughput of NGS machines, for example in 5 hours the Illumina HiSeq 4000 High-output can produce over 200M reads which are four times longer at 300bp. BSAligner is certainly the bottleneck in Methy-Pipe, as its workload cannot currently be distributed across a cluster unlike BSAnalyzer. Alignment of reads can be highly parallelised however, as the alignment of each individual read does not depend on the alignment of others.

There is a specific module creating a memory bottleneck prior to use of BSAnalyzer, responsible for the second stage of bisulfite sequencing alignment. The module performs methylation calling, whereby methylation information is recorded at single-base resolution. This requires creating methylation record for each base position in the reference genome, using roughly 24GB (the peak usage across Methy-Pipe).

2.5 Compression of Genetic Sequencing Data

Improvements in DNA sequencing technology has not only posed the challenge of reconstructing genetic information using aligners such as Methy-Pipe's BSAligner, it has also meant storage is a central bottleneck in many systems. This can prevent the acquisition of sequencing data that would otherwise prove invaluable in genomics research and healthcare. Given that a human reference

genome alone requires roughly 3GB of storage, if whole-genome sequencing pipelines such as Methy-Pipe are introduced into regular healthcare, the volume of patient sequencing information recorded may be unmanageable. General purpose compression algorithms such as `gzip`[43] can be used to achieve moderate compression, however they fail to harness inherent properties of DNA.

2.5.1 Referential compression algorithms

It is common knowledge that any two human genomes are over 99% identical[44], which suggests compression algorithms can leverage the redundancy to achieve significant compression. With the use of a common reference genome, such as the consensus human genomes[45], an encoder and decoded can compress a genome losslessly. This approach is called referential or relative compression. The popularity of this referential compression schemes is increases steadily, as many complete genomes are becoming available for shared use as a reference.

Lossless referential genome compression is a two stage problem: (1) a mapping from a given reference to a target genome must be created and (2) this mapping must be described as concisely as possible to the decoder. Below is an example of referential compression, where triples are used to describe this mapping.

Referential Compression - Example[46]

Let each mapping instruction be of the form $\langle p_i, l_i, z_i \rangle$ where p_i refers to an absolute position, l_i a length of similar symbols and z_i the edit symbol.

Target : $\underline{AATGC}\,\underline{AGGTAC}\,\underline{TAT}\,\underline{AAG}N\,\underline{A...}$

Reference :$\underline{AATGT}\,\underline{AGGTAC}\,\underline{AT}\,\underline{AA}\underline{G}\,\underline{ATG...}$

Producing instruction set $\{F\}$:

$$F_1 : \langle p_1, l_1, z_1 \rangle = \langle 1, 4, C \rangle$$

$$F_2 : \langle p_2, l_2, z_2 \rangle = \langle 6, 6, T \rangle$$

$$F_3 : \langle p_3, l_3, z_3 \rangle = \langle 12, 5, N \rangle$$

...

In [47] an approach is taken whereby two files are produced for this mapping, one consisting of single nucleotide polymorphisms (SNPs) and one for indels of multiple nucleotides. Given that many databases store common SNPs, such as dpSNP[48], algorithms such as the one presented in [49] can compress these two files losslessly.

The approach taken in [46] to referential compression does not require external libraries or databases, and is motivated by the sliding window Lempel-Ziv algorithm[50], a universal lossless compression algorithm. They manage to compress a sample of 2991MB genome data down to 6.99MB, while `gzip` compresses it to only 834.8MB. This corresponds to a greater than 99% reduction in size, whereas `gzip` reduces the size by 72%, demonstrating the utility of specialised compression algorithms in tackling the storage bottleneck created by improving DNA sequencing technology.

2.5.2 Genetic compression algorithms

In [51] a comprehensive study of genetic sequencing compression is presented, including non-referential techniques. We will briefly present two notable referential compression algorithms, FRESCO[52] and GDC[8], before summarising the results presented in this study for these compression algorithms.

Definition - Data Compression Ratio

$$Compression\ Ratio = \frac{Uncompressed\ Size}{Compressed\ Size} = \frac{Uncompressed\ Data\ Rate}{Compressed\ Data\ Rate} \text{ (for streamed applications)} \tag{2.4}$$

2.5.2.1 FRESCO

Framework For Referential Sequence Compression (FRESCO) compresses any raw genetic sequencing data with the alphabet {A, C, G, T, N} (i.e. no meta-information). FRESCO first choses a reference sequence out of a large collection, which is used to create triples similar to those previous demonstrated; matches between the text and reference are recorded using the offset in the reference, the length of the match and the first mismatching character which occurs. The matches are determined using a hash index over the potential subsequences that can be created using the genetic alphabet, for a given length of symbols. The matches are then chosen using one of three strategies, a greedy search algorithm that uses the longest matches or two optimised strategies for the look-up of short local matches (Lempel-Ziv factorisation [53]).

Similar to many genetic compression algorithms, FRESCO offers a second order of compression harnessing genetic similarity. Given a collection of sequences, due to the similarity between the raw sequences, the mappings to a common reference tend to be similar. FRESCO's second order of referential compression (post-processing stage) involves locating common reference triples with respect to a designated reference compressed sequence, allowing common triples in the other compressed sequences to be replaced by pointers. In the final stage these triples are either output as plain text, or further encoding can take place (for example, using a binary compression technique based on Huffman coding).

The selection of a reference from a large selection of sequences is described as NP-hard by the authors, with a naive approach to finding the best reference having quadratic complexity ($\mathcal{O}(n^2)$) for the number of candidate sequences. FRESCO uses heuristics to solve this problem, and can generate an artificial reference if necessary out of a collection of sequences.

2.5.2.2 GDC 2

Genome Differential Compressor (GDC) 2 compresses complete sequence collections in FASTA format variants, and conceptually extends the FRESCO's approach to raw sequence compression. Both the algorithms use a two-stage compression strategy, involving the Lempel-Ziv algorithm, with GDC 2 introducing a few new concepts to improve compression throughput.

The first stage of compression involves identifying exact matches in parallel (one sequence per thread) using Lempel-Ziv factorisation, and identifying any short matches which represent SNPs or indels after these exact matches. Unlike the exact matches, these short matches do not require consulting a hash table, and instead involve direct reference and text comparison. This improves compression speed, and the compression of these matches can be performed such that the overall compression ratio isn't harmed. The second stage involves another Lempel-Ziv factorisation over compressed sequences, whereby runs of common reference triples are encoded with respect to a reference. Finally, the encoded values and reference sequence are processed by an arithmetic coder and specific methods such as delta encoding (depending on the type of sequence data) are used on the encoded values.

2.5.2.3 Compression performance

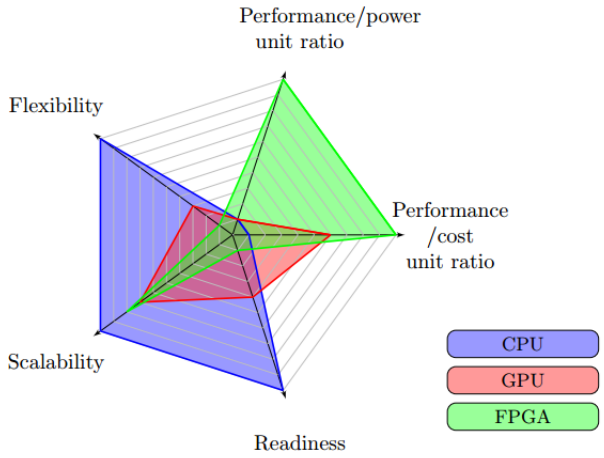
The results in [51] were taken from compression of 5 sequences of Arabidopsis thaliana DNA, with reads of roughly 94bp, that together use 243MB of storage on disk. In total, 16 compression algorithms were tested, where FRESCO and GDC 2 were the only solely reference-oriented algorithms. FRESCO and GDC 2 both achieved poor compression throughput at 0.02MB/s and 0.01MB/s respectively, which was much lower than the competition. However, FRESCO managed 21.05MB/s of decompression throughput.

Among the three FASTA compressors tested, GDC 2 achieved a significantly lower compression ratio than the two other approaches at 2.68. FRESCO was unique among the compression algorithms in that it only compresses raw sequence information, but achieved a compression ratio of 7.72, which was the third greatest overall below the two FASTA compressors. These two FASTA compressors, kpath[54] and ORCOM[55], take a statistical reference-based approach and reference-free approach to compression respectively.

2.6 Hardware Acceleration

In this section we present the various hardware platforms that can be used to accelerate problems such as sequence alignment and genome compression. For each platform, processing elements can be defined as the smallest part of the hardware architecture that can perform an algorithm (these are also known as processing cores).

Figure F.14: Comparison of hardware platform properties[11].



2.6.1 CPU

Central Processing Units (CPUs) are the most common processing units, and can be found in both personal computers and higher performance computers. They can be used standalone, or alongside others (multiprocessing). Multiple CPUs can also be placed on a single silicon chip (multi-core processors). CPUs are general purpose, providing large instruction sets that improve their scalability and flexibility (see F.14). Unfortunately the nature of general purpose processing elements means much of a CPU is inactive at any one time. The performance of CPUs is largely dependent on the frequency by which the processing elements operate, i.e. the clock rates and how many instructions can be processed in a single clock cycle. The sequential model of CPUs means that clock frequencies are often higher than the processing elements of other hardware platforms, improving suitability for tasks that do not require parallelism.

Although multiprocessing can be used in highly scalable way, creating behemoth clusters of processing elements which workloads can be distributed over, these solutions are not suitable in a clinical setting. For example, the 1000 genome project[56] uses a 1192-processor cluster to align reads, while the BGI Bio-cloud[57] biological storage platform has a current total of 14,774 processors delivering 157Tflops of performance.

2.6.2 GPU

Graphics Processing Units (GPUs) are specialised in and primarily used for managing computer graphics. GPUs are composed of a many processing elements (thousands unlike CPUs which have hundreds), allowing large blocks of data to be processed at any one time. Due to this highly parallel structure, there is an increasing amount of attention being paid by GPU manufacturers to more general purpose parallel computations, performed by General-Purpose Graphical Processing Units (GPGPUs). The most commonly used GPGPU programming language for heterogeneous platforms is OpenCL[58], and the most commonly used framework is CUDA[59]. Both use C++ primarily to create kernels that can be used to delegate work to device processing elements (GPUs) by the host (CPUs).

2.6.2.1 CUDA

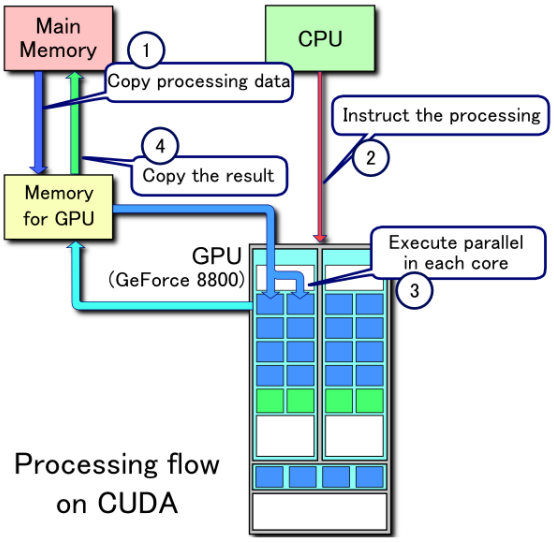
The Compute Unified Device Architecture (CUDA) is a parallel computing platform developed by NVIDIA and performs on the GPUs that they manufacture. It is built upon three key abstractions: (1) a hierarchy of thread groups (see F.15b), (2) shared memories and (3) barrier synchronization. These allow programmers to partition problems such that the sub-problems can be solved independently in parallel by groups of threads (called blocks), and each sub-problem into finer problems that can be solved cooperatively in parallel by threads within each block.

More specifically, NVIDIA GPUs are composed of multi-threaded streaming multiprocessors (SMs), which contain a significant number of cores. For example, the NVIDIA GTX Titan Black which runs on the Kepler architecture has a total of 15 multiprocessors and 2880 cores. When a GPU kernel is launched, group of threads (called blocks) are allocated to each SM, executing concurrently. Instruction level parallelism is then achieved within each thread, through instruction pipelining.

Groups of threads (blocks) can be grouped into one, two, or three-dimensional grids. Each thread can be uniquely indexed within its block, and each block can be uniquely indexed within its grid, shown in F.15b. Depending on the chosen configuration, threads can be scheduled concurrently or sequentially so that a CUDA program can execute on a desired number of multiprocessors (specified by the runtime system, see F.15a). There is a hard upper limit on thread block sizes, for example the Kepler architecture can run 1024 threads per block (32 warps[2]). Thread blocks are in turn organised in warp-sized units, and are uniformly spread throughout a grid. The Kepler architecture for example allows 2,048 threads active at any one time on each multiprocessor, allowing the following layouts: 2 thread blocks of 32 warps, 3 thread blocks of 21 warps, 4 thread blocks of 16 warps, up until 16 blocks of 4 warps.

[2]Warps are the minimum size of the data processed in a single instruction multiple data fashion by a CUDA multiprocessor.

(a) CUDA processing flow[60].



(b) CUDA parallel architecture[61].

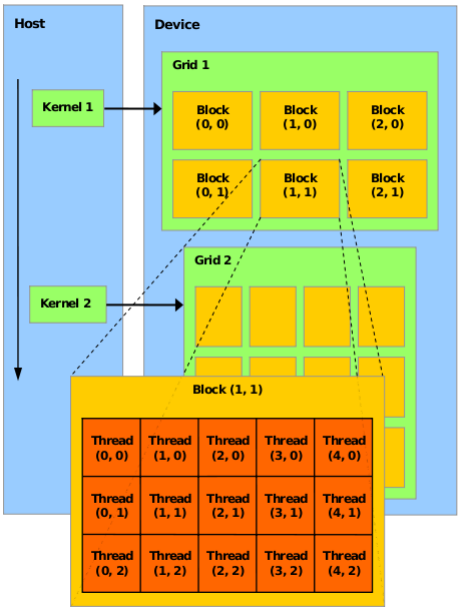


Figure F.15: CUDA parallel processing architecture[60][61]

CUDA has a multi-tiered memory hierarchy, allowing the visibility of data to vary among threads in different blocks and within blocks. Global device memory (DRAM) is large and visible to every thread alongside the host CPU and other GPUs attached to the host, yet comes at a cost of 400-800x the latency of lower-tier memory accesses. To achieve good global memory performance, access patterns maximising coalesced accesses should be used alongside storing data that meets the memory alignment. Newer GPU devices, with compute capability 3 also provide a read-only L1 cache for each SM of 64KB, and a shared L2 cache of up to 1.5MB depending on the exact GPU architecture. Local memory (thread-local) also exists in DRAM with a high latency, and is used to manage register spilling for large variables. On devices with a compute capability of 2 or greater, all local accesses are cached in both L1 and L2. Shared on-chip memory can be used to achieve a lower latency than global memory. This memory is block-local, and shares a portion of the 64KB allocated to the L1 cache (i.e. possible allocations are 48:16, 16:48 and 32:32MB). The highest shared memory bandwidth is achieved when it is divided into equally-sized memory banks allowing threads in each warp to access different data banks; otherwise accesses must be serialised. Specialised read-only constant memory and texture memory is also found in DRAM and cached for efficient access by all threads.

Significant speedup has been achieved using GPUs instead of CPUs for sequence alignment algorithms. This is incredibly promising as GPUs, although more expensive than general purpose CPUs, are still accessible and feasible for use within almost any clinical setting. For example, SOAP3-dp[5], a well known GPU-accelerated sequence aligner, aligns 6M simulate 100bp paired-end reads in 132 seconds using an Intel i7-3930k 3.2 Ghz quad-core processor and single Nvidia GTX 680 GPU. This was at least 3.1 times faster than tested software competitors.

2.6.3 FPGA

Field Programmable Gate Arrays (FPGAs) are integrated hardware circuits that can be programmed to execute combinational and sequential logic. They are composed of a matrix of configurable logic blocks (CLBs) and interconnects; the arrays of logic blocks are connected by reconfigurable wires that transmit signals adhering to the user-defined circuit. Developers define circuits that correspond functionally to applications or procedures through a hardware descriptive language

(HDL), which is then mapped to the hardware. An example of the FPGA fabric is shown in F.16, where digital signal processing (DSP) blocks and RAM blocks are interleaved between the logic blocks. FPGAs allow runtime reconfiguration, whereby the device can switch between hardware configurations, allowing processing elements to change their functionality on the go. It is for this reason FPGAs are used heavily within networking and telecommunications for routing.

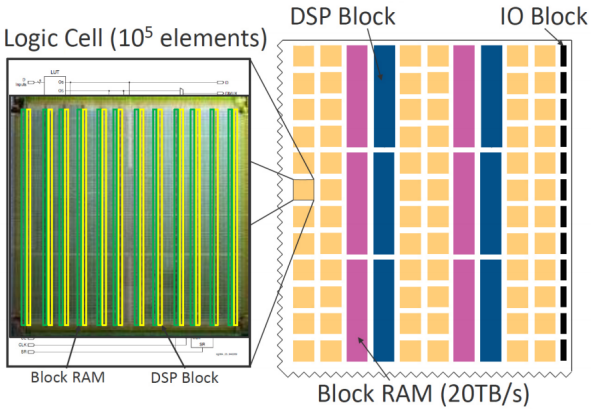


Figure F.16: FPGA Fabric - Maxeler dataflow chip[62].

FPGAs not only provide the potential for substantial speed-up due to the highly parallel custom computations that can be harnessed, but energy and power usage is often lower than that of a CPU due to the lower clock frequency by which they operate. They are also favourable in some ways to application-specific integrated circuits (ASICs) as they remove upfront non-recurring engineering costs; allow a faster time-to-market due to fewer manufacturing steps and a simple design cycle; and support post-delivery and post-fabrication changes. Unfortunately, the performance of FPGA-based sequence aligners can come at the cost of less functionality and accuracy compared to CPU solutions, due to the difficulties of developing hardware accelerated software.

2.6.3.1 Maxeler

Maxeler Technologies provides a leading platform offering a complete and comprehensive software and hardware acceleration solution, founded upon multi-scale dataflow computing. It provides a combination of traditional synchronous dataflow, array and vector processors. Using the latest and largest FPGAs available, Maxeler's hardware platforms exploit loop level parallelism in a pipelined, spatial way where high-throughput can be achieved with low-latency. Multiscale dataflow computing involves employing dataflow on multiple levels of abstraction: system level through connection of multiple dataflow engines to construct a supercomputer; architecture level by decoupling memory access from arithmetic operations; and arithmetic level and bit level through opportunities to optimise data representation to balance computation with communication.

The MaxCompiler[63] is used to write kernels that accelerate a body of host code using FPGAs (i.e. implement computational components of application in hardware), and managers that connect kernels to the CPU, RAM and other kernels or dataflow engines via MaxRing (a high-bandwidth interconnect). This kernel code is defined in a Java-like language (MaxJ), as shown in F.17b, that describes a dataflow program. In a dataflow program, execution can be modelled in a dataflow graph where data flows along the edges as tokens and is operated on by computational nodes. Instead of processing elements, Maxeler refers to each processing unit as a dataflow engine (DFE), as data streams from memory into the processing chip where data can be forwarded directly between dataflow programs. Concurrent execution naturally occurs, as many data tokens can pass through a dataflow program at any time.

(a) Maxeler architecture

(b) Maxeler dataflow example

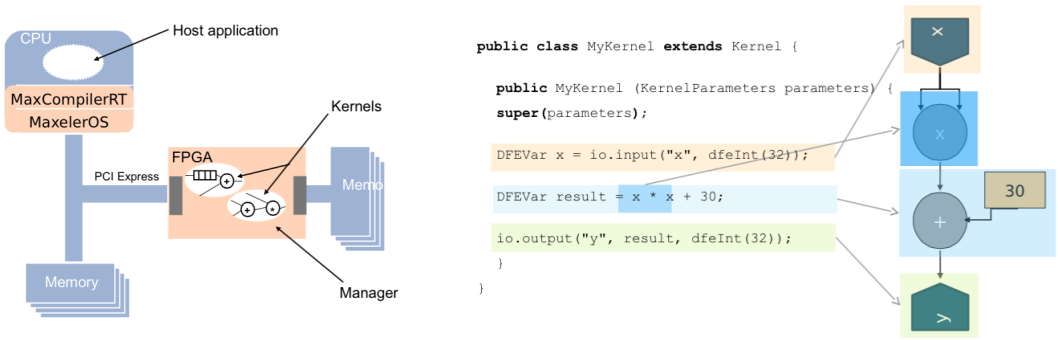


Figure F.17: Maxeler multiscale dataflow computing architecture[62].

At the host, the MaxOS and MaxCompiler compile and link the manager and kernel definitions (`.max` files), such that they can provide interfaces that can expose the accelerated regions of the application to the host code (`.c` or `.f`, C or Fortran). This interface is referred to as the Simple Live CPU Interface (SLiC). The host can then run the complete application, communicating where specified with FPGA kernels via an interconnect (LMEM, PCIe, Infiniband, or MaxRing), shown in F.17a.

This results in a simple workflow as demonstrated in F.18, where individual modules of a large application can be accelerated by carefully identifying bottlenecks, integrating a kernel, simulating the functionally and then building the final result in hardware to achieve acceleration. It is worth noting that the hardware building is entirely abstracted by the MaxCompiler, and the user is simply faced with a resource usage report explaining which lines of code use what resources in the kernel and manager. The compilation stages are roughly as follows:

1. MaxCompiler generates VHSIC Hardware Description Language (VHDL) ready for FPGA vendor tools.
2. *Synthesis* transforms this VHDL into a logical netlist - description of an electronic design's connectivity.
3. *Map* fits the basic logic into N-input lookup tables (LUTs).
4. *Place* puts these LUTs, DSPs, RAMs and other components at specific locations on the silicon chip.
5. *Route* sets up the wiring between the FPGA blocks.

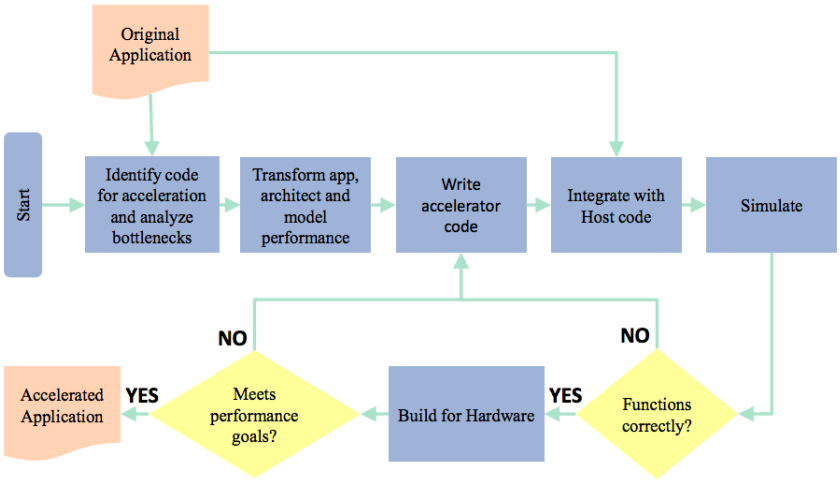


Figure F.18: Maxeler workflow for application acceleration[62].

2.7 Summary

In this chapter we have presented a broad spectrum of background content. We started by exploring the biological motivation behind this project, in facilitating whole-genome bisulfite sequencing analysis that can be used within various medical contexts such as pre-natal diagnosis and the analysis of methylcytosine-binding proteins responsible for disorders such as Rett syndrome. Bisulfite sequencing analysis was presented in the context of Methy-Pipe, a bioinformatics tool making bisulfite sequencing analysis simple and accessible. It does so through provision of a bioinformatics pipeline incorporating both the alignment of bisulfute sequencing reads and the subsequent downstream analysis expected by researchers and doctors. It is evident that tools such as Methy-pipe could prove invaluable in improving healthcare procedures, improving the livelihood of patients and potentially saving lives. It becomes clear that high-perform sequence alignment meeting the throughput of next-generation sequencing machines is not trivial, creating bottlenecks that render tools such as Methy-Pipe impractical for wide-spread healthcare usage.

We present two algorithms used heavily for sequence alignment problems: the FM-index and the Smith-Waterman algorithm. The FM-index is based on the BWT, an algorithm used within data compression and often adapted to perform bi-directional substring search; Methy-Pipe originally used a bi-directional BWT approach to sequence alignment, without any parallelisation. This led to poor performance over a typical work-load of 300M short reads, with the BSAligner module alone taking roughly 5 hours when running on a system with dual 12-core Intel Xeon processors.

In order to inform the reader about possible approaches to producing higher-throughput sequence alignment platforms, that can outperform CPU-based systems, we detail GPGPU and FPGA acceleration. It should be evident that these hardware platforms provide the potential to easily harness massive parallelism without the need to explicitly map out circuit designs. In the next chapter we will explore how these technologies have already been applied in the acceleration of Methy-Pipe, alongside related efforts.

Chapter 3

Related Work

3.1 Ramethy

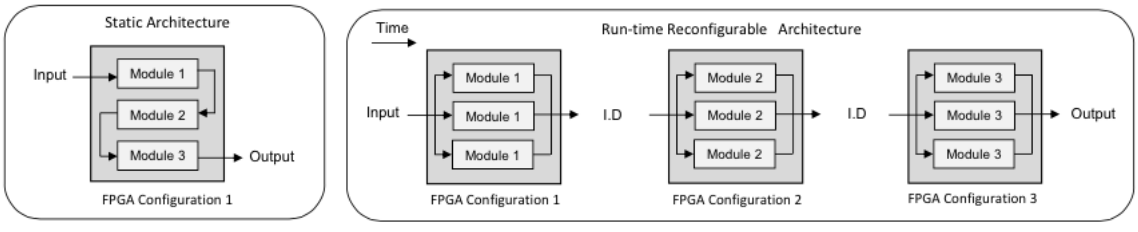
There is currently a novel runtime reconfigurable design targeting Methy-Pipe's BSAligner, as detailed by J. Arram, W. Luk and P. Jiang (Department of Computing, Imperial College London and Department of Chemical Pathology, The Chinese University of Hong Kong), at the FPGA 2015 conference, Monterey, CA, USA[64]. In this section we provide an overview of this design.

3.1.1 Reconfigurable Architecture

Ramethy exploits the reconfigurability of FPGAs, using distinct configurations for each module in the alignment pipeline, where intra-stage parallelism is improved by maximising the resources used by each algorithm stage. This is illustrated in F.1b. The user has great control over alignment parameters, as configurations can be re-ordered, added or removed at runtime to meet runtime demands such as the desired alignment workflow. This removes the potential for data hazards or unbalanced pipeline stages, the cost is loss of concurrent processing of the algorithm stages.

Many attempts to accelerate alignment pipelines on FPGAs involve statically configuring the circuit so that it is functionally equivalent to the desired alignment algorithm, producing several interlinked modules. Static configurations allow the interlinked modules to process data concurrently, however suffer numerous limitations that harm performance and utility:

1. **Data Hazards**: Alignment algorithms, as large modules in pipelined systems, naturally feature data hazards whereby execution of stages are depending on the execution of previous stages. If data hazards occur, then a subset of the modules may be left idle, reducing hardware efficiency in throughput and power usage.
2. **Module Latencies**: It is inevitable that different modules in the pipeline will take differing numbers of cycles to process a read. To balance the pipeline latencies, certain modules will be replicated more than others. However, as the other modules of the pipeline will be consuming a great deal of circuit space, the possibility for replication is limited.
3. **Extensive Resource Usage**: If a pipeline contains many modules that produce large circuits, the total static circuit may simply exceed the space available on the FPGA. Without dynamic reconfiguration, workload must then be held back on the CPU, which according to Amdahl's Law will hinder the overall system acceleration.
4. **Inflexible Alignment Parameters**: Static configurations inherently restrict the potential for controlling alignment parameters. Often the alignment reporting method, allowed mismatches and other parameters will be varied by the bioinformatician using an aligner, however substantial parameter changes may require the static circuits to be re-placed and re-routed (this can take days).



(a) Static architecture.

(b) Run-time reconfigurable architecture. Note: I.D denotes intermediate data.

Figure F.1: Types of FPGA pipeline architecture[64].

Ramethy uses an alignment pipeline composed of three stages. In stage 1, modules are designed that perform a function in the alignment algorithm. In stage 2 the modules are replicated to form an FPGA configuration, where the number of replications is specified simply by the following equation below.

Definition - Population of a module on an FPGA.

Let r_i be the fabric required for one module and A the total FPGA fabric area available:

$$P_i = \frac{A}{r_i} \tag{3.1}$$

In stage 3, for each module of the alignment algorithm, the steps are as follows: (1) each module has a configuration that is loaded onto the device, (2) data is streamed from the previous module (if there was one), (3) the data is processed concurrently and (4) the data is output into off-chip memory attached to the target device or simply stored in host memory. The performance of Ramethy's architecture can be measured simply using the equation below, where T is alignment time, N_i is the number of data items processed in the respective alignment stage, t_i is the time the corresponding module takes to process a single read and P_i the number of modules in the configuration. The overheads of the architecture are reconfiguration time t_r, and the data transmission time t_o, although these are deemed negligible for a typical alignment workload of roughly 300M reads.

Definition - Run-time reconfigurable architecture run-time.

Let t_r be reconfiguration time, t_o the data transmission time, n be the number of alignment modules, t_i be a given modules run-time, P_i the population of module i and N_i the number of data items for i:

$$T = \sum_{i}^{n} \left(t_r + t_o + \frac{N_i \times t_i}{P_i} \right) \tag{3.2}$$

3.1.2 Alignment Algorithm and Optimisations

Ramethy targets Methy-Pipe and consequently bisulfite sequencing read alignment. Specifically, the typical workload would involve aligning 300M reads of 75bp. The FM-index was chosen as the basis for its alignment algorithm. Ramethy presented a novel FM-index optimisation that improves the pattern matching performance, involving a new index structure that reduces the search steps and consequently computational complexity. The FM-index extended with backtracking was also optimised to improve the search space during inexact alignment. Both of these two optimisations are presented in this section.

3.1.2.1 n-step FM-index optimisation

The FM-index search algorithm in Alg.1 can be optimised. The basic implementation involves a for loop which steps through each character of the substring query R. After $|R|$ steps, a final interval is computed with consecutive indexes indicating areas of the suffix array with have R as a prefix. Chacón et al.[65] proposed a variation of this, the n-step index, which reduces the number of steps required. The number of steps required are reduced from $|R|$ by a factor n, by allowing n symbols in R to be matched in each iteration. This does however increase the computational complexity per step, and increases the size of the reference string's index. The n-step FM-index is illustrated in F.2.

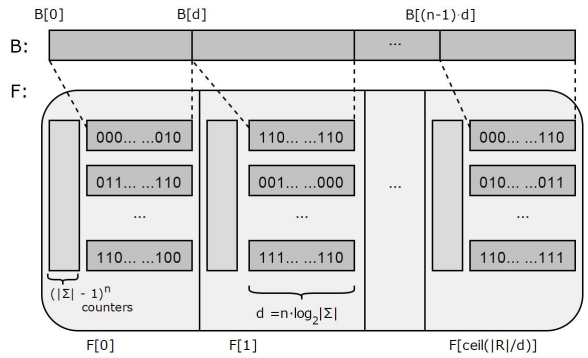


Figure F.2: n-step FM-index structure, where B is the BWT and F is the FM-index.

Ramethy extended the n-step FM-index for improved runtime and identical computational complexity to the basic FM-index search operation, using a compression and merging step when generating the n-step FM-index. Index generation for the reference genome broadly involves three steps: (1) compressing the reference genome into a reduced bitmap, (2) dividing the single BWT produced from merging n BWTs into buckets, as shown in Alg.2; and (3) interleaving buckets with their corresponding counters. This new FM-index is denoted as F, and its adapted search algorithm is shown in Alg.3. The cost of this optimisation is in the increased index size, which can be offset by larger bucket sizes d. The total memory usage can be calculated using the equation below, where a step of $n = 3$ and bucket size $d = 128$ results in the human genome consuming less than 10GB of memory.

Definition - Total memory usage of n-step FM-index.

Let n be the FM-index step, d the bucket size, $|R|$ the substring length, and Σ the reference string alphabet:

$$M = \frac{4 \times |R| \times (|\Sigma| - 1)^n}{d} + \frac{|R| \times n \times \log_2 |\Sigma|}{8} \text{ Bytes} \tag{3.3}$$

Algorithm 2 Generation of merged BWT

Input: String S, reduced bitmap S_r and suffix array SA_S
Output: Merged BWT B

```
/* generate n BWTs: b_1, b_2, ..., b_n */
for i ← 1 to n do
    for j ← 0 to |S_r| do
        b_i[j] = S_r[(SA_S[j] − i)%|S_r|]
    end for
end for
/* merge b_1, b_2, ..., b_n to form B */
for i ← 0 to |S_r| do
    B[i] = concat_bits(b_n[i], .., b_1[i])
end for
return B
```

Algorithm 3 n-step FM-index search algorithm

Input: Reduced bitmap of substring R_r, string S, F_S and SA_S
Output: $positions$ where R is a prefix in S

```
low ← 1
high ← |S|
for i ← |R| − 1 to n, step −n do
    str ← concat_bits(R_r[i − (n − 1)], .., R_r[i − 1], R_r[i])
    low ← F[low − 1/d].counters(str)
          +C(str, F[low − 1/d].B, low − 1%d)
    high ← F[high/d].counters(str)
          +C(str, F[high/d].B, high%d)
end for
for i ← low to high do
    positions_i ← SA_S[i + low]
end for
```

3.1.2.2 Bi-directional backtracking

Ramethy uses an extended n-step FM-index search operation to support inexact read alignment, i.e. alignment whereby substitutions, insertions or deletions (edits) have occurred in the read. This is done through introduction of backtracking: the state of a read is captured in a stack, while performing edits to try and achieve exact alignment. A naive brute force approach would have resulted in the search operations scaling exponentially with the number of edits allowed.

However, Ramethy extends the FM-index search by generating an FM-index for the forward and reversed reference genome, allowing both backward and forward search operations respectively. By placing constraints on the mismatch position in a read, an efficient bi-directional search can be performed where much of the search space can be removed by exact matching long segments. As Methy-Pipe's BSAligner has the alignment constraint that no more than two mismatches can occur, a simple approach to segmenting a read failing to inexactly align is used. This involves dividing the read into two segments, and forward or reverse exact matching half of the read before the search is extended with edit permutations (see F.3). This approach is based upon the 2-way BWT structure proposed by T.W. Lam et al.[66].

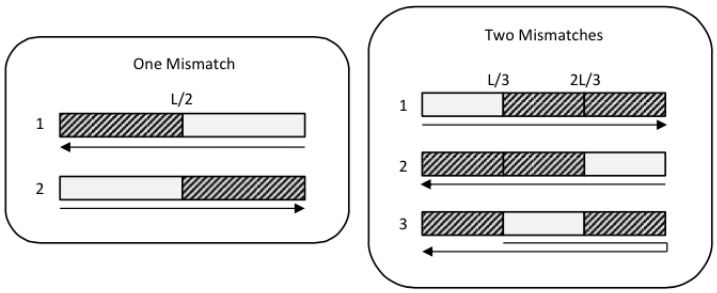


Figure F.3: One/Two mismatch alignment phases - arrows indicate search direction and reads are represented by rectangles, with shaded segments indicating where mismatches are tested and non-shaded segments indicating exact matches[64].

3.1.3 Alignment Architecture

The alignment parameters used in Methy-Pipe ensure (1) reads must be aligned to the reference genome with no more than two mismatches, and (2) reads must be aligned uniquely. Ramethy supports these parameter through three modules based on the n-step FM-index: exact match, one mismatch and two mismatch alignment. The designs overall performance is improved by pipelining the module operations to achieve high throughput and replicating modules on the FPGA to achieve high parallelism.

Ramethy runs on the Maxeler MPC-X1000 data-flow node, which provides up to 8 DFEs in a 1U form factor, with power utilisation similar to a single high-end server. Each DFE is composed of a single Altera Stratix V FPGA with 48GB of dynamic RAM (DRAM), where the DRAM consists of 6x8GB memory modules that give a 384 byte world length. A single memory controller is used to manage the read and write operations.

3.1.3.1 Exact Alignment

In the case of the exact match module, Alg.3 is simply performed on hardware. The human genome index is too large to fit in on-chip block RAM (BRAM), so is stored on off-chip DRAM attached to the FPGA (see F.4a). Accessing this memory introduces latency, which alongside FM-index iteration interdependence results in a non-full pipeline. The high level design for this module is presented in F.4b. In each cycle, the low and $high$ index elements from F are accessed from off-chip DRAM via a data buffer. The index data, current $(low, high)$ interval and relevant read symbols can then be used to compute the next interval. These new interval values overwrite the previous values in the circular buffer, and the index elements are streamed to an off-chip memory command generator. After the required number of alignment steps have been executed, the final interval is transferred to the host.

(a) General module design (b) Exact match module

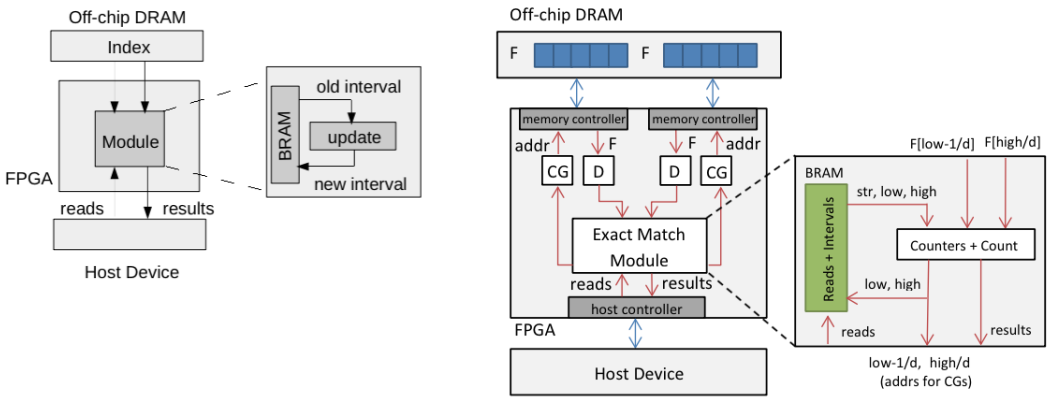


Figure F.4: Ramethy module designs - data buffers are denoted by D, and memory command generators are denoted by CG[64].

3.1.3.2 Inexact Alignment

The one and two mismatch modules are responsible for inexact matches, and are extensions of the exact match module; additional logic is simply required to control FM-index backtracking. To support the second Methy-Pipe alignment parameter of only permitting unique alignments, all possible mismatch positions in each read must be tested. A bread-first approach is taken to backtracking, with the aforementioned 2-way BWT algorithm being used to avoid the exponential number of alignment steps required for the number of mismatches allowed. This requires use of the reversed reference genome. Note that this stage is terminated after a reference position is found with one mismatch, or two mismatches if the former could not be identified.

3.1.3.3 Alignment Workflow

Before alignment, all the reads are loaded onto host memory and compressed into reduced bitmaps. Initial intervals are initialised, and precomputed for excess symbols where read lengths aren't divisible by the index step n. Once this pre-processing is complete, the FPGA device is configured for exact match. The reduced bitmap reads and initial index intervals are streamed to the FPGA, and the final intervals streamed back to the host once the concurrent processing has been completed. Depending on whether the interval indicates reads as unaligned or not, they may be filtered; the FPGA is then configured for one mismatch, and needs to only handle the unaligned reads. Again, the host can filter out reads that have been aligned, as they are indicated by a single index interval (the number of hits to determine alignment uniqueness is also returned). The process is repeated for two mismatches. Finally, all the intervals can be converted using the suffix array to reference genome coordinates.

3.1.4 Results

The performance of Ramethy was analysed using an FM-index with a step of $n = 3$ and bucket size $d = 368$ (consuming 3.3GB of memory). The runtime, energy consumption and alignment accuracy of Ramethy was compared primarily to that to aligners widely regarded as some of the fastest on their platforms: (1) SOAP2 running on a 1U server rack with dual Intel Xeon E5-2650 CPUs and (2) SOAP3-dp running on a NVIDIA GTX-580 GPU. For all tests, 10M 75bp reads were used. This is only 3% of a typical alignment workload, so the reconfiguration time of roughly 12s was not included due to the negative bias it could have introduced. Realistically, this time would also only constitute about 3.5% of the total alignment with a typical workload.

Using 10M 75bp bisulfite sequencing reads of the human genome simulated by Sherman[67], results show a 14.9 times speedup compared to SOAP2 running with 16 threads on dual Intel Xeon E5-2650 CPUs and a 3.8 times speedup compared to SOAP3-dp running on a NVIDIA GTX 580 GPU. However, upper-bound performance estimations suggest the MPC-X1000 design could achieve a maximum speedup of 88.4 times that of SOAP2 and 22.6 times that of SOAP3-dp. These upper-bound estimations are based on the situation whereby the MPC-X1000 memory modules are detached, allowing for 3 modules per configuration. It is also worth noting, experiments indicated Ramethy's reconfigurable architecture exceeds the performance of a static design, operating 11% faster when accounting for reconfiguration time. Due to its lower operational clock frequency and shorter alignment time, Ramethy uses 72W, which is roughly 3 times less total power than both SOAP2 and SOAP3-dp. The energy consumption was 1.5kJ, which was over 8 times less than SOAP3-dp and over 21 times less than SOAP2. Ramethy improves both runtime, power and energy consumption without compromising accuracy; it achieves identical alignment accuracy to the other two platforms.

As an FPGA accelerated application, Ramethy's alignment time scales linearly with read count. Therefore, Ramethy's runtime can be linearly extrapolated to that of a typical workload. Compared to the previous alignment time of 5 hours with dual 12-core Intel Xeon CPUs, the extrapolated result is an impressive 6 minutes. The upper bound estimate would give a time of just over 1 minute. These alignment times could have a significant influence on the diagnosis times, response times and consequently patient samples analysed per day.

3.2 Fernandez, Najjar and Lonardi

Fernandez et al.[68][69] have proposed designs also based on the FM-index. In [68], the index of a small reference genome is stored in on-chip BRAM, which can align 1000 reads in 60.2μs on a single Xilinx Virtex-6 FPGA. This work is extended in [69] to allow for approximate alignment, with modules for exact match, one mismatch and two mismatches alignment. For every n permitted mismatches, $n+1$ exact string matchers populate an FPGA device in a statically configured pipeline. When mismatches are detected, the incompatible symbol is substituted with the alternate possible symbols from the reference genome alphabet and sent towards the next exact matcher for re-evaluation. This design is built upon the Convey HC-1 platform, and can align 18M reads in 138s.

3.3 Olson et al

Olson et al.[70] have proposed a design based on the Smith-Waterman algorithm. They pre-compile an index in hardware that allows the mapping of seeds (22bp) of a short read (76bp) to locations in the reference genome where the seed occurs. The index is a modifiable hash table, composed of a candidate alignment location (CAL) table and pointer table, and implemented in hardware. Short reads are scored against the reference genome at each of the CALs using the Smith-Waterman algorithm. The Smith-Waterman scoring tables are also performed in hardware as a systolic array, as scoring table cells can have their values computed in anti-diagonal waves[71]. The design is built upon a platform with 8 Pico M-503s, each using Xilinx Virtex-6 FPGAs. This design aligns 50M reads in 34s.

3.4 FPGA aligner comparison

Different alignment solutions use different alignment parameters, and FPGA solutions are not easily accessible for testing, so a normalised performance metric is used to compare the different platforms. This is bases aligned per second (bps), and allows for a fairer comparison (see equation below). Although Ramethy is designed for bisulfite sequencing analysis, with alignment parameters such as total permitted mismatches being specialised to that of Methy-Pipe's BSAlign, it performs alignment in a rather generalised manner. The difference is that, for bisulphite sequencing, due to loss of DNA reverse-complementarity the aligner must be run twice for each strand of DNA. However, bps is a normalised metric that examines solely alignment throughput, which despite the differing applications of sequences aligners, largely allows comparison.

Definition - Bases aligned per second.

Let $|r|$ be read length, N_r the read count and t_a the total alignment time:

$$bps = \frac{|r| \times N_r}{t_a} \tag{3.4}$$

T.I contains bps counts presented in [64] to compare the aforementioned FPGA alignment platforms. All of the systems tested are housed in a 1U rack unit, suggesting that Ramethy could offer the highest performance per unit volume.

Program	Platform	$\|r\|$	N_r (M)	Clock F (MHz)	Devices	Time (s)	bps (M)	Speedup
SOAP2	Intel E5-2650	75	10	2000	2	168	4.5	-
SOAP3-dp	GTX-580	75	10	772	1	43	17.4	3.9x
Fernandez[69]	Convey HC-1	101	18	150	4	138	13.2	3.0x
Olson[70]	Pico M-503	76	54	250	8	34	120	26.8x
Ramethy	MPC-X1000	75	10	150	8	11.3	66.4	14.9x
Ramethy (U)	MPC-X1000	75	10	150	8	1.9	395	88.4x

Table T.I: FPGA aligner performance comparison.

3.5 Summary

In this chapter we presented Ramethy, a novel reconfigurable architecture for accelerating bisulfite sequence alignment. The architecture targetted Methy-Pipe, replicating its alignment parameters: (1) reads must be uniquely aligned and (2) alignments cannot have more than two mismatches. The architecture was also specialised in hardware for 75bp reads, as these form the typical workload for Methy-Pipe.

The design was based on the FM-index, and extended with the n-step optimisation and bi-directional backtracking for inexact alignment. Ramethy was implemented on the Maxeler MPCX-1000 platform, using 8 FPGAs. It outperformed the CPU-based SOAP2 with a 14.9$\times$ speed-up, and outperformed the GPU-based SOAP3-dp with a 3.8$\times$ speed-up. This was achieved whilst using an order of magnitude less energy, and achieving identical alignment accuracy. For a typical BSAligner workload of 300M bisulfite sequencing reads of 75bp, Ramethy would align the reads in roughly 6 minutes, whereas the original implementation took 5 hours. In light of these results, it is evident that Ramethy is a highly competitive sequence aligner and a strong candidate for integration into not just Methy-Pipe, but other bioinformatics pipelines.

However, Ramethy's design is limited in accuracy of inexact alignment, as it terminates after a reference position is found with one mismatch, or two mismatches if the former could not be identified. This renders the inexact algorithm stage unsuitable for paired-end alignment, as

all candidates positions would be required for checking repeat regions on the reference sequence. It is therefore appropriate to extend Ramethy's design to report all potential inexact alignment positions, such that the host can manage the results as desired. This will increase the number of alignment positions checked by the inexact alignment module, increasing the amount of backtracking performed and consequently reducing the design's runtime and bps. The alignment design therefore requires further optimisation to remain highly competitive.

Chapter 4

Optimisation of Short Read Alignment using FPGAs

In this chapter we present a runtime reconfigurable (FPGA) alignment architecture extending and optimising Ramethy (chapter 3), again targeting Methy-Pipe's BSAligner. The design achieves higher alignment throughput than other FPGA-based alignment designs, aligning 144.2M bases per second (bps), whereas the most competitive efforts of Olson et al. [70] achieved 120.7bps (the differences in FPGA technologies used are discussed later in the section). The design reduces the overall alignment time of BSAligner from 5 hours to 13 minutes. The novel aspects of this contribution are:

1. Oversampling of the FM-index, which prevents some unnecessary random accesses to FM-index buckets in DRAM. This comes at the cost of an increased index size.
2. Seed and compare alignment stage, which uses direct string comparison to reduce the amount of inefficient FM-index backtracking used in inexact alignment.

This work is also presented in [9] by J. Arram, T. Kaplan, W. Luk and P. Jiang (Department of Computing, Imperial College London and Department of Chemical Pathology, The Chinese University of Hong Kong). It has been submitted to IEEE Transactions on Computational Biology and Bioinformatics for review. All the newly introduced optimisations used are demonstrated in this chapter for the sake of completeness, with the interval store kernel (4.2.2) and the compare kernel of the seed and compare module (4.2.3) comprising my contribution to the alignment architecture.

4.1 Alignment Algorithm Overview

The alignment algorithm is composed of four stages, which consecutively filter an initial set of reads (in .fastq format) into a file annotated with alignment information ready for methylation calling and subsequent downstream analysis (in .bsalign format). The four stages are: (1) exact read alignment, (2) seed and compare, (3) one mismatch read alignment and (4) two mismatch read alignment. This is demonstrated in F.1. These stages are similar to those presented in Ramethy, meeting BSAligner's alignment constraint of reads being uniquely aligned to the reference genome with no more than 2 mismatches.

The reads are initially passed through the exact alignment stage, to identify reads with no mismatches. This stage uses the n-step FM-index, presented in 3.1.2.1, and has been further optimised as in a typical workload it operates on the entire set of reads. Unaligned reads are deemed to have at least one mismatch, so are passed onto the inexact alignment stages, in particular the new seed and compare stage. This involves splitting each read into three sections that are aligned to the reference genome, to check how many positions each seed aligns to. If the total sum of these reference positions is below a specified threshold, the read is compared directly against the reference genome at these positions. Any reads with a sum of reference positions exceeding the threshold are passed to the one and two mismatch stages, which perform backtracking with the FM-index. These modules are more efficient for inexact alignment with numerous candidate reference positions.

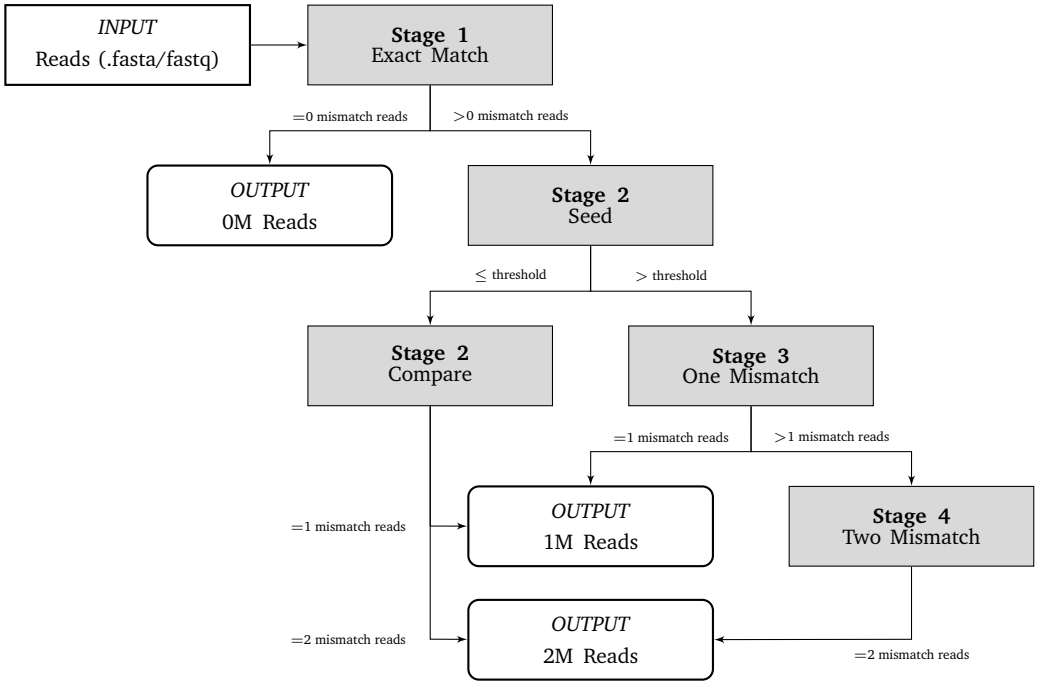


Figure F.1: Alignment algorithm stages - 0/1/2M represent 0, 1 and 2 mismatches in alignment.

4.2 Alignment Optimisations

Prior to introduction of the seed and compare stage, all alignment stages were reliant on the n-step FM-index search operation shown in Alg.3; in the exact stage it is used conventionally whereas in the one and two mismatch stages it is extended with backtracking. Naturally, this makes the FM-index search operation one of the alignment pipeline's bottlenecks, and a major consideration in alignment optimisation.

To simplify the explanation of optimisations, the notation used is summarised in T.I. Similarly to Ramethy, this new alignment architecture has a typical workload consisting of 300M short bisulphite sequencing reads (75bp), so these optimisations involve specialising the hardware acceleration for this 75bp read length. For each optimisation presented, it will be specified when it cannot be used in a general-length alignment approach.

Symbol	Explanation
L	Length of a read (bp)
L_R	Length of the reference genome (bp)
Σ_q	Alphabet size of reads, the query text
Σ_t	Alphabet size of the reference genome, the text
n	FM-index step size, number of symbols the suffix array interval is updated per search iteration
b	Number of symbols that the suffix array intervals are updated for using precomputed values
$\mathcal{L}$	Average number of symbols before the low and high interval values tend to the same FM-index bucket
d	Size of an FM-index bucket
f	Sampling factor of an FM-index bucket

Table T.I: Optimisation analysis disambiguation.

4.2.1 Oversampling

Random accesses to the FM-index are costly, as the C-depleted human reference genome requires significant storage space and consequently storage in off-chip DRAM. This is because in most FPGA architectures, many GBs of data storage are allowed in DRAM (off-chip large memory, LMem) but only several MBs of BRAM (on-chip fast memory, FMem), and off-chip DRAM accesses have a latency in the order of hundreds of cycles. In the n-step FM-index search algorithm, the size of the suffix array interval ($high - low$) will decrease or remain the same after each search iteration. This is achieved by accessing the FM-index twice each search iteration to update the high and low intervals using the respective FM-index bucket - these two DRAM accesses become very costly.

However, the interval values often tend to the same FM-index bucket after less than 10 iterations. Ideally, this would mean both low and $high$ could be updated using the same FM-index bucket, and consequently a single access to DRAM. If the number of iterations taken to tend to the same bucket is $\mathcal{L}$, and the read length is L, then the total number of random accesses is reduced from $2L$ to $2\mathcal{L} + (L - \mathcal{L})$. Clearly, if the read length is 75bp or longer, and a single bucket is required to update low and $high$ after less than 10 FM-index search iterations, there could be a drastic reduction in unnecessary off-chip memory accesses.

Currently the n-step FM-index does not currently accommodate this. Unnecessary random accesses would need to be reduced by removing cases where two adjacent buckets have values of low and $high$ at their respective boundaries. This means instead of two FM-index accesses being required to update low and $high$ interval values, a single FM-index access would be required.

Proposition: Similar in motivation to the the use of oversampling in signal processing, where increasing the frequency of signal sampling improves resolution of a signal, we can oversample the FM-index. This would involve novel sampling buckets from the BWT every d/f symbols, where d is the BWT length in symbols and f is the new sampling factor. This means the end of a given FM-index bucket will have the first d/f symbols from the following bucket. To relate this to signal processing, this means a given bucket has a more representative sample of the complete BWT due to increased bit length (similar to a signal's bit depth), so the overall resolution improves. Unless the interval values are greater than or equal to d/f apart, as f increases, it becomes increasingly likely a single bucket can be accessed. Unsurprisingly, this increases the index size by a factor of f, such that if $f = 2$, $n = 3$ and $d = 256$, the reference index would use roughly 9GB of memory (shown below in equation 4.1). Note that this is still significantly less than the DRAM available on many FPGAs, and most importantly Maxeler DFEs.

The algorithm for the FM-index search operation can be adapted to Alg.4 (for the sake of demonstration ignoring the n-step). If the suffix array interval is less than d/f, i.e. $high - low < d/f$, the low and $high$ values will be able to access the same FM-index bucket, resulting in only one off-chip access to the corresponding FM-index bucket being required to update the suffix array interval for the remaining symbols. Again, this is because thanks to oversampling the BWT when creating these buckets, the regions of the BWT required to update the interval values are more likely to be accessible within a single bucket. Note that we evaluate the performance provided by oversampling in detail in 5.5.3.2.

Fernandez et al. [69] store a small reference index (chromosome 14) in BRAM, whereas our design uses the larger DRAM to accommodate an entire human genome index. The support of whole-genome indexes is crucial, with the rise of whole-genome sequencing analysis (such as bisulfite sequencing analysis). Alongside the n-step, the use of FM-index oversampling is a novel way to reduce expensive off-chip accesses to DRAM. This problem would not necessarily be faced by designs using BRAM to store the FM-index.

Definition - Total memory usage of oversampled n-step FM-index.

Let n be the FM-index step, d the bucket size, $|R|$ the substring length, f be oversampling factor and Σ the reference string alphabet:

$$M \approx f \cdot \left(\frac{4 \times |R| \times (|\Sigma| - 1)^n}{d} + \frac{|R| \times n \times \lceil \log_2 |\Sigma| \rceil}{8} \right) \text{ Bytes} \tag{4.1}$$

Algorithm 4 Oversampled FM-index search algorithm

Input: Substring R, reference string S and FM-index F_S with bucket size d an sampling factor f
Output: SA interval where R is a prefix in S

Function: $calc\_interval(s, F_b, v)$ returns an updated interval using FM-index bucket F_b, interval (low/high) value v and symbol s.

$low \leftarrow 0$
$high \leftarrow |R|$
for $i \leftarrow |R| - 1$ to 0 **do**
  if $high - low \geq d/f$ **then**
    $low \leftarrow calc\_interval(R[i], F_s[(low1)/(d/f)], low1)$
    $high \leftarrow calc\_interval(R[i], F_s[high/(d/f)], high)$
  else
    $low \leftarrow calc\_interval(R[i], F_s[(low1)/(d/f)], low1)$
    $high \leftarrow calc\_interval(R[i], F_s[(low1)/(d/f)], high)$
  end if
end for

4.2.2 Interval store

In computational genomics, k-mers are all the possible subsequences of length k that could be obtained from a DNA read. For example, the read AGCTAG could produce 3-mers {AGC, GCT, CTA, TAG}. For a given k, the set of possible k-mers in string of length L is $L - k + 1$, where L is the string length. Given an alphabet of size Σ, the total possible k-mers is $(\Sigma)^k$. For each k-mer, the corresponding suffix array intervals can be precomputed and stored.

Proposition: Given a section of a read with b symbols, the corresponding suffix array interval has only $(\Sigma_q)^b$ values. This b can be chosen such that all the precomputed suffix array intervals can be stored using the few MB of BRAM storage available on the Maxeler DFEs. Within a cycle, Maxeler limits the number of write operations that can be made, or writes in conjunction with reads, but several reads can be made. This allows the suffix array interval to be updated for b symbols efficiently, and reduces inefficient random accesses to off-chip DRAM from $2L$ to $2(L - b)$. This could be performed for roughly 9 symbols of each read, given the limitations of BRAM storage of 8MB. This number was arrived at by the total number of suffix array interval entries and the maximum of 64 bits required to store each interval, where low and high each require 32 bits.

Generalisation Issue - This optimisation could not be used in a general length alignment platform in conjunction with the n-step FM-index optimisation.

4.2.3 Seed and compare

In 3.1.2.2 a bi-directional backtracking extension was introduced to the FM-index, specifically a variant on a 2-way BWT, allowing inexact alignment. This reduced excessive backtracking through restricting the sample space in which read edits had to be tested for exact alignment. However, the one and two mismatch stages are still comparatively slower than the exact match stage due to the fact they extend the n-step FM-index search further.

Proposition: An additional pipeline stage, seed and compare, can be introduced to remove some of the workload passed to the one and two mismatch stages. More specifically, this reduces the amount of inefficient backtracking taking place for inexact alignment. This seed and compare module will efficiently manage the inexact alignment of reads with few candidate alignment locations on the reference genome. There are four stages to the seed and comparison optimisation, illustrated in F.2:

1. Each read is split into three segments (seeds). If the read length is L, then the seeds will have ($start$,end)s of : (0, $L/3$), ($L/3$, $2L/3$), ($2L/3$, L).

2. These seeds are individually aligned to the reference genome using exact match. This identifies positions where regions of the individual read could occur without mismatch in the reference genome.

3. Given we know at least one of the seeds will contain a mismatch symbol, the aligned positions found by the three seeds must be collated and normalised. For example, if a_i stands for a position where ith seed has aligned, and u_i means there is not a connected alignment position for i, we are looking for patterns of contiguous alignments (separated by $L/3$ bases) such that: $a_1a_2u_3$, $u_1a_2a_3$, $a_1u_2a_3$, $a_1u_2u_3$ etc. In F.2, we see these valid patterns: $a_Aa_Bu_c$ and $a_Au_Ba_C$.These patterns indicate a mismatch has occurred in one or two of the seeds, and one or two seeds have aligned. This qualifies as a valid inexactly aligned read. However if all three seeds do not align to the reference genome, it can be inferred that a read contains at least three mismatches, which does not meet BSAligner's alignment constraint of up to two mismatches, so the read is discarded.

4. We are left with a series of positions on the reference genome for each read, which can now be compared to identify the specific mismatches. In F.2, these are X and Y, the positions of the first seeds in the valid alignment patterns. Assuming the positions to be compared against are below a specified threshold (i.e. fewer than a given number of positions need to be considered), direct string comparison can be used to inexactly match the read and identify the locations and symbols where mismatches have occurred. If any read has a greater number of reference positions than the threshold, then it will be passed on to the one and two mismatch stages.

The direct string comparison can be performed in a highly parallel manner on FPGAs, as each equality checks can take place over each of a read symbols in parallel. If few positions need to be checked using this direct string comparison, it should be more efficient than the backtracking peformed by the one and two mismatch stages.

This seed and compare stage is novel, as seed and expansion approaches are conventionally used when the alignment algorithm is built upon hash tables. Seeds are often shorter sequences of reads, and the hash tables record the location of these seeds in a reference genome. The entire read can be verified at the locations recorded on the hash table, for example using the Smith Waterman algorithm. This was demonstrated by Olson et al. in [70]. However, seed and comparison approaches have not been used by FM-index based designs to reduce inefficient backtracking.

For example, Fernandez et al. inefficiently support inexact alignment in [69] by re-using exact alignment modules to match all permuted reads.

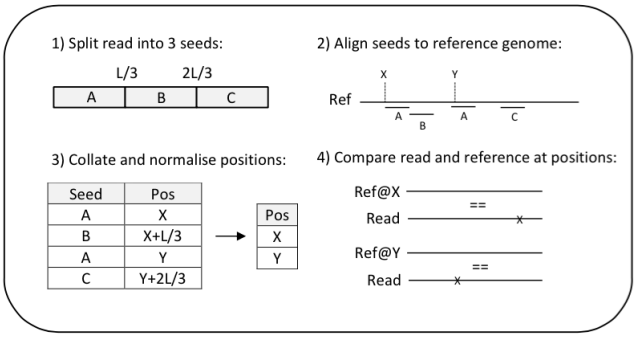


Figure F.2: Seed and compare stages, where L is the length of a read.

In order to gauge the viability of the seed and compare stage, we altered the software simulation of the alignment pipeline so that the number of candidate locations in the reference genome for inexactly aligned reads would be written to file. Using this information, it was possible to plot how many reads had particular numbers of positions to be checked. In F.3 this graph is presented for a set of 10M bisulphite reads generated by Sherman from the human genome[45], to demonstrate the sheer volume of reads with few reference positions to be checked. Although many reads had in the order of thousands of potential reference positions, 70-80% fall within a threshold of 10, which could mean a significant inexact alignment acceleration using the seed and compare stage.

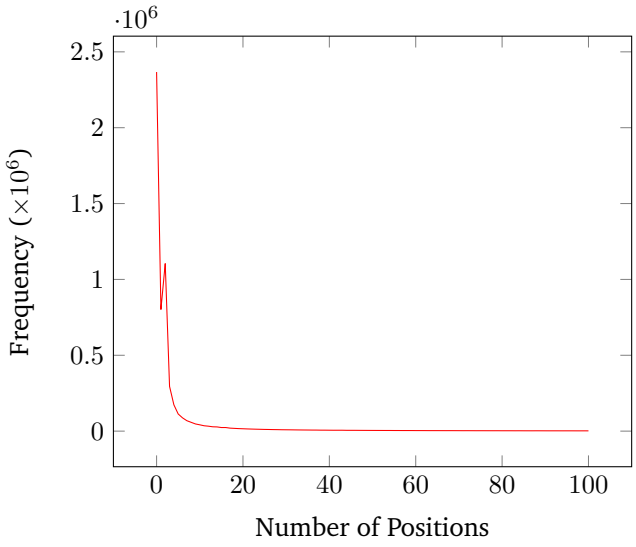


Figure F.3: Reference hits for hg19, using 10M reads produced by Sherman. The number of candidate positions for inexact alignment hits is plotted against the frequency of occurence. Note that any reads with more than 100 positions were discounted for graph clarity.

4.2.4 Optimisation Summary

The FM-index alone results in $2L$ random accesses to off-chip memory every search iteration. The pre-existing n-step optimisation reduces this significantly to $2L/n$. The interval store would reduce this to $2(L-b)$, for a small b, which results in a reasonable improvement. Oversampling would reduce this to $2\mathcal{L}+(L-\mathcal{L})$, a significant improvement. All these optimisations combined reduce the

total accesses to off-chip DRAM to $\frac{1}{n} \cdot (2 \cdot (\mathcal{L} - b) + (L - \mathcal{L}))$. Given BSAligner's typical workload, L is 75bp, the current FM-index step is $n = 3$ and the interval store pre-computations will support $b = 9$ to remain within 8MB of BRAM. Overall, these optimisations reduce the total off-chip accesses from 150 by around 6 times if $\mathcal{L}$ falls within 15-20 symbols, i.e. if it takes an average of 15-20 search iterations (over roughly a fifth of a reads symbols) for the low and high interval values to tend to the same bucket. This corresponds to fewer clock cycles by an order of six hundreds.

4.3 Hardware Design

In this section we present the hardware designs used to implement BSAligner with FPGA acceleration. This includes an explanation of the hardware platform being targeted (the Maxeler MPC-X2000 dataflow node), the high level module design for exact and inexact alignment, alongside the implementation of the interval store and seed and compare optimisations. The modules composing the alignment platform are built as a run-time reconfigurable architecture, as described in 3.1.1, where each module has its own FPGA configuration which is replicated using all the resources available on the FPGA devices when required.

4.3.1 Maxeler MPC-X2000 platform

The hardware design targets the Maxeler MPC-X2000, which is part of the MPC-X dataflow node series with architecture shown in F.4. It provides up to 8 maia DFEs, with a 1U form factor and the power consumption of a high-end server. Each DFE is comprised of a single Altera Stratix V FPGA and up to 96GB of DRAM, with a single on-chip memory controller. These DFEs can all communicate directly with one another using a dedicated high-speed interconnect, the MaxRing. Similar to a GPU, the DFEs are a shared resource on a network and simply connected to the host CPU via by dual fourteen or quad data rate (FDR or QDR) Infiniband. Infiniband is a high throughput and low latency interconnect that uses switching fabric technology, as opposed to a shared medium such as Ethernet. This is combined with remote direct memory accesses (RDMA) to ensure memory transfers take place without inefficient memory copies.

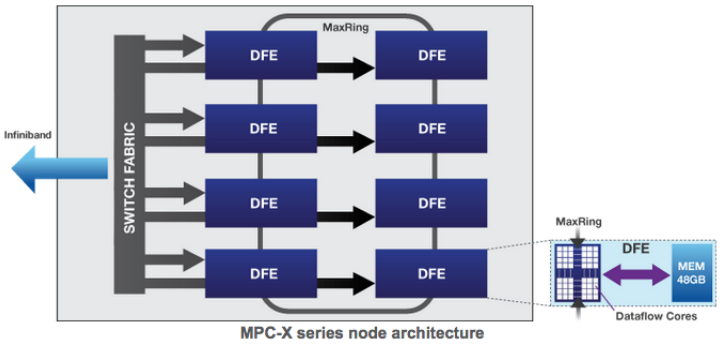


Figure F.4: Maxeler MPC-X dataflow node architecture [72].

4.3.2 Interval store

The interval store is accessed using a FPGA kernel (BramKernel.maxj) which takes as input the bit compressed reads and their ids from the host CPU, and as output provides an exact match module with reads and the pre-computed suffix array interval values of low and $high$. The kernel is used for both the seed and exact match modules. Specifically, for the exact match module the output stream connects to the exact match module 1 kernel (Em1Kernel.maxj), which has its functionality described in the following section. The code from the exact match kernel manager responsible for this stream mangement is shown in Alg.4.1.

The way in which the interval store is accessed by the interval store kernel is shown in Alg.4.2. The first 9 bits are extracted from the bit compressed read, and each symbol's value is multiplied according to the possible number of k-mer permutations possible to create an address. This is derived from the fact that $(\Sigma_q)^b$ possible suffix-array values can exist for the reads first b symbols, where Σ_q is 3 (A, G, T) due to C-depletion, and entries are stored lexicographically. The access itself uses Maxeler fast memory (FMem), where the low and $high$ stores are mapped ROMs from the host CPU. To update the $high$ and low values for the first 9 symbols, these two stores can just be read in a highly abstracted manner given the address computed using the symbol values.

```
KernelBlock kBram = addKernel(new BramKernel(makeKernelParameters("BramKernel")));
KernelBlock kEm1 = addKernel(new Em1Kernel(makeKernelParameters("Em1Kernel")));

DFELink readInput = addStreamFromCPU("readInput");
kBram.getInput("readInput") <== readInput;

DFELink bramOutput = kBram.getOutput("alignOutput");
kEm1.getInput("readInput") <== bramOutput;
```

Code Extracts 4.1: Extract from Em1Manager.maxj - interval store stream management.

```
DFEVector<DFEVar> readSym = readInput.get("readSym");
readSym = readSym.rotateElementsRight(readLength - 9);

// interval store address = sym1 + (3*sym2) + (9*sym3) + (27*sym4) + (81*sym5)...

DFEVar storeAddr = dfeUInt(15).newInstance(this);
DFEVar multiplier  = constant.var(dfeUInt(13), 1);
for (int i = 8; i >= 0; ++i)
{
    storeAddr +=  readSym[i].cast(dfeUInt(15)) * multiplier;
    multiplier = (multiplier << 1) + (multiplier << 0);
}

readSym = readSym.rotateElementsLeft(readLength - 9);

Memory<DFEVar> lowStore = mem.alloc(dfeUInt(32), memSize);
lowStore.mapToCPU("lowStore");
Memory<DFEVar> highStore = mem.alloc(dfeUInt(32), memSize);
highStore.mapToCPU("highStore");

DFEVar low = lowStore.read(storeAddr);
DFEVar high = highStore.read(storeAddr);
```

Code Extracts 4.2: Extract from BramKernel.maxj - BRAM address computation for interval store accesses.

4.3.3 Exact alignment modules

The exact alignment stage in the alignment pipeline is composed of two modules, whereby the first has been optimised with the interval store. These modules will be detailed separately, and are illustrated in F.5. Note that these modules can be merged to form a single module, yet these are presented seperately for clarity.

(a) Exact match module 1. (b) Exact match module 2.

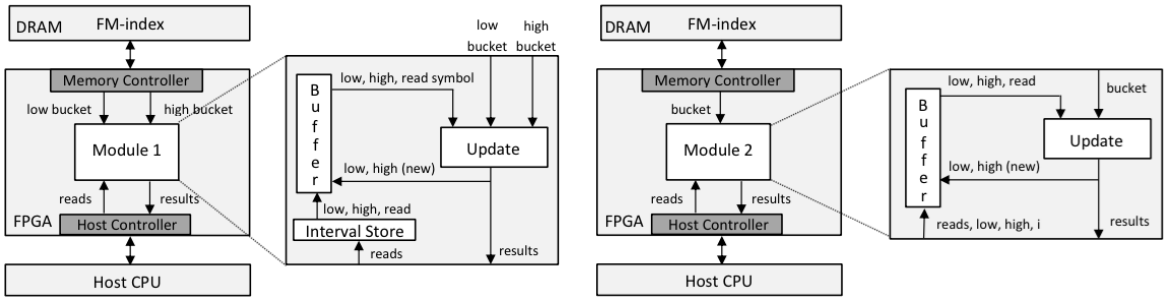


Figure F.5: Exact match module hardware designs[9].

4.3.3.1 Module 1

Module 1 is based on the FM-index search algorithm, and extended with the interval store. This optimisation allows the suffix array interval to be updated for the first 9 symbols of each read, through use of pre-computed interval values stored in BRAM. The appropriate symbols are extracted from each read passed through the input stream, and converted into an address, allowing the corresponding values of low and $high$ to be accessed from BRAM. The suffix array interval must then be updated for the remainder of the symbols, $L - b$, using the standard FM-index search.

The FM-index is stored in off-chip DRAM, and in each search iteration two buckets accesses are required. Two streams to DRAM are used to update the suffix array interval using low and $high$, halving the DRAM bandwidth. To propagate the current values of low and $high$ into subsequent computation, a circular buffer is used. This is also able to store multiple reads, allowing loop pipelining.

Following oversampling of the FM-index, a few different cases each search iteration must be considered. If $high - low < d/f$, the suffix array interval values have tended to the same FM-index bucket, and the result is streamed back to the host CPU where it can be managed by module 2. If $low > high$, a symbol is unaligned and indicates a mismatch, terminating the FM-index search and streaming the result back to the host. Alternatively, the read may exactly align, again requiring the result to be streamed back to the host.

4.3.3.2 Module 2

If $high - low < d/f$, the suffix array interval can be updated for the remaining read symbols using module 2. This module is also based on the FM-index search algorithm, however only one stream to DRAM is required as only one FM-index bucket must be accessed. This allows the single stream to fully utilise the DRAM bandwidth. The search operation continues until the entire read is aligned, or $low > high$, indicating a mismatch. The result is then streamed back to the host CPU. If a mismatch has occurred, the read is passed to the new seed and compare stage, or alternatively if it has been aligned it can have its suffix array interval converted into a reference genome position at the host.

4.3.4 Seed and comparison module

The seed and compare stage in the alignment pipeline is composed of two modules, seed and compare, each with their own kernels and corresponding managers. The stages of seed and comparison were previously illustrated in F.2.

4.3.4.1 Seed

The read is first split into 3 seeds, which are exact aligned using the FM-index search. Similar to exact match module 1, the suffix array interval for the first several symbols of these seeds can be precomputed using an interval store, and the FM-index on off-chip DRAM must be used to update the suffix array interval for the remainder of the symbols. The seed kernel extends the exact match module 1 with the management of the 3 seeds. The alignment positions of the 3 seeds are streamed back to the host CPU, where the suffix array intervals can be converted into reference genome positions.

4.3.4.2 Compare

If the total number of reference genome positions collated over the three seeds is less than a pre-specified threshold, the read and reference are directly compared at each position using the compare kernel (CompareKernel.maxj). Reads are streamed to this kernel with an equal length segment from the reference genome, corresponding to the candidate alignment position. The string comparison used is an incredibly simple equality test over each character, and shown in Alg.4.3. The number of mismatches, their symbols and positions are streamed back to the host CPU. Note that additional logic is not required to manage in excess of two mismatches, as any reads with more than two mismatches will have been filtered by the seed stage (i.e. if all 3 seeds do not align to the reference).

```
for (int i = 0; i < MAX_READ_LENGTH; ++i) {
  posMismOne = i < readLen & noMism === 0 & pattern[i] !=== text[i] ?
        constant.var(dfeUint(8),i) : posMismOne;
  symMismOne = i < readLen & noMism === 0 & pattern[i] !=== text[i] ?
        text[i].cast(dfeUint(8)) : symMismOne;
  posMismTwo = i < readLen & noMism === 1 & pattern[i] !=== text[i] ?
        constant.var(dfeUint(8),i) : posMismOne;
  symMismTwo = i < readLen & noMism === 1 & pattern[i] !=== text[i] ?
        text[i].cast(dfeUint(8)) : symMismTwo;
  noMismatch = i < readLen & pattern[i] !=== text[i] ? noMismatch + 1 : noMismatch;
}
```

Code Extracts 4.3: Extract from CompareKernel.maxj - read (pattern) and reference (text) comparison

4.3.5 Inexact alignment module

Any read with seeds aligning to a number of positions exceeding the specified threshold will be processed by the one and two mismatch modules. These are similar in design to Module 1 of exact match, yet incorporate bi-directional backtracking allowing inexact alignment using the FM-index. The inexact match hardware design is illustrated in F.6. Both the one and two mismatch modules contain additional circular buffers that allow the storage of mismatch state, where each mismatch record is composed of a symbol, position, low and $high$ value. These states are used to control backtracking.

The number of alignment positions is unpredictable, so the results of backtracking must be stored in off-chip DRAM before being streamed to the host CPU once processing has finished. This requires creation of a third stream to DRAM, for writing these results, which further divides the bandwidth. Given that there may be a range of alignment hits, the output of the mismatch modules is a set of suffix array intervals which can all be converted into reference genome positions at the host.

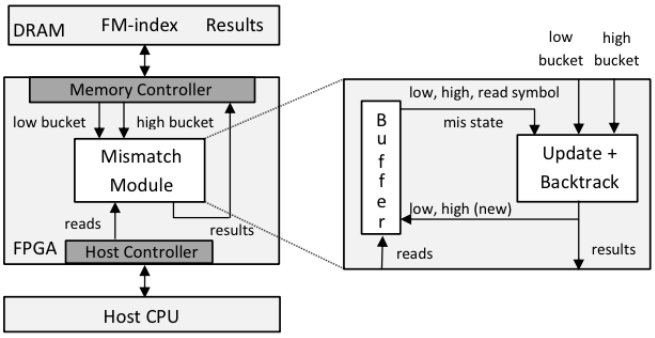


Figure F.6: Inexact match module hardware designs[9].

4.4 Performance Evaluation

In this section we evaluate the performance of the bisulfite sequence alignment design, providing comparisons against the fastest CPU, GPU and FPGA-based alignment solutions currently available. We compare performance using simulated sequencing data, namely bisulfite sequencing data and data replicating raw reads from a consensus genome. Beyond the runtime and throughput of the alignment design, FPGA resource usage and system energy consumption is also evaluated. First, the experimental platform is specified, before describing the chosen sequencing data sets and finally presenting the alignment performance.

4.4.1 Platform specification

The bisulfite sequence alignment design is run on the MPC-X2000 using 8 DFEs, each equipped with a Altera Stratix V FPGA and 48GB of DRAM. The FM-index used is constructed with a step size of $n = 3$, bucket size $d = 256$ and oversampling factor $f = 2$. This produces a 8.7GB index for a C-depleted version of the human genome[45]. MPC-X2000 is limited in that only a single memory controller is available per FPGA, limiting our design to a single module populating each of the 8 FPGAs. The only exception to this is the compare module, which does not require data streams to DRAM, and can consequently have 8 modules populating each FPGA. In the cases where the memory controller is required, the random access pattern of the FM-index search algorithm reduces performance for the MPC-X2000, as the controller is optimised for coalesced memory accesses. Theoretical peak memory bandwidth per FPGA is 38GB/s, however measurements indicate that only 4.2GB/s is achieved.

Given the architecture is runtime reconfigurable as opposed to static, there is an FPGA reconfiguration time. This is roughly 3s per FPGA, and ignored as the sequencing data sets used for testing are small compared to a typical workload to avoid introducing a negative bias. The FM-index transfer time of roughly 2.5s and disk IO is also ommitted from run-time measurements. In a typical workload of 300M short reads, performance suggests these overheads would amount to $< 5\%$ of total run-time. We do however include the time taken by the host CPU to perform operations such as convert suffix array intervals into reference genome positions. This does not include operations such as creating the FM-index and pre-computing suffix array intervals; although these are slow operations, the concencus human genome undergoes irregular changes and consequently these resources can be re-used considerably.

4.4.1.1 Competitor test platforms

The bisulfite sequence alignment design we have presented will be compared to the following reputable CPU-based tools: SOAP2 v2.21[41], Bowtie2[73][1] and BWAaln+samse[74]. In our alignment design, the host CPU gets returned a suffix array interval of all possible hits, allowing uniqueness of alignment to be identified. In all testing however, the other platforms will be run such that they report the random best hit out of the best possible alignments (that is, if there is more than one candidate). The maximum of two-mismatches property of our alignment is not enforced by these other aligners, and has been specified for SOAP2. To alleviate any performance loss that results, BWA has been run with the -very-fast-flag set, and all the platforms were run on 16 threads. The tools are run using dual Intel Xeon X5650s with 120GB of DDR3-1333GHz RAM. The performance is also compared to GPU-based SOAP3-dp[5], also aligning with a maximum of 2 mismatches and finding the best random hit. It is run on a NVIDIA Tesla C2070, with 448 cores, 6GB of GDDR5 and a memory bandwidth of 144GB/sec. It runs on the Kepler architecture and has compute capability 2.0.

4.4.2 Sequencing data sets

Three different types of sequencing data are used to test the performance of our bisulfite sequencing design against the competitor platforms:

1. **Simulated bisulfite sequencing reads:** As a bisulfite sequencing design, primarily bisulfite sequencing data was chosen to test performance. Realistic bisulfite sequencing reads can be simulated using Sherman, as bisulfite conversion rates can be adjusted at C-G and non C-G (C-H) sites. We used the following command to generate 10M reads of length 75bp from hg19:

   ./ Sherman -q 40 -I 75 -CG 20 -CH 98 -e 0

   Bisulfite sequencing however is a rather unique case of sequence alignment, as the alphabet changes from $\{A, C, G, T\}$ to $\{A, G, T\}$ and there is a loss of complementarity between DNA strands. Although the CPU and GPU platforms can be tested with the simulated bisulfute sequencing reads, it is not possible to run the competitor FPGA platforms, let alone with the sequencing data of our choosing given the inflexibility of hardware designs. Therefore, we must use the previously defined bases aligned per second (bps) metric, whilst aligning a more generalised sequencing data set to ensure fair comparison.

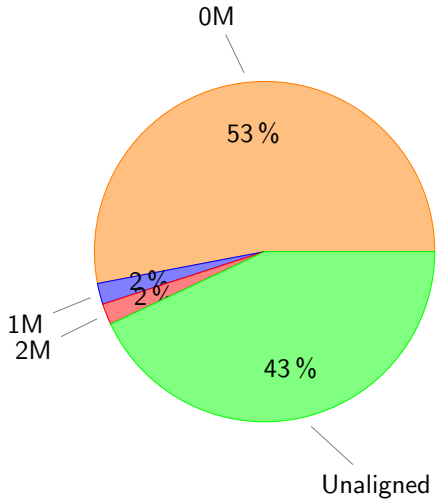
2. **YH genome sequencing reads:** In order to arrive at a bps metric allowing comparison to other FPGA alignment designs, we will use 10M reads of 100bp attained by whole genome sequencing over the YH genome[75]. Clearly our alignment platform is specialised to 75bp reads, so we analysed SOAP2's alignment output for the YH genome reads to simulate a 75bp data set demonstrating the same alignment percentages. The differing alignment percentages between the bisulfite reads and YH genome reads are demonstrated in F.7. Note that any reads with above 2 mismatches were deemed unaligned, due to the constraint on our alignment design.

3. **0,1 and 2 mismatch reads:** In order to attribute the demonstrated results to specific modules of our alignment design, we have also generated three datasets of 10M reads of 75bp by directly sampling hg19. These three sets have 0, 1 and 2 mismatches respectively inserted into random positions of the reads. This sheds some light into the relative performances

[1] Bowtie2 is the alignment platform used by Bismark, a bisulfite sequencing mapper[36].

of the exact match and inexact match modules. These are tested against only SOAP2 and SOAP3-dp, due to the difficulties in enforcing mismatch constraints on the other platforms.

(a) Bisulphite sequencing reads produced by Sherman.



(b) Raw sequencing reads from the YH genome[75].

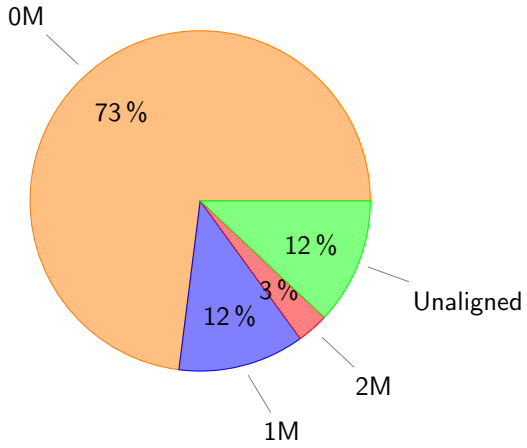


Figure F.7: Alignment percentage comparison between bisulfite and non-bisulfite data sets. Note that any reads with above 2 mismatches were deemed unaligned for the sake of testing.

4.4.3 Runtime and throughput evaluation

4.4.3.1 0, 1 and 2 mismatch reads

The performance of our design is measured for exact match, one mismatch and two mismatch alignment using corresponding simulated data sets with 10M reads taken from hg19. This was compared to the performance of SOAP2 and SOAP3-dp, CPU and GPU aligners respectively, as their alignment parameters are highly configurable allowing the number of mismatches to be specified across the full read being aligned[2]. The performance of our design relative to these two platforms, alongside the performance previously achieved by Ramethy, is shown in F.8.

Our design was significantly faster for exact and inexact matching with up to 2 mismatches, with the largest performance difference being seen for exact match, where the run-time was 80.9 times faster than SOAP2 and 24.8 times faster than SOAP3-dp. This strong performance can in part be attributed to the interval store and oversampling optimisations, which reduced the number of random accesses required in the FM-index search operation. The exact improvement of the exact kernel achieved through optimisation is roughly 9.5%, as shown in T.II.

The performance difference was less substantial over inexact matches. For one mismatch, our design was 17.4 times faster than SOAP2 and 8.3 times faster than SOAP3-dp. For two mismatch, our design was only 3.5 times faster for SOAP3-dp, but 14.6 times faster than SOAP2. The performance gap would have closed significantly without the seed and compare stage, with inexact matching becoming roughly 2.5 times worse. This is because by setting a comparison threshold of 20, almost 80% of the reads were able to rely on the lightweight compare kernel for inexact alignment.

For each set of reads, our design is faster than Ramethy, however in the next section we present a total bps value less than that of Ramethy for bisulfite sequencing reads. This was initially surprising, as Ramethy terminates inexact alignment after a single valid reference position has

[2]Bowtie2 and BWA only allow the number of mismatches in alignment seeds to be specified, i.e. in the first n characters used to determine whether a location in the reference genome is a suitable alignment candidate.

been identified, whereas our design reports all possible positions. This outcome is likely due to the way in which the reads are being simulated; the two mismatch reads created from raw reads likely do not align to a large number of reference positions, in turn resulting in a performance improvement following the seed and compare optimisation.

Figure F.8: Run-time tests for 10M 0, 1 and 2 mismatch reads from the hg19.

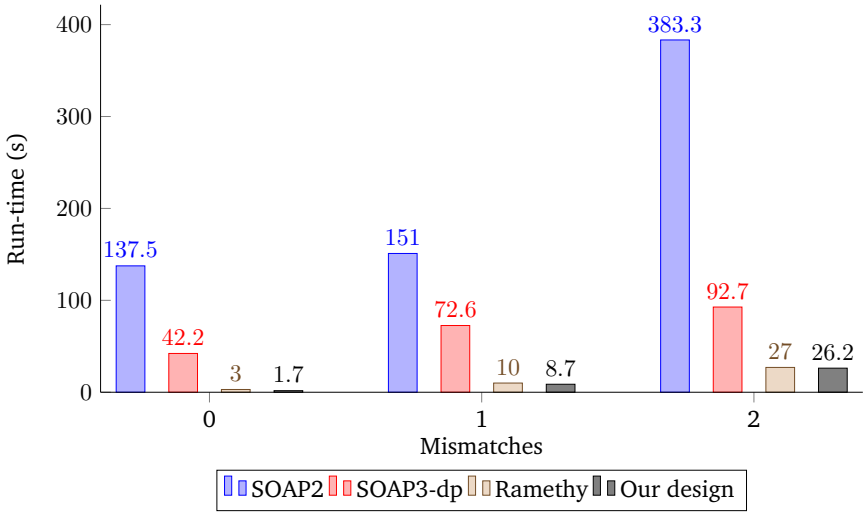


Table T.II: Exact match and seed kernel acceleration with oversampling and the interval store.

Kernel	Exact Match			Seed		
Reads (M)	10	20	30	10	20	30
Speed-up (%)	9.47	10.17	**9.48**	19.23	26.21	**27.65**

4.4.3.2 Bisulfite sequencing reads

The performance of our alignment platform using bisulfite sequencing reads representative of actual data found in BSAligner's typical workload is incredibly promising. Compared to CPU aligners, T.III indicates that our design is at least 15.4 times faster. Our design is also 6.2 times faster than the GPU aligner. Overall, this results in a significantly higher bps metric of 56.82.

In F.7a the alignment percentages for this data set were presented: 53% exactly matching, 2% with a single mismatch, 2% with two mismatches and 43% being deemed unaligned. In terms of the percentage of run-time spent on the corresponding modules: 7.5% on exact match, 12.1% on seed and compare, 9.1% on one mismatch and 71.3% on two mismatch. Two mismatch alignment took almost 75% of the total run-time, for alignment of a small fraction of the total read set. This is clearly the bottleneck of our design, and should be the target of further optimisations. This bottleneck can be eliminated by terminating inexact alignment after the first reference position is found - this was demonstrated by Ramethy, which achieved 66.4bps over a similar set of bisulfite sequencing reads. This however reduces the aligner's utility, reducing accuracy, and making the algorithm stage unsuitable for paired-end alignment (this was motivation behind the optimised alignment design).

Following our definition of a run-time reconfigurable alignment architecture's run-time in 3.1.1, which scales linearly with the read count, it is possible to extrapolate this 10M read workload to that of a typical 300M workload. The 13.2s required to align 10M bisulfite sequencing reads becomes 396s (6.6 minutes) for 300M over a single strand, so little over 10 minutes for 300M reads over both the Watson and Crick strand. This is a tremendous improvement over the 5 hour alignment time taken when running BSAligner on dual 12-core Intel Xeon processors.

Program	Platform	Clock F (MHz)	Devices	Time (s)	bps (M)	Speed-up
SOAP2	2x Intel X650	2660	2	256.1	2.93	1.00×
BWA-aln+samse	2x Intel X650	2660	2	241.4	3.11	1.06×
Bowtie2	2x Intel X650	2660	2	203.0	3.69	1.26×
SOAP3-dp	NVIDIA C2070	1150	1	82.0	9.15	3.12×
Our design	MPC-X2000	150-200	8	13.2	56.82	**19.40**×

Table T.III: Comparison of state of the art CPU, GPU and FPGA alignment using 10M bisulfite reads of 75bp generated by Sherman

.

4.4.3.3 YH genome sequencing reads

The performance of our alignment platform is demonstrated over a data set more representative of a standard sample used to test non-bisulfite sequencing alignment designs. Specifically, we adapted a set of real 10M reads of 100bp sequenced from the YH genome, with alignment percentages as demonstrated in F.7b. This increased the number of reads aligned by the highly optimised exact match module by 20%, it also increased the number of one mismatch reads by 10%, but it barely increased the number of reads processed by the inefficient two mismatch module. The results presented in T.IV demonstrate our design is 10.9 times faster than the design produced by Fernandez which also uses the FM-index search algorithm, and it is 1.19 times faster than the Smith-Waterman approach used by Olson.

It is important to note that although bps is a normalised alignment throughput metric, it is still unfair to compare different FPGA technologies. For example, our design uses Altera Stratix V FPGAs which using 28nm transistor technology. Olson et al. use Virtex-6 FPGAs which have 40nm technology, and they could achieve greater performance if using more recent Stratix V FPGAs. If the transistor technology had a linear relationship to performance, they could improve run-time by a factor of roughly 1.43. Equally, we could use the recent Convey HC-2 platform, allowing more modules to populate each FPGA; currently we struggle with memory restrictions, which will explained in a later evaluation of resource usage. Assuming this could bring about a linear improvement in the performance of the corresponding alignment stages, we can project a speed-up of 2.42. These calculations involved *very naively* scaling the corresponding module's contribution to total run-time by the number of potential module population. The results are shown in T.V.

Clearly, it is difficult to directly compare FPGA-based designs without authors collaborating to use identical hardware platforms and identical data sets, which may not be feasible due to hardware specialisation for different alignment parameters. However, three notable strengths of our design are:

1. A run-time reconfigurable architecture is used, allowing flexible and efficient alignment. The other alignment designs are static designs, with pitfalls detailed in 3.1.1.

2. All alignment hits are found that meet the constraint of up to 2 mismatches, producing accuracy identical to software, where all the hits are often found and filtered as desired during downstream analysis. It is again worth noting that Ramethy does not support this.

3. The architecture stores the reference genome FM-index in DRAM, allowing a large reference to be used, such as the entire human genome. This is made possible through use of optimisations such as the interval store, n-step and the newly introduced novel oversampling of the FM-index; these in part reduce expensive loads from DRAM.

Program	Platform	$\|r\|$	N_r (M)	Clock F (MHz)	Devices	Time (s)	bps (M)
Fernandez[69]	Convey HC-1	101	18	150	4	138	13.2
Olson[70]	Pico M-503	76	54	250	8	34	120.7
Our design	MPC-X2000	75	10	150-200	8	5.2	**144.2**

Table T.IV: Comparison of novel FPGA alignment designs - the read volumes and read lengths detailed in the papers have been used. Note, our design made use of reads derived from the YH genome.

Program	FPGA Technology	Time (s)	bps (M)	Scaled Time (s)	Speed-up	Scaled bps (M)
Olson[70]	Virtex-6	34	120.7	23.78	1.43×	172.6
Our design	Stratix V	5.2	144.2	2.15	2.42×	309.9

Table T.V: Scaled performance comparison of novel FPGA alignment designs - note Fernandez et al. was omitted due to time limitations.

4.4.4 Power and energy usage

The power and energy usage of our design is compared to SOAP2 and SOAP3-dp, CPU and GPU-based tools respectively. The MaxOS was used to measure the device power for each of the 8 FPGAs, however the CPU and GPU power values were conveniently sourced from vendor product information. Our design uses significantly less energy than both SOAP2 and SOAP3-dp, as demonstrated in T.VI. This is not surprising, as our design runs significantly faster than the others, and FPGAs run at a much lower operational clock frequency than CPUs and GPUs. Specifically, our design is run using a maximum of 200MHz clock frequency, whereas SOAP2 and SOAP3-dp run at 2660MHz and 1150MHz respectively. Although expensive to purchase in comparison to CPUs and GPUs, the operational costs (and recurring engineering costs) are low for FPGA solutions. Alongside the small form factor of 1U for systems such as MPC-X2000, FPGAs are certainly practical within a clinical environment.

Program	Device Type	Device Power (W)	Energy Usage (kJ)
SOAP2	CPU	190	48.7
SOAP3-dp	GPU	238	19.5
Our design	FPGA	86 (avg)	1.1

Table T.VI: Comparison of power and energy usage between different hardware platform aligners.

4.4.5 Resource usage

The proportion of the FPGA resources utilised for our alignment designs different alignment stages is demonstrated in T.VII. All modules based on the FM-index search algorithm require at least one DRAM stream, and the achievable population is limited by Maxeler's memory architecture; each FPGA has a single memory controller, resulting in random access commands being processed sequentially. We previously mentioned that only 4.2GB/s memory bandwidth is achieved per FPGA. Given the latency in the order of hundreds of cycles to access an off-chip FM-index bucket from DRAM, using multiple modules would not improve performance. The compare module is unique in that it does not require streams to off-chip DRAM, and as such can be configured with a population of 8 modules. The total achievable population is simply limited by the number of I/O streams to the DFE. Currently only 16 are possible, however without any limitations on these streams a total population of 32 modules could be achievable.

Hardware platforms with multiple memory controllers, and random access speeds that can rival sequential access, could improve the performance of our alignment design significantly. If multiple memory controllers were supported, the FPGA could fit 3 of each module during any one

configuration. An example platform that could be used to achieved greater performance is the Convey HC-2[76], which provides 8 memory controllers and a total of 16 DDR2 memory channels. This provides a a bandwidth of 80GB/s, which is over twice the bandwidth supposedly achievable with the Maxeler MPC-X2000. The HC-2 memory crossbar also uses Convey Scatter-Gather DIMMs, which improves the performance of applications with random memory access patterns[3].

Module	Clock (MHz)	LUT	FF	BRAM
Exact (module 1)	150	69535 (26.6%)	130062 (25.1%)	763 (29.9%)
Exact (module 2)	150	65582 (25.1%)	119255 (23.0%)	609 (23.8%)
Seed	150	71576 (27.3%)	134190 (25.9%)	1220 (47.7%)
Compare	200	51675 (19.7%)	116023 (22.4%)	948 (37.1%)
One mismatch	150	74084 (28.3%)	141293 (27.3%)	835 (32.7%)
Two mismatch	150	74636 (28.5%)	142583 (27.5%)	845 (33.1%)

Table T.VII: Resource usage on an Altera Stratix V FPGA - the percentage use is listed for look-up tables (LUTs), flip-flops (FFs) and BRAMs. Note that a single module is used on an FPGA for each alignment stage, excluding compare where 8 are used due to lack of DRAM streams.

4.5 Summary

In this chapter we have presented a series of optimisations targeting a bisulfite sequencing alignment design built upon the Maxeler MPC-X2000 FPGA platform. The design leverages the reconfigurability of FPGAs, using a run-time reconfigurable architecture to allow highly efficient and flexible alignment. The platform has a 1U form factor and a low operational cost, using less energy than CPU and GPU alternatives without compromising alignment accuracy. The design could also be used to support paired-end read alignment, which was not previously possible using Ramethy.

Evaluations indicates the platform is faster than the competitors tested, but more importantly, the design dramatically reduces the alignment time of Methy-Pipe's BSAligner from 5 hours on dual 12-core Intel Xeon processors to roughly 10 minutes. Consequently, downstream analysis modules within BSAnalyze will become Methy-Pipe's bottleneck.

The optimisations presented extend a novel algorithm based on the n-step FM-index search algorithm: oversampling and the interval store reduce the number of costly off-chip DRAM accesses to retrieve FM-index buckets; and the seed and compare stage significantly improves the inexact alignment by 2.5 times. The optimisations have improved the exact kernel and seed kernel performance by approximately 10% and 25% respectively, however even after introduction of the seed and compare stage the two mismatch stage is highly inefficient and will be the target of future work. Future work will also involve applying reconfigurable architectures such as the one presented to other bioinformatics pipelines, as bisulfite sequence alignment although computationally demanding is a very unique type of sequence alignment.

[3] Convey's Scatter-Gather DIMMs are optimised for transfers of 8-byte bursts, allowing near peak bandwidth to be achieved by non-sequential 8-byte accesses.

Chapter 5

Optimisation of Short Read Alignment using GPUs

In this chapter we present a bisulfite sequencing alignment design accelerated using GPUs, and specifically the CUDA architecture. Following the previous section, where we presented an optimised bisulfite sequence alignment design built upon the Maxeler MPC-X2000 FPGA platform, this solution leverages the highly optimised alignment algorithm on a GPU-based platform. The design outperforms state-of-the-art GPU-based sequence aligners SOAP3-dp[5] and BarraCUDA[6], by 2.8 and 6 times respectively. Projections suggest a full implementation could align BSAligner's typical workload in roughly 40 minutes.

The novel aspect of this design is the use of FM-index oversampling to reduce the number of off-chip accesses to CUDA global memory. Without this optimisation, performance is poor due to un-coalesced load patterns in accessing FM-index buckets.

5.1 Heterogeneous Platform

The motivation behind this contribution is to assess the potential contribution of GPU-based alignment to a heterogeneous bisulfite sequencing analysis platform using GPUs and FPGAs. This kind of heterogenous system would be novel, and we believe it could support the wide-spread clinical adoption of Methy-Pipe.

A heterogeneous bisulfite sequencing alignment and analysis pipeline would allow a multitude of separate tasks to be shared, or delegated such that the appropriate hardware acceleration platform is used. In terms of sequence alignment, this could involve delegating (in a fully abstracted manner) stages of the pipeline with an inefficient reconfiguration time away from an FPGA and towards a GPU. Heterogeneity could also prove useful for downstream analysis, as GPUs are heavily optimised for performing linear operations on vectors and matrices efficiently - determining the relative strengths of different hardware acceleration platforms is always highly algorithm dependent. Numerous heterogeneous bioinformatics platforms with multiple kinds of processors exist, such as message passing platforms using reconfigurable co-processors alongside conventional processing with streaming SIMD instructions[77]. However, there does not currently exist an integrated alignment and analysis pipeline using both GPUs and FPGAs alongside conventional processors.

Within the scope of this project, we only asses the feasibility of a GPU-based alignment design, by implementing the exact matching alignment stage; this module aligns the majority of reads in a typical workload, and is built upon the FM-index search algorithm which would be extended for the inexact alignment stages.

5.2 Alignment Algorithm and Optimisations

The previous FPGA-based alignment algorithm was composed of four stages which consecutively filter a set of reads: (1) exact match, (2) seed and compare, (3) one mismatch and (4) two mismatch read alignment. We assess the feasibility of a GPU-based alignment architecture for exact read alignment alone, as this module processes the bulk of an alignment workload, is algorithmically extended by the inexact alignment stage, and has been the primary target of most optimisations. This alignment architecture would be extended to target Methy-Pipe, although we solely implement exact match, so are currently not concerned with BSAligner's alignment constraint of

reads uniquely aligning to the reference genome with no more than 2 mismatches.

The FM-index search algorithm forms the basis for the exact alignment algorithm, and is extended in this design with an n-step and oversampling. These algorithmic changes are not hardware specialised, but minimise accesses to off-chip DRAM given the FM-index is typically too large to be stored using on-chip memory in both FPGAs and GPUs. We use a variant of the interval store optimisation. It previously targeted on-chip BRAM, whereas it will now reside in a specialised partition of the GPU's DRAM with a dedicated cache for each multiprocessor.

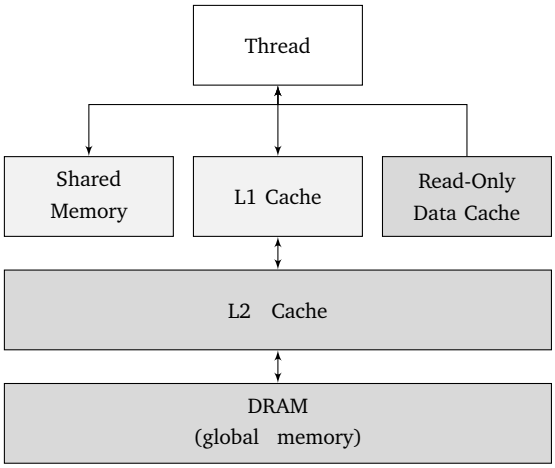
5.3 CUDA Architecture

In this section we present some important aspects of the CUDA architecture, expanding upon the introduction provided in 2.6.2.1. These insights inform the design of our alignment architecture. Specifically, these insights indicate that although CUDA has significant caching capabilities, inefficient global memory usage could greatly harm kernel performance.

5.3.1 Cache hierarchy

In the CUDA architecture, there are three caches: (1) a 64KB configurable shared memory and L1 cache, (2) a 48KB read-only data cache and (3) a L2 cache. This is illustrated in F.1.

Each streaming multiprocessor (SM) has 64KB of on-chip memory for shared memory and L1 cache. This can be configured to prioritise either shared memory or L1 cache, with a 48KB/16KB distribution. In the Fermi architecture, all global loads stores are through this L1 cache, whereas in Kepler this is often not the case. There is also a new 48KB cache which is read-only for the duration of a function (supporting the L1 cache), which takes the load footprint away from the shared memory and L1 cache path. The read-only data cache's higher bandwidth also supports full speed unaligned memory access patterns. Directing use of this cache is simple, with the `const __restrict__` keyword or `__ldg()` intrinsic. Finally, the L2 cache is a data unification point between all SM units, serving load and store requests. Execution with random accesses common to multiple SMs benefit from this cache.



Memory Level	Latency (cycles)
Zero-copy DRAM	1000+
Global DRAM	400-800
L2 Cache	200-400
L1 Cache	10-20
Shared Memory	10-20
Constants	4-64

Figure F.1: CUDA memory architecture and corresponding memory latencies. Note that these are general figures and will vary on each CUDA device.

5.3.2 Global memory

Global memory refers to GPU DRAM, which is accessible to both the host and all threads of the device. It typically has a significant latency of 400-800 cycles, whereas the L2 cache has a latency of 200-400 cycles. Performance of global memory usage is achieved with memory coalescing, aligned data accesses and minimising memory strides. CUDA also supports zero-copy global memory, whereby the GPU threads directly access host memory, with an unsurprising latency in 1000s of cycles over PCIe. This memory is page-locked by pinning or mapping. However, on discrete GPUs this data is not always cached on the GPU, so it should ideally be read and written only once.

In cases where significant global accesses take place, memory transactions may regularly miss the L1 cache. When this occurs, the miss value is returned to L1 cache and the respective warp scheduler is notified to replay the instruction. If the sheer number of load and store instructions are also increasing the load/store unit (LSU) to its maximum resource usage, this will cause instructions to be replayed. Overall, this can cause a high global memory replay overhead and prove to be the kernel performance bottleneck.

5.4 Alignment Architecture

In this section we present an alignment architecture, starting by discussing memory organisation, before explaining the kernel configuration and the workload management. The memory organisation section tackles the primarily challenge of accommodating whole-genome sequencing analysis in GPU memory, as whole-genome index sizes will often exhaust the memory available. The kernel configuration section tackles the challenge of harnessing the GPUs potential for parallelisation, in producing an accelerated alignment workflow.

5.4.1 Memory organisation

5.4.1.1 Constraints

The FM-index alone for the human genome consumes 8.7GB of memory, when constructed with a step size of $n = 3$, bucket size $d = 256$ and oversampling factor $f = 2$. The DRAM available on the majority of GPU devices is 4-6GB, however high performance devices are available with 12-24GB of DRAM, such as the Tesla K40. Devices running the new Maxwell architecture are available with 12GB of DRAM at a very reasonable price and will support this design, such as the GeForce GTX TITAN X.

5.4.1.2 Design

Our design is illustrated in F.2. As this architecture would be extended to specialise for whole-genome bisulfite sequencing, i.e. optimising Methy-Pipe's BSAlign, we design our architecture such that the FM-index is stored in DRAM (global memory). The is necessary for two reasons: (1) all threads need FM-index visibility, and (2) it would be challenging to store an FM-index for the whole human genome in a more granular region of the CUDA memory hierarchy without significantly compromising alignment throughput. The FM-index could be stored in zero-copy host memory as opposed to global memory, but the increased memory latency would render the design infeasible.

The most *crucial* problem our design consequently faces is that random accesses to the FM-index in device global memory are costly. CUSHAW[78], a reputable GPU-based short read aligner, uses a bi-directional BWT approach to alignment which requires only 2.2GB memory for the human genome. However, it also uses global memory, and highlights poor data locality and consequent warp execution divergence as its performance bottleneck. Our unstinting use of memory, due

to the n-step and oversampling optimisation of the FM-index, improves the potential alignment performance beyond that of CUSHAW. Specifically, although index bucket sizes increase, the sheer number of global memory loads are reduced. The use of oversampling to achieve this is novel, and is key in alleviating the problem of random access costs.

Due to the poorly coalesced nature of FM-index accesses and little scope for thread co-operation with the given alignment algorithm, we also make use of the L1 cache. The `const __restrict__` keywords are used to use the read-only data cache for bucket accesses, and we configure the memory for the kernel such that the L1 cache is preferred to shared memory (using `cudaFuncSetCacheConfig(exact_match, cudaFuncCachePreferL1)`).

Zero-copy memory must be used however to bypass the memory limitations of GPUs with less than 24GB of DRAM; the suffix array for a human genome can use in excess of 10GB, and the reads themselves may use as much as 69GB for a typical workload. This is achieved by creating device pointers to these resources for the kernel (`cudaHostGetDevicePointer()`), after loading them into host memory and creating a mapped, pinned host buffer (`cudaHostRegister()`). The reads could be copied in batches to the kernel, however we have chosen to store them in zero-copy memory for two reasons: (1) a thread addressing scheme can be used that results in coalesced accesses, as shown in Alg.5.1; and (2) only a single load to get read symbols is required, and a few writes to update read position and hit count is required following alignment, which should be less costly in cycles with coalesced accesses than the time taken to copy reads to and from the device.

```
static __global__ void
exact_match(const uint32_t BATCH, const uint32_t VOL, read_t *reads, ...)
{
  int i = (threadIdx.x + blockIdx.x * blockDim.x) + ((BATCH - 1) * N_READS_BATCH);
  if (i >= (N_READS_BATCH * BATCH) || i >= VOL)
    return;
  read_t read = reads[i];
    ...
}
```

Code Extracts 5.1: Extract from `exact.cu` - thread addressing scheme used within exact match kernel, where BATCH identifies the stream of the given kernel call and VOL identifies the number of reads being processed across all streams.

Although nowhere near as important as the oversampling optimisation, the suffix array interval store optimisation can be implemented using constant memory, given there is a total of 64 KB constant memory on a device. Clearly this store must be significantly smaller than that using the 8MB of BRAM available in the FPGA-based design. The constant memory space in CUDA has a dedicated 8KB cache on each SM, meaning the store values are cached separately to FM-index buckets, and will not consequently use space that would other be used to reduce FM-index DRAM accesses.

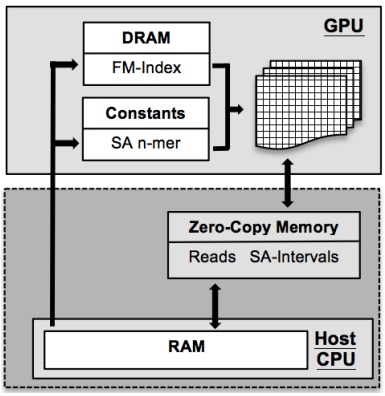


Figure F.2: Memory organisation for CUDA alignment architecture - SA standards for suffix array, and n-mer refers to the pre-computed suffix interval values

5.4.2 Kernel configuration

Beyond memory management, the way in which the GPUs parallel potential is harnessed is important in achieving high sequence alignment performance. Following our discussion of memory management, we can propose a simple workflow for our kernel which lets us achieve reasonable performance despite the memory limitations.

The exact match kernel is written such that each thread takes responsibility for the alignment of a read. Due to the dependence of FM-index search steps on previous steps, this is an intuitive way of managing the workload as it avoids the complexity of sharing work within blocks of threads. We launch the kernel in batches from the CPU, such that each batch processes a total of 4000 reads.

The kernel is specifically configured in a 64×1 grid of blocks, which each contain a 64×1 grid of threads, as this produces the highest performance. CUSHAW also uses 64 threads per block, however uses multiple passes over batches of short reads to alleviate memory pressures. We perform exact alignment of reads in a single pass, with the parallelism achievable at any one time limited by the competing accesses to the FM-index in global memory; arrays allocated in DRAM are aligned to 256-byte memory segments by the CUDA driver, and the device is limited in how many 128-byte transactions it can make to global memory at any one time (each FM-index record is typically over 128-bytes in size).

The fact we are using a single pass is not novel, however, it is worth noting that designs such as CUSHAW using multiple passes may create a lot of memory transfer overhead. This would be the case if many non-overlapped calls to `cudaMemcpy()` are made between each pass, transferring data inefficiently between host and device. Our design does not encounter this problem, as we have stored reads in zero-copy memory to avoid using any storage space in DRAM - we devote this space solely to a highly optimised FM-index.

We also launch the kernel over batches of reads using separate kernel streams. In CUDA, streams are sequences of operations that execute on the device in the order issued by the host, yet operations within streams can be interleaved and run concurrently with those of other streams. This is organised by the CUDA work distributor, however concurrent behaviour is not guaranteed. We create separate streams for each batch of reads, each being non-blocking with respect to the host CPU. After submitting the batches to the GPU, the host blocks until the streams have all terminated and synchronised, using `cudaDeviceSynchronize()`. This is shown in Alg.5.2

```
uint32_t batches = !(N_READS % N_READS_BATCH) ? N_READS/ N_READS_BATCH :
          ((N_READS_BATCH - (N_READS % N_READS_BATCH)) + N_READS)/N_READS_BATCH;

for (int batch = 1; batch <= batches; ++batch)
{
  // Launch the kernel configuration of blocks B and threads T (64x64)
  exact_match<<<B, T, batch>>>(batch, ...);
}
cudaDeviceSynchronize();
```

Code Extracts 5.2: Extract from `exact.cu` - launching streams of read batches to the kernel.

5.5 Performance Evaluation

In this section we evaluate the performance of the sequence alignment design, providing comparisons against some of the fastest GPU-based alignment solutions currently available. These are SOAP3-dp[5] and BarraCUDA[6]. Unfortunately we were unable to test CUSHAW as incompatible C++ compilers were available for the target CUDA version. Similar to the previous section, we test the performance of the following CPU aligners using 16 threads: SOAP2, Bowtie2[1] and BWAaln+samse[2]. The performance of our FPGA-based design is also presented. We compare performance using simulated read data extracted from chromosome 22 of a concensus genome. We also project performance over the human genome and a typical bisulfite sequence alignment workload. First, the experimental platform is specified, before describing the chosen sequencing data sets and finally presenting the alignment performance.

5.5.1 Platform specification

We use the NVIDIA TITAN Black to test the performance of our alignment design against SOAP3-dp and BarraCUDA. The TITAN Black has Compute Capability 3.5 and can run the Kepler GK110 architecture. It has 2880 cores with a base clock frequency of 889Hz, and a memory bandwidth of 336GB/s. This card only has 6.14GB of GDDR5 DRAM, so does not have enough memory to store the entire human genome index following oversampling. Due to inaccessibility of more recent GPUs with greater DRAM, we will only be able to project results for the human genome. The host CPU used is an 8-core Intel E5-1620 v2 3.70GHz, with 16GB RAM. This CPU will be used to test the performance of the CPU-based aligners, and the experimental platform specified in the previous chapter is used to test the performance of our FPGA-based design.

5.5.2 Sequencing data set

As we are limited in the hardware available for testing the performance of our GPU design, we cannot use whole-genome sequencing data without removing the oversampling optimisation - constructing the FM-index with a step size of $n = 3$ and bucket size $d = 256$ uses only 5.4GB. We use two different types of sequencing data to test the performance of our bisulfite sequencing design against the competitor platforms:

1. **0 mismatch reads from chr22:** In order to demonstrate the performance of our design with the intended optimisations, we have generated a dataset of 10M reads of 75bp by directly sampling chromosome 22 (chr22) from a consensus human genome (hg19). In this case,

[1] Bowtie2 was run with the `--very-fast` flag and scoring options `--score-min 'C,0,-1'` to ensure exact matches.
[2] BWAaln+samse was run with the options `-n 0 -o 0 -k 0` to ensure exact matches.

we use an FM-index with a step size of $n = 3$, bucket size $d = 128$ and oversampling factor $f = 2$. The performance is tested against CPU and GPU-based alignment solutions, alongside our proposed FPGA-design.

2. **0 mismatch reads from hg19:** Although we do not intend this design to target GPUs which cannot store the FM-index in memory, we will test the performance of our design against 10M reads of 75bp taken by directly sampling hg19. This involves using an FM-index with a step size of $n = 3$ and bucket size $d = 256$, and no oversampling, allowing us to demonstrate the importance of the oversampling optimisation.

5.5.3 Runtime and throughput evaluation

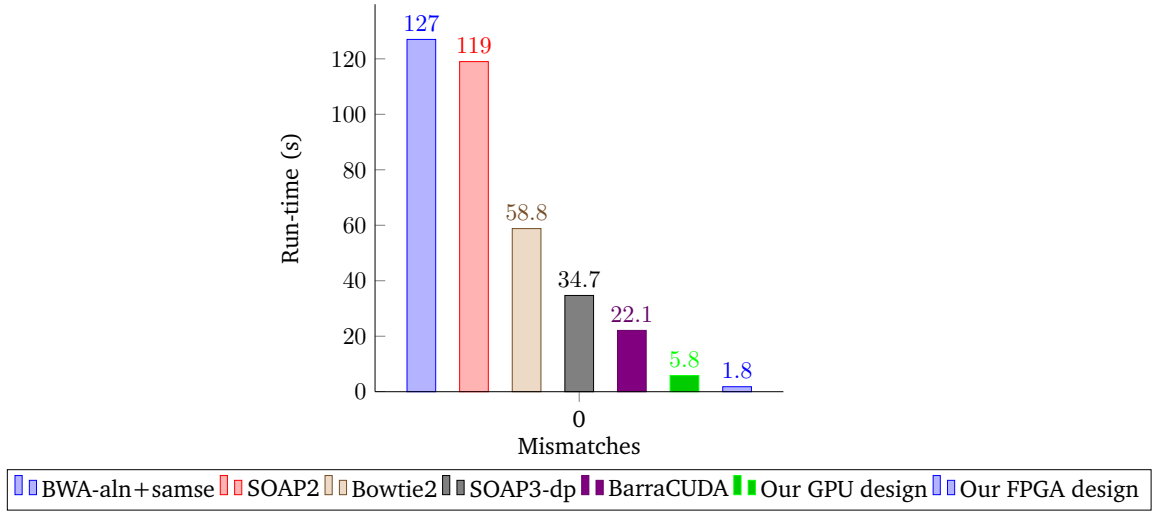
5.5.3.1 0 mismatch reads from chromosome 22

The performance of our design is measured for exact match over 10M reads taken from chr22. This was compared to the CPU aligners running with 16 threads, SOAP3-dp and BarraCUDA (GPU) and our FPGA-based design. The results are presented in F.3. We have not tested CUSHAW, a reputable GPU aligner, due to incompatibilities with our available test platform. Similar to the approach taken in the previous chapter, we have ignored the time taken to configure hardware and load the index, instead focusing solely on alignment time.

Our design was significantly faster than the GPU based platforms, running 3.8 times faster than BarraCUDA and almost 6 times faster than SOAP3-dp. Unsurprisingly, our design exceeded the run-time performance of all CPU aligners. It ran 21.9 times faster than the slowest, BWA-aln+samse, and 10.1 times faster than the fastest, Bowtie2. On a single GPU, our design is only 3.2 times slower than our FPGA-based design which uses 8 FPGAs. For each additional GPU introduced, the read workload could be distributed evenly, such that if N_g is the number of GPUs and T_0 is the single-card run-time, the new time T_1 would simply be: $T_1 \simeq T_0/G$. For example, if $G = 3$, the performance of our design over this set of sequencing reads would be roughly 2s, which is very similar to that of our FPGA-based design.

The simplicity of using multiple GPUs allows us to perform this simple calculation to distribute workload. All GPUs and CPUs on a network will have a unified virtual address space, which can be used to access the zero-copy memory (including reads and suffix array). With respect to the index however, which is stored in DRAM, the host CPU can simply load the index into each GPU by programmatically switching device. This would be done using `cudaSetDevice()`, before calling another `cudaMalloc()` and `cudaMemcpy()`. Naturally, this would result in a slightly larger start-up overhead. The multi-GPU approach would not require any complicated peer-to-peer memory copies, and the host code required to manage the workload between different GPUs would not be complicated. The cost incurred would instead be the significant power usage, which is high on a single GPU alone - this is discussed later in the chapter.

Figure F.3: Run-time tests for 10M 0 mismatch reads of 75bp from chr22.



5.5.3.2 0 mismatch reads from the human genome

The performance of our design is measured for exact match over 10M reads taken from hg19. The FM-index generated does not use oversampling due to the fact our target GPU has only 6GB of DRAM. The FM-index has a step size of $n = 3$ and bucket size $d = 256$, and is 5.4GB in size. The results of this test demonstrated the importance of the oversampling optimisation in minimising costly global memory accesses to retrieve FM-index buckets, as alignment time was 38.5s whereas with oversampling we observed a run-time of 5.8s. This corresponds to slowing in runtime by 6.64 times, i.e. it took 664% longer.

In F.4 we modified our exact alignment CPU simulation to report the number of FM-index search iterations taken each read before single-bucket accesses can take place, i.e. when oversampling would usually kick in. For this data set, the average number of FM-index search iterations was only 6, with 88.3% of the reads benefiting from oversampling after less than 8 search iterations. We can calculate the total off-chip accesses to DRAM for the FM-index using the following equation, $\frac{1}{n} \cdot (2\mathcal{L} + (L - \mathcal{L}))$ (ignoring the interval store optimisation), where L is sequence length, n is the step and $\mathcal{L}$ is average iterations until an oversampled access. Using this equation, the average number of off-chip accesses to DRAM is reduced from 50 to 27 with this set of sequencing data. For this design, that corresponds to a 46% decrease in global memory accesses which each require 400-800 cycles. Therefore, the alignment time increasing by 664% for alignment of 10M reads of 75bp without an oversampled FM-index is by no means surprising.

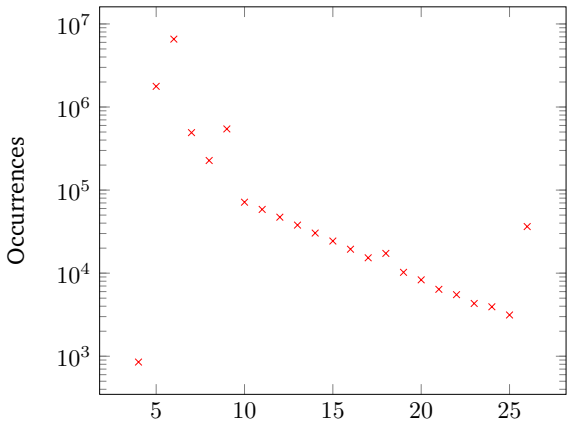


Figure F.4: Test of average number of search iterations taken before single-bucket access with oversampling optimisation - using 10M reads of 75bp and 0 mismatches from hg19. Note that a logarithmic scale has been used on the Y axis.

5.5.3.3 Projected performance

We can make projections of the performance our design would achieve if inexact alignment was incorporated, and the GPU chosen had sufficient DRAM to store an oversampled FM-index for the human genome. Assuming the extended design implemented a seed and compare stage and inexact alignment stage identical, algorithmically speaking, to that of the FPGA-based design, we could also assume the speed-up achieved by the FPGA-based design for exact match is similar for the following pipeline stages. This is not unrealistic, as all the optimisations carried through from the FPGA-design are not hardware specialised, but support hardware acceleration with any device where the FM-index can be stored in DRAM.

Our FPGA-based design achieves a 3.22 times speed-up over this design for exact match of 10M reads of 75bp from chr22, and achieves a total runtime of 13.2s for the alignment of 10M bisulfite sequencing reads of 75bp generated by Sherman from hg19. This indicates this design could align the same 10M bisulfite sequencing reads in 42.5s. This would be 6.02 times faster than SOAP2, and 1.93 times faster than SOAP3-dp. This would mean achieving 17.65Mbps, as shown in T.I. If this is scaled to project the alignment time for a typical BSAligner workload of 300M bisulfite sequencing reads of 75bp, a full GPU-based alignment design would take a little over 21 minutes.

Program	Platform	Clock F (MHz)	Devices	Time (s)	bps (M)	Speed-up
SOAP2	2x Intel X650	2660	2	256.1	2.93	1.00×
BWA-aln+samse	2x Intel X650	2660	2	241.4	3.11	1.06×
Bowtie2	2x Intel X650	2660	2	203.0	3.69	1.26×
SOAP3-dp	NVIDIA C2070	1150	1	82.0	9.15	3.12×
Our GPU-design*	NVIDIA TITAN Black	889	1	42.5*	17.65*	**6.02**×*
Our FPGA-design	MPC-X2000	150-200	8	13.2	56.82	**19.40**×

Table T.I: Comparison of state of the art CPU, GPU and FPGA alignment using 10M bisulfite reads of 75bp generated by Sherman. This includes the projected performance of our GPU-design (indicated with an asterisk, *).

5.5.3.4 Power usage

We have so far ignored the power usage of our design, which compared to our previous FPGA-based design, is significant. In the last chapter we used MaxOS to measure the device power for each of the 8 FPGAs being used, and noted an average power usage 86W over several trials. This

can in part be attributed to the low operational clock frequency of FPGAs. It was however clear that the power usage of CPU and GPU-based aligners, such as SOAP2 and SOAP3-dp were much greater. Our GPU-based design is no exception, as when run on the NVIDIA GTX TITAN Black, with a base clock frequency of 889MHz, stock power usage is 250W (see T.II). Unfortunately we could not manually measure power usage due to host drivers failing to recognize the CUDA device properly. It is clear that with a single device, the power usage is 2.9 times greater than that of our FPGA-based design, which could become even more costly following introduction of more GPUs to rival the FPGA-based performance.

Program	Device Type	Device Power (W)	Alignment Speed-up
SOAP2	CPU	190	1.0×
SOAP3-dp	GPU	238	3.12×
Our design	GPU	250	6.02×
Our design	FPGA	86 (avg)	19.40×

Table T.II: Comparison of power usage between different hardware platform aligners, including our GPU-based design. This is shown alongside the speed-up over 10M bisulfite sequencing reads, to highlight the performance of each solution.

5.6 Profiling

The NVIDIA CUDA Profiler (`nvprof`) provides a greater insight into the memory utilisation and compute resources utilisation of our design. The profiling results shown in T.III were taken when aligning the 10M and 0 mismatch reads of 75bp taken from chr22. It should be evident that our design, although producing competitive alignment speeds for exact match, is highly inefficient.

The glaringly obvious observation is that the theoretical resource occupancy of our approach is only 50%, with only 31 registers being allocated per thread for our small threadblock dimensions of $64 \times 1 \times 1$ threads and $64 \times 1 \times 1$ blocks. Optimising a kernel typically involves improving concurrent access patterns, saturating the bus width by increasing occupancy, however we found that larger threadblocks increase LSU utilisation to the point that lower performance was achieved. Our alternative means of improving concurrent access patterns would be to process several reads per thread, changing the way in which we manage batches.

Due to the random-access usage of the FM-index, the global memory address pattern is not coalesced, i.e. adjacent threads are unlikely to perform adjacent memory transactions. The address pattern can sometimes improve by using non-caching loads (which can be enabled by a compiler directive), whereby smaller transactions are made from the SM to L1 (32B instead of 128B). This however is not feasible for our design, given the size of our FM-index accesses (i.e. bucket size) are in excess of 128B; alignment performance is roughly 2.19 times slower, likely due to the increased number of memory transactions that non-caching loads creates.

The actual effect of high transaction volume and non-coalesced addressing of global memory for the FM-index can be quantified numerically, as it largely contributes to the global memory replay total for the kernel. The global memory replay is 45.1%, which is a significant proportion of replayed instructions - this is calculated using the equation below. Naturally, our design's poor access pattern also results in high global memory divergence; threads within each warp will experience an uncorrelated cache hit or miss outcome, in turn causing differing memory latencies and replays. The actual hit/miss ratio for L1 global cache loads is reasonable though, at roughly 16:1 (note that these transactions could be in chunks as low as 32B).

Definition - Global Memory Replay (%)

$$G.M.Replay\ (\%) = 100 \times \frac{l1\ global\ load\ miss}{instructions\ issued} \tag{5.1}$$

Event	Detail
Kernel run-time	2.356ns
Registers allocated per thread	31
Peak theoretical occupancy	50%
Single instructions issued per cycle	1.686×10^7
Global load memory divergence replays	1.218×10^7
Global load requests	5.453×10^5
L1 global load transactions	1.311×10^7
L1 global load hits	1.196×10^7
L1 global load misses	7.603×10^5

Table T.III: NVIDIA Profiler details for an average kernel launch (batch) in our GPU-based alignment design - run with 10M and 0 mismatch reads of 75bp from chr22. For some of the notable CUDA events, we provide a description and the corresponding metric.

5.7 Summary

In this chapter we have presented a GPU-based bisulfite sequencing alignment design, using the CUDA architecture. Specifically, we have produced the exact alignment stage, however this could be extended. The motivation behind this was to test the feasibility of a GPU-based aligner that could contribute to a heterogeneous alignment platform, supporting our FPGA-based design. For example, when having to configure FPGAs sequentially (which is a current limitation faced with the Maxeler MPC-X2000 platform), the overhead may render the GPU-based design faster for smaller alignment workloads.

The design is faster than the GPU competitors tested, as it was 3.8× faster than BarraCUDA and almost 6× faster than SOAP3-dp. Projections based on this test suggest the GPU-based design could reduce the alignment time of Methy-Pipe's BSAligner from 5 hours on dual 12-core Intel Xeon processors to roughly 42 minutes on a single GPU with 12GB DRAM (for bisulfite sequence alignment of the Watson and Crick strand). This is around 28.8 minutes slower than the performance expected of our FPGA-based design, which targets 8 FPGAs running on the Maxeler MPC-X2000 platform. Using three GPUs should result in a near identical alignment time for a typical workload, suggesting the GPU-design could at least share the alignment workload with the FPGA-based design in a heterogeneous system, alongside accelerating tasks further downstream that it may be better suited to. However, the power usage of this design is much greater than that of the FPGA-based design, which could prove costly in the long run.

Profiling the design supported the test using a non-oversampled FM-index in exposing the limitation of this design: performance is memory bound, as the FM-index is so large that it must be stored in global memory, and non-coalesced random-accesses result in a high global memory replay and costly memory accesses to DRAM. Although oversampling greatly reduced the effect of this inefficiency, there is still significant pressure on the LSU that results in a poor computational resource occupancy. Further work to this design should involve identifying spatial access patterns to the FM-index, such that the workload of reads can be spread allowing warps to access coalesced buckets. This in turn could leverage more architecture-specific CUDA features, such as Kepler and Maxwell's dynamic parallelism, which allows work distribution to be performed by the GPU.

Chapter 6

Accelerating Compression of Sequencing Data

In this chapter we present a novel approach to lossless referential compression, adapting the algorithms used throughout this report for alignment. Specifically, we use the exact match design to identify matches between the to-be compressed sequence, and a given reference. This work was inspired by M. Pflanzer's comments in [51] on our observations that in our FPGA-based design for short read alignment, potentially 70-80% of reads could be aligned using exact match alone. This suggested a simple alignment design, lacking in any inexact alignment support, could be suitable for accelerated referential compression.

The poor compression throughput of FRESCO and GDC 2 (presented in 2.5.1) could prove problematic for clinical settings where Methy-Pipe and related tools are used for aligning or analysing sequencing data, due to the increasing importance of these tools in standard clinical routines and consequently volume of patient sequencing data.

Therefore, we use the proposed compression algorithm to produce novel FPGA-based and GPU-based compression designs, whereby the GPU-based design is presented in this chapter. In the case of compressing 10M bisulfite sequencing reads from a human genome, our FPGA-based design runs 401 times faster than GDC 2 at 3.95 seconds. For this data set, our design achieves a compression ratio of 2.20, however GDC 2 with its superior compression algorithm achieved 5.94.

6.1 Compression Mapping

Given an arbitrary sequence, we represent a mapping to the reference sequence using a tuple $\langle pos, len \rangle$ where pos is the offset of the sequence within the reference text and len is the sequence length. For example, if the query sequence was a single sequencing read of 75bp that exact aligned to the reference sequence, a single tuple would be required to store the read. If a sequence does not exactly align to the reference sequence, it can be split into shorter sequences which may match. For a given sequence, it can have a set of tuples T, where the length of the sequence is equal to $\sum_{i=0}^{|T|} snd\ \langle pos_i, len_i \rangle$. An example is shown in F.1. In the most extreme case, a tuple could naively encode a single base, ensuring lossless compression, i.e. $\langle p, 1 \rangle$ where p could be any position where the respective base occurs.

Reference : ACGTGCGAGTCCAAGTGCGTAC...

Query_1 : ACGTGC	~~CAAGTA~~			$\langle 0, 12 \rangle$ is unaligned.
Query_2 : ACGTGC	CAA	~~GTA~~		$\langle 0, 6 \rangle$ is aligned as is, but $\langle 7, 6 \rangle$ is unaligned.
Query_3 : ACGTGC	CAA		GTA	$\langle 7, 3 \rangle$ and $\langle 10, 3 \rangle$ are aligned to $\langle 11, 3 \rangle$ and $\langle 18, 3 \rangle$.

Figure F.1: Example of mapping tuples for sequence compression.

6.2 Compression Algorithm

Our compression algorithm is simple, and involves generating, exact matching and splitting tuples successively until they are all aligned to the reference sequence. In this section we step through the algorithm at a high level, as shown in Alg.5. The pseudo-code for the auxiliary functions is shown in Appendix A. We have not included the exact match pseudo-code as it only requires a few modifications to the previous fully optimised implementation: (1) input records contain sequence

symbols and a tuple $\langle pos, len \rangle$ and (2) n-step FM-index search iterations are limited by len as opposed to the sequence length. Although these modifications are made, the exact match alignment is not concerned with the origins of each tuple - they are considered individual sequences throughout the algorithm. This maintains a high level of abstraction between the alignment module and tuple management logic. The alignment stages are as follows :

1. Given a set of sequences, in FASTA or FASTQ format, we load them from file into memory and create records which contain the sequence symbols alongside the tuples that will be used to store the compressed sequence. Large sequences in FASTA format are read in chunks of 150 symbols (this is relatively arbitrary and could be altered). The algorithm is written in C++, so we store the sequences in arrays of `structs`, where each `struct` contains a `std::vector` for the tuples. The first tuple will have a length equal to the length of the sequence read from file.

2. A batch of records that can be processed by the exact match alignment stage is created using the first tuples[1]. Exact alignment then takes place over this batch of records.

3. The tuples are parsed and their $(low, high)$ interval values checked to see if they were aligned successfully. There are now two cases for each tuple,

   (a) If aligned, then a full sequence match has taken place and the corresponding reference position can be calculated from the interval values.

   (b) If unaligned, we repeat the prior stages for the reverse complement of the sequence. This accounts for any strand differences between the sequence and reference; unlike our previous bisulfite sequencing alignment procedures, where reverse complementarity is lost during bisulfite treatment, allowing us to accept more generic sequencing data.

4. Remaining unaligned tuples following the full-sequence reverse match are split in half, such that each tuple refers to a half of the sequence. This means each sequence which has not been exactly matched, in full, will have two tuples in its `vector`. These tuples are then added to a new batch of records for processing by the exact match algorithm.

5. Using a non-terminating while loop, the tuples get exact matched in batches and split until parsing of an output batch indicates all remaining tuples have been aligned (i.e. $low \leq high$ for all tuples).

6. Finally, the tuples can be written to file.

Algorithm 5 Referential compression with exact match alignment

Input: Set of sequencing $reads$, reference FM-index F, suffix array intervals SAI and reference suffix array SA
Output: Sequence-to-reference mapping tuples, T

```
step ← 1                                          ▷ Initially, single tuple per read
T ← generate_input(reads, step)                   ▷ Add each unaligned tuple in all read mappings to an array
exact_align(T, F, SAI)                            ▷ FM-index search algorithm, ideally using n-step and oversampling
parse_output(reads, step, T)                      ▷ Check low/high values of tuples, then update reference position or split tuple
reverse_complement(reads)                         ▷ Get DNA reverse complement of read sequences
while unaligned tuple in T do                     ▷ Keep generating tuples and aligning them until termination
    T ← generate_input(reads, step)
    exact_align(T, F, SAI)
    parse_output(reads, step, T)
    step ← step × 2                               ▷ Increase step, causing each unaligned tuple to be split in half on input generation
end while
```

[1] This is simplified in 5, it is worth noting that these records do not contain the tuples alone - they also contain the id of the read they come from and the read symbols.

6.3 Match length and tuple splitting

Achieving a competitive compression ratio relies on few tuples being required to store each sequence, and consequently few tuple splits taking place. Trivially, if tuples are regularly being used to store sequences of only a few symbols, the compression will be inefficient. We perform a few simple estimations to justify the worst case performance of this algorithm.

6.3.1 Statistical estimation

Without taking into consideration the unique properties of genomic sequencing data, we can treat the reference sequence as a random string of uniformly distributed symbols in the alphabet {A, C, G, T} and show statistically that generally, few sequence splits need to take place. We can use the equation below to prove halving the sequence dramatically improves probability of occurrence, where n is query sequence length, m is reference sequence length, Σ is alphabet size and P_o is the probability of occurrence. Concisely written, this equation calculates the probability of not not seeing a sequence of n desired characters in the $m - n$ possible locations, given Σ^n possible subsequences.

Definition - Subsequence occurrence probability estimation

Let P_o be probability of occurence, n be the query length,
m be the text (reference) length and Σ be the alphabet size.

$$P_o = 1 - \left(\frac{\Sigma^n - 1}{\Sigma^n}\right)^{m+1-n} \tag{6.1}$$

For example, taking an alphabet size of $\Sigma = 4$, sequence of length $n = 10$ and reference sequence of length $m = 1000$, P_o is 0.0094% whereas a split to $n = 5$ improves this probability to 62%. The ratio of $n : m$ in this example is $1 : 100$, whereas if compressing reads of MethyPipe's typical workload of 75bp against a genome of roughly $3.2{\times}10^9$ bases, where the ratio is $1 : 42666667$, it can be assumed the effect of splitting is a *lot* greater statistically.

6.3.2 k-mer estimation

We can however generate k-mers and attempt to align them to the human genome, in order to get an idea as to when a given read will be split, i.e. wont exactly align. With the correct genetic distribution in the reference text, and all permutations of query texts aligned, this will indicate the base case by which the average read will always exactly align following splits. For each k, all permutations of reads with length k are generated, for example a 5-mer produces $4^5 = 1024$ reads. We identified that it was only at 11-mer, that 32 of 4194304 reads didn't exactly align (7.63×10^{-4}%). At 12-mer, only 4491 of 16777216 reads didn't exactly align (0.027%). For a sequence of length n, we can be *certain* that in the worst case $s = \lceil \log_2 \lceil n/10 \rceil \rceil$ splits must occur. This corresponds to 2^s tuples of 5B being required to store a sequence.

Take 300M reads with an alignment distribution similar to that of the YH genome we tested in chapter 4. We stated that 70-80% of the reads could exactly align, in the worst case producing 210M tuples immediately which requires 1.05GB storage. For the remaining 90M reads, we can be *certain* that no more than 3 splits occur: 75bp splits to 37/38bp, which splits to 19/19/19/18, which splits to 10/9/10/9/10/9/9/9 reads that will all exactly align. This means a maximum of $2^3 = 8$ tuples would be required to store each remaining read, which would require 3.6GB storage. The raw bit compressed storage of these 300M reads would require 5.63GB storage, indicating that overall a compression ratio of 1.21 in the absolute worst case for inexact alignment. Note that

it is highly unlikely this would be the outcome, given the likelihood of an 11-mer not aligning is less than 1%.

This indicates that although our algorithm is simplistic compared to the two pass compression approaches used by FRESCO and GDC 2, it could achieve a reasonable compression ratio.

6.4 Alignment Architecture

It should be clear to the reader that the foundation of the proposed compression algorithm is the exact match alignment algorithm used in the last two chapters of this report. Therefore we implement this compression design on CPU, GPU and FPGA. It is the high throughput nature of our FPGA and GPU alignment designs that improve the matching process of our referential compression algorithm in a novel way, providing the scope to surpass the achievable compression times of FRESCO and GDC 2.

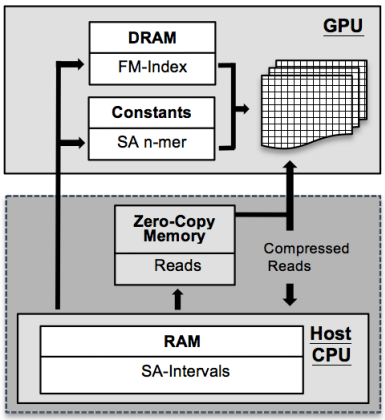
The majority of changes made to both the FPGA and GPU-based designs involve changes to the host code, for the management of tuple batches and parsing of tuple alignment results. For demonstration purposes, we will present the GPU-based design, which modifies that presented in the previous chapter (despite the inefficiency of the design).

6.4.1 Memory organisation

There are two main differences in the memory organisation between this design and that used in the previous section: (1) the suffix array is stored in host RAM, instead of zero-copy memory and (2) compressed reads are additionally copied between host and device between kernel executions. This is illustrated in F.2a.

We have designed the compression algorithm such that the exact match kernel should be oblivious to the management of tuples. If the GPU was managing tuples on the device, instead of simply aligning batches of tuples, it would undergo significantly memory strains; alongside the FM-index, at any one time there may 2^n tuples for each read, where n is the number of tuple splits that have occurred. For the sake of Methy-Pipe, for each tuple we require a `uint32_t` for storage of the position, to allow for an offset throughout a human genome, and only a `uint8_t` for the length as we expect to be compressing reads of only 75bp. Ignoring the other information required for alignment, with only 10M reads and no tuple splits, this would require a minimum 0.4GB of DRAM. For a typical workload with no overlapping of the read loading into host CPU RAM, the 300M reads with no tuple splits would require 12GB of GPU DRAM, which is infeasible. The alternative of storing the tuples in zero-copy memory would also not be feasible, due to the increased memory latency overhead.

Similar to the previous design, where optimum performance was achieved with a configuration allowing batches of reads to be aligned at any one time, we copy tuples from host RAM to the GPU DRAM in batches. The read symbols are stored in zero-copy global memory, so the record sent between the host and GPU requires only: (1) `uint32_t` for read id, (2) `uint32_t` for low and high values, (3) `uint8_t` for the tuple length value and (4) `uint8_t` for position of tuple (in read as opposed to in reference, so doesn't require `uint32_t` this time). The `struct` is shown in Alg.6.1. This sizing means the average batch requires only 0.45MB, which is permanently allocated for the duration of compression in DRAM, allowing the majority of DRAM to be dedicated to an optimised FM-index.



(a) Memory organisation for CUDA compression architecture.

```
typedef struct
{
  uint32_t id;
  uint8_t start;
  uint32_t low;
  uint32_t high;
}
ref_t;
```

Code Extracts (6.1) (b) Extract from `index.h` - `struct` for a tuple sent to DRAM.

Figure F.2: Memory management for CUDA compression acceleration.

6.4.2 Kernel configuration

As we are now sending batches of tuples to and from the GPU between kernel invocations, additional logic demonstrated in Alg.6.2 is added to our host code responsible for launching the kernel. Specifically, batches of tuples stored in host memory are copied to and from the host using `cudaMemcpyAsync()`, and the respective `cudaMemcpyDeviceToHost` or `cudaMemcpyHostToDevice` flag.

We have found that using significantly larger batch sizes than previously achieves greater performance, using batches of 128,000 tuples instead of only 4000. The number of threads per block remains the same, with a group of 64 threads ($64 \times 1 \times 1$). The block configuration has been increased to a larger linear layout with 2048 blocks ($2048 \times 1 \times 1$). Unlike the previous exact match alignment implementation, we suspect this is in part due to a reduction in the number of zero-copy global memory accesses required to update read alignment properties - instead, the device global memory is updated.

The kernel's volume parameter no long refers to the size of the entire workload, but instead the number of tuples in the batch delegated to the specific kernel invocation. Usually this will be the batch size, however if the number of tuples to be processed is not divisible by this then an additional batch will have less tuples. The actual thread addressing scheme is then simplified, with a thread managing each tuple. A thread's tuple id is calculated by `threadIdx.x + blockIdx.x * blockDim.x`, and if this exceeds the volume of the batch then the kernel terminates.

```
//Records used to store tuple, corresponding read symbols and low/high interval values
ref_t *tuples, *device_tuples;
tuples = new ref_t*[n_tuples_ceil];
...
//Allocate device memory for tuples
size_t tuple_batch_size = sizeof(ref_t) * N_TUPLES_PER_BATCH;
cudaError_t err = cudaMalloc((void**)&device_tuples, tuple_batch_size);
...
for (int batch = 1; batch <= batches; ++batch)
{
  //Calculate number of tuples in batch, and offset of batch in host tuples array
  uint32_t volume = batch < BATCHES ? N_TUPLES_PER_BATCH : n_items % N_TUPLES_PER_BATCH;
  uint32_t offset = (batch - 1) * N_TUPLES_PER_BATCH;
  //Copy tuples to device, run exact match, then copy the updated tuples back
  cudaMemcpyAsync(device_tuples, tuples + offset, tuples_batch_size,
    cudaMemcpyHostToDevice);
  exact_match<<<B, T, batch>>>(batch, volume, device_tuples, ...);
  cudaMemcpyAsync(tuples + offset, device_tuples, tuples_batch_size,
    cudaMemcpyDeviceToHost);
}
cudaDeviceSynchronize();
```

Code Extracts 6.2: Extract from `exact.cu` - launching streams of compressed tuples to kernel.

6.5 Performance Evaluation

In this section we evaluate the performance of our CPU, GPU and FPGA-based compression designs against the referential compressor GDC 2. Due to time restrictions it was not possible to perform a comprehensive evaluation. Primarily this means excluding the performance of FRESCO, as it required significant modifications to ensure only first order compression was performed.

6.5.1 Platform specification

We use the NVIDIA TITAN Black again to test the performance of our GPU-based design. This card only has 6.14GB of GDDR5 DRAM, so again we are unable to store an oversampled FM-index for the human genome on-chip. Consequently we provide projections of the performance for the GPU-based design over data sets involving the human genome, whereas we perform all other tests. The host CPU used is an 8-core Intel E5-1620 v2 3.70GHz, with 16GB RAM.

The performance of the FPGA-based design targets the same platform as our FPGA-based bisulfite sequencing alignment design, the Maxeler MPC-X2000. It is run using 8 DFEs, each equipped with a Altera Stratix V FPGA and 48GB of DRAM. The host CPU used has dual Intel Xeon X5650s with 120GB of DDR3-1333GHz RAM. This CPU is also used to test the performance of our CPU-based implementation, which is compiled with the g++, and paralellised using OpenMP and 16 threads. GDC 2 was also run on this CPU.

6.5.2 Sequencing data sets

We use four different types of sequencing data to test the performance of our compression design against GDC 2, note that due to our design the following files are in the FASTA or FASTQ format:

1. **YH genome sequencing reads:** We re-use the YH genome sequencing reads, which are 10M reads of 100bp sequenced attained by whole genome sequencing. Now that our design is not specialised to a specific read length, these should be representative of a relatively typical

set of reads to be compressed. The reference used for compression is the reference genome (hg19).

2. **Simulated bisulfite sequencing reads:** We also re-use the bisulfite sequencing reads, which are 10M reads of 75bp and simulated using Sherman. This should provide a representative compression performance for the compression of reads generated for use in Methy-Pipe. The referenced used for compression is the C-depleted version of the reference genome.

3. **Whole genome shotgun sequenced chr22[79]:** Realistically, the storage and hence compression of larger sequences will also be required. Therefore, we test the compression of chromosome 22, sequenced using an Illumina HiSeq2000 and whole genome shotgun sequencing. The ambiguous bases (N symbols) in the sequencing data have been trimmed. In the case of the GPU-based aligner, the reference used is simply chromosome 22 from the reference genome as this permits oversampling. With the other platforms, which are not under memory pressure, the reference genome itself is used.

4. **chr1 from the reference human genome (hg19):** As chromosome 22 is rather small, we also compress chromosome 1 to gauge the performance over a longer single FASTA sequence. Again, the ambiguous bases have been trimmed. This is sampled from the reference genome, and compressed against the reference genome. Note that the alignment stage of this compression is performed over 150 symbol segments of the chromosome, as opposed to one large sequence.

6.5.3 GDC 2 Modifications

In order to produce compression ratios and throughputs comparable to that of our simplistic design, we have made a series of modifications to GDC 2:

1. GDC 2 would typically perform two stages of compression, whereby the first would involve generating triples (similar to our tuples). We have only implemented a design with a single stage of compression due to time limitations. Therefore, we have omitted the second round of compression used by GDC 2.

2. We also found the GDC 2 experienced integer overflow when attempting to perform referential compression with a complete human genome, so have split the reference genomes into smaller chunks. Specifically, we divided the genome by two over chromosome 11. The fastest compression time attained over the two sections was used, as this typically involves the most effective mapping, and consequently achieved the highest compression ratio. This ratio should be representative of compression if the integer overflow bug was fixed.

3. The to-be compressed FASTA/FASTQ files in the case of sequencing reads were split into numerous chunks and compressed sequentially. The maximum time to compress a chunk was taken as the compression time, as this would have determined overall run-time in a multi-threading solution (which could have been used). This allows fairer compression time comparisons against our hardware-accelerated solutions.

6.5.4 Compression evaluation

The results in T.I compare the performance of our various acceleration designs against GDC 2. Note that we have omitted I/O times for all designs (including GDC 2), for fairness of comparison.

GDC 2 greatly exceeded almost all of our achieved compression ratios, with the largest difference observed for chr1.trim at 9.1 times. This is because GDC 2 can achieve high compression ratios using variable length encoding over consecutive matches. Specifically, a match of length up

to 2^{23} characters can be encoded in only 7B. Therefore, in cases where the to-be-compressed files are near identical to regions of the reference, compression ratios will greatly exceed those achieved by our design. Unsurprisingly, this was observed for the chr1 and chr22 sequencing data. The bisulfite and YH genome reads are concatenated by GDC 2, and mapped as a single sequence instead of independently. This means the compression ratio for numerous sequences can still be significant, for example the bisulfite read compress rate achieved by GDC 2 was 2.7 times greater than ours. Considering 43% of the bisulfite reads do not exactly align to the reference, a significant number of tuples need to be created for storage with our design, producing an inefficient compression rate of 2.20. We could improve our design such that tuples of the format $\langle pos, len\rangle$, $\langle pos + len, len\rangle$,$\langle pos + 2 \cdot len, len\rangle$ are compressed into a single tuple $\langle pos, 3 \cdot len\rangle$.

In light of the differences in compression ratio, it is unsurprising the run-time of our design is shorter than that of GDC 2 for even the CPU compression design. The difference in run-time greatly exceeded the difference in compression ratio. For example, the compression time for chr22.trim was 1135.0 times slower for GDC 2 than our FPGA-based design, whereas the compression ratio was only 8.8 times better. This indicates that some basic operations, such as tuple concatenation, could greatly reduce the difference in compression ratio without compromising overall run-time. The performance differentials between the CPU, GPU and FGPA aligners weren't entirely surprising. The FPGA implementation was at least 5.9 times faster than the CPU implementation, yet the GPU implementation was only 1.8 times faster. This meagre improvement is due to the increased overhead of copying tuples between device memory and host memory, on top of the already inefficient use of global memory in the design.

Data	Time (s)			Ratio	Time (s)	Ratio
	CPU	GPU	FPGA		GDC 2	
chr22.trim	0 .48	0.26	0 .02	10.18	22.70	89.75
chr1.trim	3.05	1.65*	0.34	11.34	113.40	102.72
YH genome	17.80	9.64*	2.17	8.06	1321.80	8.24
bisulfite	23.58	12.77*	3.95	2.20	1585.40	5.94

Table T.I: Peformance comparison of GDC 2 and our compression designs (CPU, GPU and FPGA). Note that Ratio refers to the compression ratio, and projected GPU performed is indicated with an asterisk (*).

6.6 Summary

In this chapter we have presented an approach to lossless referential compression. We used the alignment algorithms detailed earlier in this report to identify matches between a to-be compressed sequence and a given reference sequence. The motivation behind this was to create a high-throughput referential compression algorithm, in light of the recent advances in next-generation sequencing methods and our considerable optimisation of Methy-Pipe's BSAligner.

We observed much greater compression ratios achieved by GDC 2 than expected, especially for large sequences with few differences to a given reference. This was due to the use of variable run-length encoding, something which our design does not currently implement. The nature of the data was important however, as GDC 2 had at least an 8.8 times greater compression ratio for the chromosomes we compressed, but only a 2.7 times better compression ratio for the bisulfite reads. This difference naturally came at the cost of a significantly reduced compression run-time. For example, GDC 2 barely improved upon our achieved compression ratio for the YH genome reads, but was 609.0 times slower than our FPGA-based design and 124.2 times slower than our projected GPU-based performance.

It is very positive that hardware-accelerated alignment can be used to efficiently perform the matching stage of referential compression. We could easily make modifications to our compression

algorithm, to improve the compression ratio without compromising the low-runtimes. For example, we could run-length encode tuples similar to GDC 2's appriach. We could also explore possible techniques for second order compression.

Chapter 7

Optimisation of Methylation Calling and Downstream Analysis

In this chapter we present software-based optimisations targeting the methylation calling and downstream analysis stages of Methy-Pipe, where methylation calling is the stage following alignment before starting downstream analysis using BSAnalyze. These optimisations address the fact that following hardware acceleration of Methy-Pipe's alignment module, the bottleneck may in fact shift towards the latter pipeline stages. There is a framework for distributing analysis workload across a cluster already in place, so we simply explore software-based optimisations that support individual nodes. Note that we will present these optimisations solely for single-end alignment, however these concepts can be easily extended to paired-end alignment

For systems with significant memory, we have produced a performance optimised version of the C++ module responsible for methylation calling, which should reduce overall runtime from roughly 30 minutes to 8.8 minutes. For systems with less available memory, we reduce the significant memory overhead by almost 52%, from 23.39GB to 11.3GB, with a projected run-time of 33 minutes. This is achieved using a lightweight hash-associative container, Google's `sparse_hash`, alongside genome bit compression. The novel aspect of this contribution is the parallelisation of file output using OpenMP and `system()` calls that leverage Unix processes, improving run-time of the output stage by 2.5 times.

We also translate one of the many Perl scripts used by BSAnalyzer into parallelised C, achieving an improvement in run-time by 45 times, corresponding to a fall in run-time from over 3 hours to around 5 minutes.

7.1 Methylation Calling

Prior to the methylation calling stage of bisulfite sequencing alignment, BSAligner will have produced a set of bisulfite sequencing reads corresponding to C-depleted reads that have successfully aligned to the C-depleted reference genome. For each of these bisulfite sequencing reads, methylation calling involves checking each cytosine against the corresponding base in the reference genome, to see if methylation has occurred at single-base resolution. This is illustrated in F.1.

Figure F.1: Methylation calling against reference genome - for each cytosine in the reference that is covered by an (aligned) read, the corresponding base is checked. If it is a cytosine, methylation has occurred.

```
typedef struct
{
  unsigned short C; // covering cytosines
  unsigned short T; // covering thymines
  unsigned short Z; // neither 'C' nor 'T'
} call;
```

Code Extracts 7.1: Extract from meth_call.cpp - single-base methylation call type definition.

7.2 Existing Design

In Methy-Pipe, the methylation calling modules report methylation at a single-base resolution, the number of CpG sites covered and methylated, the number of uncovered CpG sites and the number of cytosines covered by reads in each chromosome. This is all managed by a C++ module, `meth_call.cpp`, with the primary output file being a `.call` file that details the position of each base in the reference genome with cytosine or thymine coverage in the reads. The exact format is shown in Appendix B(IXb).

In order to record methylation at single-base resolution, the struct shown in Alg.7.1 is used, which stores the number of cytosines, thymines or other bases covering a cytosine in the reference genome. In the existing design, this is consequently initialised for each base in the reference genome, i.e. for roughly 3.2×10^9 bases. This results in a large static heap allocation of roughly 16GB.

For single-end reads, the process of methylation calling is simple, and is composed of four steps: (1) the reference genome is loaded from a given FASTA file (`.fa`) into a `std::map<string, string>` with the key being the chromosome's string identifier and the value being the reference bases; (2) the specified `.bsalign` file is opened and reads are read one at a time, identifying cytosines they cover in the reference genome and updating corresponding `call` structs; (3) the `call` records covered by cytosines and thymines are written to file (`.call`); and finally, (4) the number of coverage bases per chromosome are written to file (`.chr_count`). This is demonstrated in Alg.6.

Algorithm 6 Methylation calling for single-end reads

Input: Reference genome file $hg.fa$ and aligned reads file $reads.bsalign$
Output: Methylation call file $meth.call$ and chromosome counts $cover.chr\_counts$

```
genome ← load_genome(hg.fa)
call* ← (call*) malloc(length(genome))
while reads in reads.bsalign do
    read ← split_soap_line(reads)
    frag_vs_CpG(genome, call, read])
end while
write_methcall(meth.call)
write_chr_counts(cover.chr_counts)

function FRAG_VS_CPG(genome, call*, read)
    for j ← 0 to read.length do
        if genome[i + read.pos] is C then
            update(call[i + read.pos], read.symbols[i])      ▷ Checks methylation state and updates call
        end if
    end for
end function
```

7.2.1 Inefficiencies and bottlenecks

Performing profiling analysis on the original `meth_call.cpp` module using the GNU Profiler, `gprof`, gives an indication of the relative performance of its functions. It was tested using simulated alignment data, comprised of 10M bisulfite sequencing reads sampled from hg19. All the reads

cover cytosines in the reference genome, and are methylated at a rate similar to that of real bisulfite sequencing data. The experimental platform used for this profiling and subsequent testing consists of dual 8-core Intel Xeon E4650 2.70 GHz CPUs with over 32GB main memory. Note that this platform is shared, so resource utilisation may vary and consequently skew results on different runs.

We have used the Intel C++ compiler, `icpc` with optimisation flag `-O3` to compile `meth_call` throughout our experimentation. The notable results are shown in T.I. The total run-time was 70.4s, which indicates that a typical workload would run in roughly 30 minutes. Using the Intel compiler instead of GNU g++ has already accelerated the module by 9.83 times.

For each record processed from an alignment file, it is tokenized using `split_soap_line` before updating the calling records using `frag_vs_CpG`. These operations are incredibly lightweight and close to I/O bound; they run in linear time, $\mathcal{O}(n)$, and together constitute 26.52% of total run-time. The initial memory allocation of calling records, which largely constituted `__intel_memset`, took longer than the time to process the 10M records. To some extent this is unavoidable, and for a typical workload should account for less than 2% of the total run-time. A similar argument applies to `load_genome`, which will account for even less than 2% over a typical workload. Writing the methylation call file is certainly a bottleneck however, with the slow sequential writing of each methylation record constituting a sizable 21.06% of total run-time.

The module has a peak memory usage of 23.39GB, which is also Methy-Pipe's peak memory usage. When Methy-Pipe is used alongside other processes, be it due to multi-tasking or alignment and analysis pipelining, this high memory usage has proven problematic. This memory usage is constituted largely by 16GB of calling records and 4.54GB for the genome reference.

Function Name	Calls	Self Time (s)	% of Total Time
`__intel_memset`	-	22.11	31.42%
`frag_vs_CpG`	10M	17.00	24.16%
`write_methcall`	1	14.82	21.06%
`load_genome`	1	9.09	12.91%
`split_soap_line`	10M	1.59	2.26%

Table T.I: Profiling (`gprof`) results for `meth_call.cpp` - total execution time was 70.4s with peak 23.39GB memory usage. Note that some results have been omitted. These provide smaller contributions to total run time, for example internal `std` libary calls.

7.3 Performance Optimisation

Most systems used in computational genomics will be high performance to meet the demands of high throughput sequencing machines. They will often have numerous cores and in excess of 16GB main memory. Therefore, despite the considerable memory footprint of `meth_call.cpp`, we start by presenting performance optimisations. In the profiling results, note that (P) indicates performance optimisation.

7.3.1 Parallelised file I/O

The previous `meth_call.cpp` module wrote the methylation call file sequentially, one cytosine at a time. This involves inspecting each base in the reference genome sequentially, which is slow, taking roughly 21% of the modules run-time when calling over 10M alignment records.

Calling records are produced for over 24 chromosomes[1], one chromosome at a time. Instead of writing these records sequentially, we have proposed a novel solution that generates

[1] Sequencing and even reference genome construction results in more chromosome sequences than those of the human genome. In downstream analysis the typical chromosomes are often filtered out, namely chromosomes 1-22, X and Y.

chromosome-specific methylation call files in parallel, such that they can then be concatenated to form a single file. Fundamentally, if these files can be written in parallel, the process of merging them will be faster than writing the files sequentially; concatenating two files effectively should only require a one-shot re-allocation of memory and pointer updates.

7.3.1.1 Design

We have used OpenMP to construct a task producer and consumer pattern within `write_methcall`, shown in Alg.7.2. This involves creating a parallel region with `N_THREADS` threads (thread pool), where the code is run sequentially in a single threaded manner until a region denoted by `pragma omp task` is reached. A thread in the pool then claims the task associated with this pragma, launching a function asynchronously. Once the sequential code has been completed, the threads executing tasks (child tasks) will be waited on, synchronising execution using `pragma omp taskwait`.

The function performed by each task is `write_meth_for_chr(...)`, which is responsible for file output. More specifically, it involves creating a unique output stream for one of the chromosomes found in the specified reference genome, writing the corresponding calling records (for methylated cytosines) to file. This allows calling records to be written concurrently, such that while larger chromosomes have their methylation records written to file, numerous smaller chromosomes can also be written to file.

As Methy-Pipe targets Linux x86_64, instead of writing a significant body of code to concatenate these chromosome-specific methylation records into a single file, Unix processes can be used. The Unix processes used are highly optimised C binaries, designed for file management, and consequently outperform any code that could have been written within the scope of this project. The command used is constructed at the end of Alg.7.2 (`concatenate_cmd`), and is passed to a `system()` call, invoking the command processor (`int status = system(concatenate_cmd)`). The command passed to `system()` has several components: (1) `find` is used to identify all chromosome calling files with the path and name specified by the user; (2) `sort` is used to arrange these files by chromosome and remove newline characters; and finally (3) `cat` is used to redirect the content of these calling files into one merged file.

The `system()` call creates detached processes for invoking the specified commands, which can have a considerable overhead. Therefore, a single `system()` call is made, using pipes and redirects. On failure of any one of the chained Unix processes, the error value returned can be analysed and the program can fall back on serial writing of methylation call records to a single file.

```
#pragma omp parallel shared(genome, meth_call_file), num_threads(N_THREADS)
#pragma omp single
{
  for (bit = base_counts.begin(); bit != base_counts.end(); ++bit)
  {
    ...
#pragma omp task shared(base_counts), firstprivate(len, chr, chr_bases)
    write_meth_for_chr(meth_call_file, len, chr, chr_bases, base_counts);
  }
#pragma omp taskwait
}
...
string concatenate_cmd  = "find " + dir + " -maxdepth 1 -type f -name '" + fname + "_*'
    -print0 " + "| sort -z | xargs -0 cat -- >>" + merged_fname + ".call && rm " +
    dir_fname + "*";
int status = system(concatenate_cmd.c_str());
...
```

Code Extracts 7.2: Extract from `meth_call.cpp` - parallelisation of methylation call writing using OpenMP and producer-consumer design.

7.3.1.2 Limitations and drawbacks

Although parallelised file I/O is not novel in itself, this usage of `system()` alongside OpenMP is seemingly novel. However, this is likely because developers have a rightful aversion to the `system()` function.

By using a `system()` call, the portability of code is reduced dramatically. Although this is not currently problematic in the case of Methy-Pipe, as it targets the Linux x86_64 platform, it could become so if wide-spread clinical adoption of Methy-Pipe required porting it to different platforms such as Windows. Preprocessor directives would then be required to switch between `write_methcall` implementations.

`system()` can also produce unpredictable, if not dangerous, results. This isn't surprising, given it involves the creation of detached processes, which cannot be wrangled with error handling the same way that bespoke code could.

Typically, a programmatic approach to managing parallel file I/O involves using a dedicated I/O thread, which uses worker threads to submit formatted data into an output buffer (which can then be written to file). The difference between this approach and that of using `cat` with a `system()` call is that the programmer has an understanding of internal file structure, whereas `cat` simply manages byte streams. This could result in a more complicated concatenation process than a bespoke programmatic approach if internal file structure has to be considered.

7.3.1.3 Evaluation strategy

We evaluate the performance of this approach using bisulfite alignment records simulated from alignment against the reference human genome. The specific schema is specified in Appendix B(IXa). Methy-Pipe performs whole-genome bisulfite sequencing alignment, which should always use the same reference human genome, and consequently produce aligned reads across all chromosomes. Therefore, the `write_methcall` implementation performance is tested solely against this predictable data type.

The number of alignment records requiring methylation calling will influence the run-time for `write_methcall`, as more alignment records will produce more methylated cytosines requiring calling records. The `system()` call overhead should not vary significantly, however the time taken to write chromosome-specific calling records in parallel will increase, alongside the time taken

by the Unix processes to concatenate these records into a single calling file. We therefore test performance of the sequential and parallelised implementations for alignment records varying in size between 10M and 40M. Note that the times, similar to all results previously presented in this paper, are mean run-times from 10 trials over each record set. This is especially important now, as it accounts for the potentially un-predictable behavior of the created Unix sub-process.

7.3.1.4 Performance Evaluation

Profiling demonstrated a significant performance improvement, with the `write_methcall` function contribution half as much as before to the total `meth_call.cpp` execution time. This is shown in T.III. With larger sets of alignment records, it is demonstrated in T.II that the run-time improvement gained through parallelisation is roughly 2.5 times. This is consistent among record sets, although a significant standard deviation and consequent performance loss may only be observable over significantly larger alignment volumes closer to that produced by BSAligner's typical workload of 300M reads - we could not feasibly test this due to memory limitations.

Despite the inherent problems of `system()` calls, the novel use of OpenMP for multi-threading and delegation of file concatenation to Unix processes provides a highly abstracted and simple means of parallelising file output. Although a bespoke implementation would improve code portability and integrity, this approach could be extended to other problem domains as an initial means of facilitating file output parallelisation.

Function Name	Calls	Total Self Time (s)	% of Total Time
`write_methcall`	1	14.82	21.06%
`write_methcall` (P)	1	7.03	11.23%

Table T.II: Profiling (`gprof`) results for `meth_call.cpp` with optimised writing of methylation call file - using 10M alignment records.

Script	Sequential				Parallelised			
`.bsalign` records (M)	10	20	30	40	10	20	30	40
Run-time (s)	100.42	154.96	203.85	256.62	39.48	62.36	78.45	97.38
Speed-up	-				2.54×	2.48×	2.60×	2.58×

Table T.III: Performance improvement of `meth_call.cpp` following parallelisation of writing methylation call file.

7.3.2 Improved tokenisaton and alignment record analysis

The parsing and analysis of alignment records is close to becoming I/O bound. Improving the performance of I/O bound applications can prove challenging, and can be achieved by upgrading a system's hardware. For example, the introduction of SSDs, RAID striping and PCI express hard drives. For large scale clinical adoption of Methy-Pipe, these hardware improvements may not be affordable or achievable for many users. Therefore we optimised the tokenization and analysis of alignment records to ensure the run-time was as close to I/O bound performance as possible.

Initially we experimented with parallelising this stage of methylation calling, by buffering alignment records and passing them to threads. Considering the operation is close to I/O bound however, the overhead of managing these multithreaded tasks did not provide a speed-up. We did however re-write the alignment record tokenisation, using character pointers (`char*`) to iterate between delimiters and an enumerated type to switch over the column numbers that presented the useful information. This meant the read symbol sequence, the chromosome string and position of alignment could be extracted through pointer swapping. The previous implementation used a

series of for loops to iterate between successive columns, and string appended characters individually to a token buffer when the column was one of interest. Other than re-writing the record analysis for clarity, little was altered. We also ensured these functions were inline, to remove any function call overhead and potentially allow for further compiler optimisation.

7.3.2.1 Performance evaluation

Few changes were made to the tokenisation and analysis of alignment records; the `split_soap_line` and `frag_vs_CpG` functions respectively perform trivial tasks, with little scope for improvement. The performance improvements are demonstrated in T.IV. The total time taken to parse and analyse each record experienced a speed-up of 3.76 times.

Function Name	Calls	Total Self Time (s)	% of Total Time
`split_soap_line`	10M	1.59	2.26%
`frag_vs_CpG`	10M	17.00	24.16%
`split_soap_line` (P)	10M (inline)	1.02	1.46%
`frag_vs_CpG` (P)	10M (inline)	4.03	7.02%

Table T.IV: Profiling (`gprof`) results for `meth_call.cpp` with optimised record parsing - using 10M alignment records.

7.4 Memory Optimisation

To ensure methylation could be explored at single-base resolution, the struct shown in Alg.7.1 was previously initialised for each base in the reference genome, i.e. for roughly 3.2×10^9 bases. This requires a large static heap allocation of roughly 16GB. Following successful performance optimisations, we present a few memory optimisations explored to try and soften the memory footprint of `meth_call.cpp` without significantly compromising performance. We have used Valgrind's Massif tool[80] for heap profiling, in order to evaluate memory usage at a fine granularity. In the profiling results, note that (M) indicates memory optimisation, and (PM) indicates memory optimisation targeting the previous performance optimisation.

7.4.1 Reduced bitmap genome

The current implementation of `meth_call.cpp` stores the reference genome as strings of bases, using a `std::map<string, string>` where the key is the chromosome identifier and the value the bases of the chromosome. In 3.1.2.1 we presented how Ramethy creates the n-step FM-index using a reduced version of the human genome, following compression of each character in the alphabet of DNA into a 2-bit equivalent, such that {A, C, G, T} becomes {00, 01, 10, 11}. When loading the reference genome during methylation calling, it can again be stored as a reduced bitmap.

This can be implemented for each chromosome using bit-masking and a vector of `uint64_t` types. An additional integer used to store the number of bases stored in case empty bits in the trailing `uint64_t` are misinterpreted as adenine (A). This type is named `chr_encoded`, and shown in Alg.7.3 alongside a snippet of the masking process. This unsurprisingly slows the loading of the genome, so we have parallelised it using a producer-consumer OpenMP pragma. This involves buffering lines of bases in the FASTA file until a new chromosome header is reached, whereby a task to compress a chromosome is launched asynchronously (`encode_bases` seen in 7.3 is responsible for this).

```
typedef pair<int, vector<uint64_t> > chr_encoded;
typedef unordered_map<string, chr_encoded> genome_encoded;

  ...
static void encode_bases(string bases, chr_encoded &chr)
{
  ...
  uint64_t mask = base == 'T' ? 3 : (base == 'G' ? 2 : (base== 'C' ? 1 : 0));
  mask <<= chr_len % 32 == 0 ? 0 : 64 - 2 * (chr_len % 32);
  ...
}
```

Code Extracts 7.3: Extract from `meth_call.cpp` - bit compression of genome.

7.4.1.1 Memory usage and performance evaluation

Using bit compression to store a reduced bitmap of the genome as opposed to strings of bases reduces genome heap usage from 4.54GB to 1.03GB. This in turn reduces overall memory usage reduced by 15%. We have used OpenMP to improve the rate by which the genome is compressed, however this is a one-off cost that will contribute little to total run-time of a typical workload. The profiling results in T.V demonstrate that the burden of the memory reduction is taken by `frag_vs_CpG`, due to the bit shifting operations needed to regularly retrieve bases from the bit compressed genome. However, with a 15% reduction in memory usage, the time taken to process each alignment record remains faster than the original implementation. Note that for the moment we have omitted `write_methcall` as the previous I/O parallelisation can be used to greatly offset the loss in performance caused by the introduction of bit shifting required by this function.

Version	Function Name	Calls	(M)	(P)	Total Self Time (s)	% of Total Time
v1	`load_genome`	1			9.09	12.91%
	`frag_vs_CpG`	10M			17.00	24.16%
v2	`frag_vs_CpG`	10M (inline)		✓	4.03	7.02%
v3	`load_genome`	1	✓	✓	14.13	23.55%
	`frag_vs_CpG`	10M (inline)	✓	✓	12.41	20.68%

Table T.V: Profiling (`gprof`) results for `meth_call.cpp` with genome bit compression - using 10M alignment records.

7.4.2 Lightweight hash-associative container for calling records

Instead of creating a static array of calling records for each base in the human genome, in turn creating a large region of static memory on the heap, a dynamic data structure can be used. The human genome has approximately 42% GC content, i.e. 42% of nitrogenous bases are either cytosine or guanine. This indicates that at least 58% of the 16GB static allocation could be redundant.

Two approaches can be adopted to manage calling records using dynamic data structures: (1) a fully dynamic approach, where calling records are only created when analysis of an alignment record indicates cytosine coverage on the reference genome; or (2) pre-construction of the calling records, where calling records are created for each cytosine identified in the reference genome when it is loaded. The only difference in parsing and analysis of alignment records is the way in which calling records are updated. When pre-constructing calling records, the number of cytosines, thymines or other bases can simply be incremented for the corresponding record, whereas a fully dynamic approach involves upserting calling records into the respective positions in the data structure. This is achieved by overloading the `+=` operator of the `call_t` struct, as shown in 7.4. We will now explore associative containers, the data structures chosen to be used in these

approaches, before presenting the performance of each of these approaches to identify an optimal solution.

```
struct call_t
{
  unsigned short C;
  unsigned short T;
  unsigned short Z;

  ...

  call_t& operator+=(const call_t& a)
  {
    C += a.C;
    T += a.T;
    Z += a.Z;
    return *this;
  }
}
```

```
...

if ((bucket >> shift) & 0x3)
{
  call_t call_record;
  switch (read.seq[j])
  {
    case 'C' : call_record.C++; break;
    case 'T' : call_record.T++; break;
    default : call_record.Z++;
  }
  calling_records[chr][pos] += call_record;
}
```

Code Extracts 7.4: Extract from `meth_call.cpp` - upserting calling records using an overloaded += operator.

7.4.2.1 Associative container alternatives

In order to store calling records dynamically, a data structure is required that will allow a calling record to be identified by its position within a specific chromosome. The calling records are currently accessed for each chromosome using `std::map`, an associative array found in the C++ standard library. Searching for an element, alongside insertion and deletion is performed in logarithmic time with associative containers - $\mathcal{O}(\log(n))$.

Given that the human genome has approximately 42% GC content, the number of calling records accessed using a chromosome identifier will not equal the number of bases found within the chromosome. Therefore calling records must not only store methylation state, but the position within each chromosome. As these records are identified using positions within the genome as opposed to sequence numbers within the dynamic structure, associative containers are also suitable to store pairs of positions and calling records (`std::pair<int, call_t>`). The structure is illustrated in F.2.

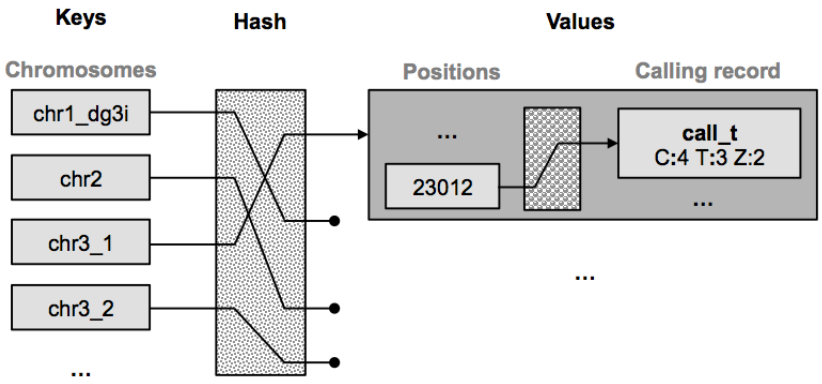


Figure F.2: Methylation call dynamic data structure. In this example, the cytosine at position 23012 in chr3_1 is covered by 4 cytosines, 3 thymines and 2 other bases.

The choice of associative container is not trivial. Assuming 21% of the human genome is composed of cytosine, there may be 672×10^6 calling records. Each pair is composed of three unsigned shorts (shown in 7.1) and an additional unsigned integer for the position, resulting in

10 bytes. Ignoring the associative container storing these associative containers of records, in an ideal world this would require 6.72GB. However, dynamic data structures require housekeeping information to manage records. For example, in the 64 bit GNU Standard C++ Library, `std::map` needs 48 bytes for map alone, and 32 bytes for each object contained[81]. This would increase the total storage size to 28.2GB, which is much larger than the static allocation. In general, an `std::map` storing n elements has a memory usage of $((size\_of(key) + size\_of(value) + 3)) \times n$ bytes, indicating only 8.74GB may be required, which would be a reduction in memory usage by almost 50%. We will consider the following three associative containers:

1. `std::map`: This is an associative array implemented as a balanced tree, requiring traversal and re-balancing. These operations can become expensive and typically make the structure less cache-friendly. It is ordered, which results in a readable methylation call file. This is not crucial however, as many of the downstream Perl modules do not require this ordering.

2. `std::unordered_map`: This is a hashed and unordered associative container which generally uses more memory than `std::map`. It should provide constant operations faster than $\mathcal{O}(\log(n))$ as hashing over positions should not result in collisions. This could be a preferable structure if `std::map` uses a satisfactory amount of memory.

3. Google `sparse_hash`[7]: This is an incredibly lightweight unordered hash-associative container, which is reported to use 2 bits of overhead for each element. According to results presented on Google's documentation, the memory used in each `map_grow` operation is merely 84.4MB compared to the 236.8MB used by the STL map[2]. This comes at a cost however, as the operation is over twice as slow at 665ns. Insertions, deletions and other operations are also slower. This is not surprising given the container's stingy use of memory. `sparse_hash` is used by a few platforms performing bisulfite sequencing analysis, most notably MOAB[82].

7.4.2.2 Memory usage and performance evaluation

To test the dynamic and pre-construction approaches of managing calling records, we have run `meth_call.cpp` using alignment files of varying sizes over the aforementioned associative containers. These alignment files were simulated using hg19, which contains almost 593×10^6 cytosine bases. We have run the Massif heap profiler with default memory snapshot settings, and have omitted the parallelisation of `write_methcall` as this produced spurious results. The results are presented in T.VI. We have chosen not to use `gprof` to evaluate the performance of these strategies with respective to specific calls, as the results become difficult to interpret given the number of internal calls used to manage the dynamic data structures.

It is evident from the results of the pre-construction approach that in the worst case memory utilisation where all the genome's cytosines were covered, both `std::map` and `std::unordered_map` are inappropriate. With pre-construction, the peak memory usage of these containers exceeded the original implementation by over 13%. Unsurprisingly, this is due to the significant overhead required to store an individual calling record. We did not expect the `std::map` to use more memory than `std::unordered_map`, let alone 39% more at 36.8GB.

Memory optimisation using associative containers requires use of `sparse_hash`, with memory usage being reduced by almost 52% in the worst case. Therefore the performance of the two different approaches must be considered. Although the performance of pre-construction was slower over all trials, this can largely be attributed to the increased time taken to load the genome and build the calling records. For example, this took 182.5s on average, which corresponds to 72.9% of the 10M alignment record run-time. This would correspond to only 17.3% of a typical

[2]These results were attained with code compiled with `gcc2.95.3` and the optimisation flag `-O2`. The system used was a 2.80GHz Pentium 4 CPU with 2GB of memory

workload. This also means the calling time for pre-construction using 10M alignment records was roughly 35s on average, whereas for a dynamic approach this was 122s on average. The difference in these times can be attributed to the additional time required to create calling records on the fly; Google claims `sparse_hash` takes 117ns for a `map_fetch` operation, and a hefty 665ns for `map_grow`. Overall, it is evident that the performance of the pre-construction strategy will significantly outperform the dynamic approach, as although the maximum number of costly `map_grow` calls are made when loading the genome, these can be parallelised to mitigate the cost.

Container	`.bsalign` records (M)	Dynamic management		Pre-construction	
		Time (s)	Mem (GB)	Time (s)	Mem (GB)
`std::map`	2	45.0	2.7	164.3	36.8
	5	87.7	5.2	182.25	36.8
	10	132.3	9.0	199.7	36.8
`std::unordered_map`	2	45.2	2.0	77.8	26.5
	5	82.9	3.8	83.4	26.5
	10	128.4	6.3	93.7	26.5
`google::sparse_hash`	2	52.2	1.3	204.7	11.3
	5	110.4	1.8	221.9	11.3
	10	208.3	2.6	250.5	11.3

Table T.VI: Run-time and Massif results for `meth_call.cpp` with dynamic management of calling records.

7.5 Optimum Implementation

Following the various performance and memory optimisations, we can summarise the two versions of `meth_call.cpp` created which maximise performance and minimise memory usage respectively. The performance optimised version contains the calling file output parallelisation and modifications made to record parsing and analysis. The memory optimised version extends the performance optimised version, with genome bit compression and a pre-constructed methylation calling records stored in a dynamic hash-associative container (`sparse_hash`). These two versions and the original version were tested with 10M `.bsalign` alignment records for a final point of comparison, and the results are presented in T.VII.

We were able to achieve close to a 2.5 times improvement in run-time through performance optimisation, without increasing memory usage. This includes a constant time to load the genome, of roughly 19s, which if ignored results in a 3.4 times improvement in run-time. The original developers of Methy-Pipe reported a total run-time of roughly 30 minutes, which would be reduced to 8.8 minutes.

We were also able to reduce the modules significant memory footprint from 23.4GB to only 11.3GB. On the face of it, for the 10M record workload we tested, the performance cost is more than a 1.9 times increase in run-time. The majority of this is due to the time taken to load the genome and construct the dynamic calling record map. For a typical workload of 10M reads, this would correspond to a small proportion of the total runtime, which would be roughly 33 minutes - slightly more than the original implementation.

`meth_call.cpp` Version	Time (s)	Speed-up	Mem (GB)	% Mem Reduction
Original	125.4	-	23.4	-
Perf. Optimised	50.3	2.49×	23.4	0%
Mem. Optimised	242.1	-1.93×	11.3	51.7%

Table T.VII: Performance results for optimised `meth_call.cpp` - comparison of the original version against our two versions for 10M `.bsalign` alignment records

7.6 Translating Perl scripts

The management of methylation call records after `meth_call.cpp`, but before and during subsequent downstream analysis is performed almost solely by Perl and R scripts. Perl is used heavily within bioinformatics as a scripting language given it is highly portable, has a wide selection of libraries, uses dynamic typing and has a useful regex engine. However, large volumes of alignment data are handled in a typical Methy-Pipe workload, which can result in poor performance.

We have translated one of Methy-Pipe's Perl scripts into parallelised C in order to demonstrate the performance gain that can be achieved, in turn accelerating modules which may otherwise prove to be downstream bottlenecks. In particular, we translated the `calc_met_per_chr.pl` script which merges the calling records (`.call`) by chromosome for the Watson and Crick strands and reports methylation density. This module was chosen given its simplicity, yet inefficient design, shown in Alg.7.5. It iterates over two sets of calling records line by line, adding the cytosine and thymine coverage to the map `met`. For each chromosome, the methylation density is then printed such that the shell output can be piped into a log for subsequent analysis.

```
#Params: <W.CpG.call> <C.CpG.call>\n";

my %met;
while(<>)
{
  chomp;
  my @F=split;
  $met{$F[0]}{C}+=$F[4];
  $met{$F[0]}{T}+=$F[5];
  if($F[0]=~/chr\d+/)
  {
    $met{chrT}{C}+=$F[4];
    $met{chrT}{T}+=$F[5];
  }
}
for('T', 1..22, 'X', 'Y')
{
  my $chr="chr$_";
  if(exists $met{$chr})
  {
    my $metD=$met{$chr}{C}/($met{$chr}{C}+
    $met{$chr}{T})*100;
    print join("\t", $chr, $metD), "\n";
  }
  else
  {
    print join("\t", $chr, 0), "\n";
  }
}
```

Code Extracts 7.5: `calc_met_per_chr.pl` - merges of Watson and Crick calling records and reports methylation density.

This script can be trivially parallelised using OpenMP in C, calculating the cytosine and thymine coverage of each chromosome for the Watson and Crick strands independently before iterating through each chromosome, merging the records and finally printing the methylation. To avoid atomic operations that introduce serialisation between the two threads needed for managing each strand, we have avoided using a shared structure and simply created two static arrays. Alg.7.6 is an extract from the translated script, `calc_met_per_chr.c` which demonstrates the parallelisation.

```
#pragma omp parallel num_threads(2)
{
  if (omp_get_thread_num() == 0)
    calc_met_per_strand(w_call_file, w_counts);
  else
    calc_met_per_strand(c_call_file, c_counts);
}
```

Code Extracts 7.6: Extract from `calc_met_per_chr.c` - parallelisation using OpenMP over Watson and Crick strands.

Performance gain: We tested the C and Perl scripts using pairs of `.call` records produced using `meth_call.cpp` on simulated data sets of bisulfite sequencing read alignments (`.bsalign`). The largest data sample used was 20M alignments, which corresponds to roughly 13% of a typical workload. The results are displayed in T.VIII. The improvement in runtime was over 45 times following translation from Perl to optimised C. This is not surprising, as roughly half the workload can be immediately eliminated by parsing the Watson and Crick strands in parallel, and compiled code can perform significantly faster than interpreted languages such as Perl.

Script	Perl			C		
`.bsalign` records (M)	5	10	20	5	10	20
Run-time (s)	220.7	452.12	821.10	4.84	8.77	18.10
Speed-up	-			45.60×	51.6×	45.4×

Table T.VIII: Comparison of run-time performance between Perl and C version of `calc_met_per_chr` module.

7.7 Summary

In this chapter we presented optimisations targeting the C++ module primarily responsible for methylation calling, `meth_call.cpp` The previous implementation used a significant amount of memory, however the lightweight implementation meant a complete workload could be processed in roughly 30 minutes. We have demonstrated how performance optimisations, including novel parallelisation of file I/O, could reduce run-time to 8.8 minutes. We have also demonstrated a couple of memory optimisations that can reduce the memory usage to 11.3GB, a 51.7% reduction, which comes at the cost of a 10% slower run-time. This would correspond to a negligable three minutes for a typical workload.

We also demonstrated the performance that can be achieved by translating the Perl scripts which follow `meth_call.cpp` into C. The example used was the module which provides chromosome specific methylation densities, `calc_met_per_chr`, which experienced a 45 times improvement in run-time. Projections based on this run-time suggest the run-time for a typical workload could consequently fall from over 3 hours to under 5 minutes.

The most significant improvement in run-time we achieved was evidently through translation of the Perl script. Although we saw an improvement in the run-time for methylation calling, it seems that there is significant potential for further improvement to be made further downstream. This suggests that although extensive hardware acceleration has targeted Methy-Pipe's BSAligner, the further translation of highly inefficient and crucial Perl scripts could render BSAligner the pipeline bottleneck once again.

Chapter 8

Conclusion

The results of optimising Methy-Pipe are summarised in T.I, and we can reiterate the key achievements initially outlined in the introduction to this thesis:

1. Optimisation of Methy-Pipe's BSAligner:

   (a) We have optimised Ramethy, a runtime-reconfigurable bisulfite sequencing alignment design, in order to return all candidate inexact alignment locations to the host CPU. This is necessary if the design is to support paired-end alignment. We presented two optimisations to reduce the off-chip DRAM accesses used by this design, and one to reduce the burden of inexact alignment placed on the inefficient FM-index oriented modules. The design reduces the overall alignment time of BSAligner from 5 hours to 13 minutes, corresponding to a significant speed-up of 22.7 times.

   (b) We implemented the first stage of the same bisulfite sequencing alignment design using GPUs. The design outperforms state-of-the-art GPU-based sequence aligners SOAP3-dp and BarraCUDA, by 2.8 and 6 times respectively. Projections suggest a full implementation could align BSAligner's typical workload in 42 minutes. This suggests a complete GPU-based alignment design could support our FPGA-based design in a novel heterogeneous system.

2. Optimisations targeting BSAnalyzer: we presented software optimisations targeting downstream analysis scripts, and the C++ module responsible for methylation calling (the process proceeding all downstream analysis). This module had a significant memory footprint, so we reduced the memory usage by almost 52%, from 23.39GB to 11.3GB, with a negligible effect on run-time. In the case that host CPUs have 24GB of RAM, we have also produced a performance optimised version of this module that is 2.5 times faster with the same memory utilisation. We also translated one of the many Perl scripts used by BSAnalyzer into parallelised C. The module was responsible for reporting the methylation of each chromosome in the genome, a basic regional statistic. We achieve an improvement in run-time by 45 times, corresponding to a fall in run-time from over 3 hours to around 5 minutes. This indicates that any inefficiencies remaining in Methy-Pipe could be eliminated.

3. Additionally, we presented a novel approach to lossless referential compression of genetic sequencing data. This approach leveraged the alignment algorithms used throughout this report, and we created novel FPGA and GPU-based designs. With a simple alignment algorithm, we achieved a significantly greater compression speed than referential compressor GDC 2. For example, in the case of compressing 10M bisulfite sequencing reads from a human genome, our FPGA-based design runs 401 times faster than GDC 2 at 3.95 seconds. For this data set, our design achieves a compression ratio of 2.20, however GDC 2 with its superior compression algorithm achieved 5.94. Following a few small extensions that could dramatically improve the compression ratio, our approach to compression could consequently be utilised on hosts running Methy-Pipe to counter the increasingly demanding storage requirements of patients' bisulfite sequencing data.

Methy-Pipe Module	Original Time	New Time	Speed-up
BSAligner using 8 FPGAs	≈300m0s	13m12s	22.7×
BSAligner using 1 GPU	≈300m0s	42m30s	7.1×
`meth_call`, high performance	30m45s	8m48s	3.5×
`meth_call`, 52% less memory	30m45s	32m51s	-1.1×
`calc_met_per_chr` in parallelised C	205m17s	4m32s	45.4×
Referential compression using 8 FPGAs*	792m42s	1m59s	401.4×
Referential compression using 1 GPU*	792m42s	6m23s	124.2×

*compared to GDC 2 first order compression

Table T.I: Final optimisation results for the targeted Methy-Pipe modules - these results are arrived at through projections based upon the testing presented throughout this report, for a typical workload of 300M bisulfite sequencing reads.

8.1 Future Work

In this section we present possible avenues of further research that this project has enabled, alongside further optimisations that can be applied to Methy-Pipe.

8.1.1 Further Methy-Pipe Work

The optimisations presented in this thesis largely targeted Methy-Pipe's BSAligner, as it was previously the pipeline bottleneck; the BSAnalyzer unlike BSAligner is able to be parallelised in its current implementation over numerous nodes in a cluster. Following the optimisation of BSAligner and the methylation calling module, it is evident that any significant bottlenecks could only occur further downstream in BSAnalyzer. We were able to accelerate the performance of a downstream analysis script written in Perl by 45 times, simply by translation to C. This corresponds to a reduction in runtime for a typical workload from over 3 hours to under 5 minutes. This was a basic script, performing regional methylation analysis, suggesting there may be more elaborate scripts demanding additional attention.

The one aspect of the optimisations presented requiring immediate attention is the parallelisation of file output for methylation calling. This involved using OpenMP and C++ `system()` calls in a novel way to create numerous files in parallel before concatenating them with Unix processes. We discussed the limitations of this approach, largely highlighting the fragility of the `system()` function. In the future, a more typical programmatic approach should be taken.

8.1.2 Clinical and Medical Research Aspects

The scientific significance of this project is yet to be determined, as the proposed optimisations are yet to be realised by Methy-Pipe within a clinical or research environment. The dramatic reduction in alignment time for a typical workload could result in unconventional forms of future work however, such as altering clinical appointment schedules for patients, due to changes in the achievable patient throughput. If the accelerated performance of Methy-Pipe helps enable the adoption of whole-genome bisulfite sequencing procedures, such as non-invase pre-natal diagnosis, there may also be significant clinical work required to adopt these procedures.

From a research perspective, the reduction in alignment time may support use of higher throughput sequencing techniques, and in turn larger genetic sequencing data sets which could be used to study the patterns of methylcytosines in numerous areas of medicine including: epigenetics, embryonic development, post-natal development, carcinogenesis and even the study of methylation in different organisms (such as bacteria).

8.1.3 Computing Research Aspects

Although the contributions presented in this report largely target Methy-Pipe, many of the optimisations are not software or hardware specialised, and can be used widely to accelerate alignment and compression solutions. Our exploration of using FPGA and GPU-based techniques in particular to accelerate multiple sequence alignment and referential compression has posed a few avenues for further work:

- *Different FPGA-based architectures:* Accessing off-chip FM-index buckets from DRAM was expensive for both the FPGA and GPU-based designs, with a latency in the order of hundreds of cycles. Resource utilisation in the FPGA-based architecture was roughly a third of that available, limited by the number of available memory controllers. If a different FPGA-based architecture to the Maxeler MPC-X2000 was used, such as the Convey HC-2, we would benefit from a substantially higher bandwidth of 80GB/s and improved non-sequential access speeds thanks to Convey Scatter-Gather DIMMs. In turn, this would allow us to increase the number of modules that could be fabricated during any one configuration.

- *Coalesced FM-index access patterns:* In the case of the GPU-based design, CUDA attempts to coalesce global memory loads to allow efficient DRAM usage, however the FM-index is not accessed sequentially and buckets are large, resulting in load serialisation. If the occupancy of the design is to improve, maximising compute resource utilisation, the FM-index access pattern must be coalesced. Research could be undertaken to understand how read workload could be balanced across CUDA warps to allow FM-index access localisation, for example using pre-computation and heuristics.

- *Heterogeneous alignment platform:* We presented significant GPU-based alignment performance, which despite an inefficient design, could contribute to a heterogeneous design. The true power of a heterogeneous system however would be achieved when playing to the true strengths of the GPU, allowing different applications (not simply alignment) to use the system appropriate for their unique demands on computing resources. This would require exploring the operations performed downstream for analysis, as GPUs can outperform FPGA-based platform for tasks common to high productivity computing such as matrix and vector arithmetic[83] - these are commonplace tasks in scientific research. A specific application of GPUs in bioinformatics is regression analysis used to identify associations between genetic variants[84], which could in turn be used to identify common methylation patterns at a single-base resolution. Alternatively, Methy-Pipe's approach to DMR identification using a sliding window, presented in 2.4.4, could be parallelised using CUDA - parallel sliding windows are explored in the context of genomic analysis in [85]. Methy-Pipe is unique in that it incorporates both alignment and analysis, and the creation of a heterogeneous integrated alignment and analysis platform using FPGAs and GPUs would be novel.

- *Alternate hardware-accelerated referential compression strategy:* We presented an incredibly simple approach to referential compression, involving exact matching sequences and subsequently splitting them if an exact match cannot be found. Each sequence is stored as a a series of tuples, containing the positions and lengths of regions in the reference that can be concatenated to form the query sequence. Although this approach required minimal alteration of the exact match alignment designs already produced on GPU and FPGA, an alternate approach could be adopted to improve the compression ratio and throughput: modify the hardware designs to exact match a given sequence numerous times, saving intermediate mismatch symbols in triples also containing position and length. These triples can be stored in faster on-chip memory (e.g. BRAM on the FPGA), before being streamed to the host following full alignment. This would reduce the host workload drastically.

8.2 Closing Remarks

In this project we have attempted to address the increasing strain placed on platforms intending to harness the plethora of genetic sequencing data that can be created by next-generation sequencing machines. We have focused our efforts on the optimisation of Methy-Pipe, an integrated bioinformatics pipeline which facilitates both the sequence alignment and analysis requirements for healthcare and scientific study of DNA methylation. We have successfully been able to accelerate and optimise both the alignment and analysis modules of Methy-Pipe, dramatically reducing the runtime of its bottlenecks for a typical workload from 5.5 hours to 22 minutes; shown in T.I, alignment and methylation calling alone previously took 5.5 hours, and the analysis could take several hours if the workload is not distributed across numerous processors.

This improvement bears the potential to facilitate further adoption of life-saving healthcare applications, such as non-invasive pre-natal diagnosis and the detection of aberrant methylation patterns in cancer biopsies. If Methy-Pipe was adopted widely, it may not simply be found within a clinical environment, but is now feasible for extensive use within research, which could further our understanding of epigenetic abnormalities and how we can reverse them pharmacologically in novel healthcare applications.

This project has also demonstrated the sheer extent to which computational techniques, namely hardware acceleration of substring index search algorithms, can be applied to multiple sequence alignment and related problems such as referential compression. Given the cost of a state-of-the-art heterogeneous alignment platform can be in the magnitude of thousands less than a next-generation sequencing machine, there is the definite potential for hardware-accelerated integrated alignment and analysis platforms to become tightly coupled with sequencing machines. This could ease the management of high throughput sequencing data, alongside further automating sequencing data analysis to improve the personalisation of medical care and patient throughput.

List of Tables

List of Figures

List of Algorithms

List of Code Extracts

Bibliography

[1] E.C. Hayden. *Technology: The 1000 dollar genome*. URL: http://www.nature.com/news/technology-the-1-000-genome-1.14901.

[2] P. Jiang et al. "Methy-Pipe: An Integrated Bioinformatics Pipeline for Whole Genome Bisulfite Sequencing Data Analysis". In: *PLOS ONE* (2014).

[3] J. Kitzman et al. "Noninvasive whole-genome sequencing of a human beingus." In: *Science Translational Medicine* (2012).

[4] K.C.A. Chan et al. "Noninvasive detection of cancer-associated genome-wide hypomethylation and copy number aberrations by plasma DNA bisulfite sequencing." In: *Biological Sciences, PNAS* (2013), pp. 18761–18768.

[5] R. Luo et al. "SOAP3-dp: Fast, Accurate and Sensitive GPU-Based Short Read Aligner". In: *PLOS ONE* (2013).

[6] P. Klus et al. "BarraCUDA - a fast short read sequence aligner using graphics processing units". In: *BMC Research Notes* (2012).

[7] Google. *Sparse Hash*. URL: https://code.google.com/p/sparsehash/.

[8] S. Deorowicz and S. Grabowski. "Robust relative compression of genomes with random access". In: *Bioinformatics* (2011), pp. 2979–2986.

[9] J. Arram et al. "Leveraging FPGAs for High Throughput Bisulfite Sequence Alignment". 2015.

[10] D. Graur and R.A. Cartwright. "The multiple personalities of Watson and Crick strands". In: *Biology Direct* (2011).

[11] B. Strengholt and M. Brobbel. "Acceleration of the Smith-Waterman algorithm for DNA sequence alignment using an FPGA platform". 2013.

[12] SIGMA-ALDRICH. *Introduction to DNA Methylation*. URL: http://www.sigmaaldrich.com/technical-documents/articles/biofiles/introduction-to-dna-methylation.html.

[13] J.T. Attwood, R.L. Young, and B.C. Richardson. "DNA methylation and the regulation of gene transcription". In: *Cellular and Molecular Life Sciences CMLS 59* (2002), pp. 241–257.

[14] K.D. Robertson. "DNA methylation and human disease". In: *Nature Reviews Genetics* (2005), pp. 597–610.

[15] W. Yang and H. Pan. "Regulation mechanism and research progress of MeCP2 in Rett syndrome." In: *Hereditas* (2014), pp. 625–30.

[16] R. Brown and G. Strathdee. "Epigenomics and epigenetic therapy of cancer". In: *Trends in Molecular Medicine* 8 (2012), pp. 457–463.

[17] atdbio. *Sequencing forensic analysis and genetic analysis.* URL: http://www.atdbio.com/content/20/Sequencing-forensic-analysis-and-genetic-analysis.

[18] F. Sanger, S. Nicklen, and A.R. Coulson. "DNA sequencing with chain-terminating inhibitors". In: *Proceedings of the National Academy of Sciences* 74 (1977), pp. 5463–7.

[19] L.M. Smith et al. "Fluorescence detection in automated DNA sequence analysis". In: *Nature* 321 (1986), pp. 674–679.

[20] Illumina. *Illumina Sequencing Introduction*. Tech. rep. URL: http://www.illumina.com/content/dam/illumina-marketing/documents/products/illumina_sequencing_introduction.pdf.

[21] E. Zimmerman. *50 Smartest Companies: Illumina*. 2014. URL: Technologyreview.com.

[22] A. Regalado. *EmTech: Illumina says 228000 Human Genomes will be sequenced this year*. 2014. URL: Technologyreview.com.

[23] S. Young. *Illumina Claims It's Reached 1,000 dollar-Genome Milestone*. 2014. URL: Technologyreview.com.

[24] S. D. Kahn. "On the Future of Genomic Data". In: *Science* (2011), pp. 728–729.

[25] James Hadfield. *Comparing Illumina Sequencers*. 2015. URL: http://core-genomics.blogspot.co.uk/2015/02/comparing-illuminas-sequencers.html.

[26] W.R. Pearson and D.J. Lipman. "Improved tools for biological sequence comparison". In: *Proceedings of the National Academy of Sciences* 85 (1988), pp. 2444–2448.

[27] P.J.A. Cock et al. "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants". In: *Nucleic Acids Research* 38 (2010), pp. 1767–1771.

[28] B. Ewing et al. "Base-calling of automated sequencer traces using phred. I. Accuracy assessment." In: *Genome Research* (1998), pp. 175–85.

[29] B. Ewing and P. Green. "Base-calling of automated sequencer traces using phred. II. Error probabilities". In: *Genome Research* (1998), pp. 186–94.

[30] P. Ferragina and G. Manzini. "An experimental study of an opportunistic index". In: *Society for Industrial and Applied Mathematics - Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms* (2001), pp. 269–278.

[31] T.F. Smith and M.S. Waterman. "Identification of Common Molecular Subsequences". In: *Journal of Molecular Biology* (1981).

[32] M. Burrows and D.J. Wheeler. "A Block-sorting Lossless Data Compression Algorithm". In: *SRC Research Report* (1994).

[33] U. Manberand and G. Myers. "Suffix arrays : A new method for on-line string searches." In: *SIAM Journal on Computing* (1992), pp. 935–948.

[34] Wikipedia. *Smith Waterman Algorithm, Wikipedia*. URL: http://en.wikipedia.org/wiki/Smith-Waterman_algorithm.

[35] R. Lister et al. "Highly Integrated Single-Base Resolution Maps of the Epigenome in Arabidopsis". In: *Cell Volume 133* (2008), pp. 523–536.

[36] F. Krueger and S. R. Andrews. "Bismark: a flexible aligner and methylation caller for Bisulfite-Seq applications". In: *Bioinformatics* (2011), pp. 1571–1572.

[37] PY. Chen, S. J. Cokus, and M. Pellegrini. "BS Seeker: precise mapping for bisulfite sequencing". In: *Bioinformatics* (2010).

[38] T. Benoukraf et al. "GBSA: a comprehensive software for analysing whole genome bisulfite sequencing data". In: *Nucleic Aids Research* (2012), p. 55.

[39] K. D. Hansen, B. Langmead, and R. A. Irizarry. "BSmooth: from whole genome bisulfite sequencing reads to differentially methylated regions". In: *Biology* (2012).

[40] T.W. Lam et al. "High Throughput Short Read Alignment via Bi-directional BWT." In: *IEEE International Conference on Bioinformatics and Biomedicine* (2009), pp. 31–36.

[41] R.Q. Li et al. "SOAP2: an improved ultrafast tool for short read alignment." In: *Bioinformatics* (2009).

[42] Zymo Research. *Bisulfite Beginner Guide*. URL: http://www.zymoresearch.com/bisulfite-beginner-guide.

[43] J. Gailly and M. Adler. 2003. URL: http://www.gzip.org/.

[44] E. S. Lander et al. "Initial sequencing and analysis of the human genome". In: *Nature* (2001), pp. 860–921.

[45] *Assembly of the human genome*. 2009. URL: http://hgdownload.cse.ucsc.edu/goldenPath/hg19/chromosomes/.

[46] B. G. Chern et al. "Reference Based Genome Compression". In: *IEEE Information Theory Workshop* (2012), pp. 427–431.

[47] D. A. Wheeler et al. "The complete genome of an individual by massively parallel DNA sequencing". In: *Nature* 452 (2008), pp. 873–876.

[48] National Center for Biotechnology Information. *dbSNP - Short genetic variations database*. URL: http://www.ncbi.nlm.nih.gov/snp/.

[49] S. Christley et al. "Human Genomes as email attachments". In: *Bioinformatics* 25 (2008), pp. 274–275.

[50] A. Lempel and J. Ziv. "A universal algorithm for sequential data compression". In: *IEEE Transactions on Information Theory* 24 (1977), pp. 337–343.

[51] M. Pflanzer and W. Luk. "Optimised Compression of Genetic Sequencing Data". 2015.

[52] S. Wandelt and U. Leser. "FRESCO: Referential Compression of Highly-Similar Sequences". In: *IEEE Transactions on Computational Biology and Bioinformatics* (2013), pp. 1275–1288.

[53] J. Karkkainen, D. Kempa, and S. J. Puglisi. "Linear Time Lempel-Ziv Factorization: Simple, Fast, Small". In: *Data Structures and Algorithms, Cornell University Library* (2012), pp. 89–200.

[54] C. Kingsford and R. Patro. "Reference-based compression of short-read sequences using path encoding". In: *Bioinformatics* (2015).

[55] S. Grabowski, S. Deorowicz, and L. Roguski. "Disk-based compression of data from genome sequencing". In: *Bioinformatics* (2014).

[56] 1000 Genomes Project Consortium et al. "A map of human genome variation from population-scale sequencing." In: *Nature* (2010), pp. 1061–1073.

[57] China National GeneBank (CNGB). *BioCloud - biological data storage service.* URL: http://biocloud.cngb.org/.

[58] Khronos Group. *OpenCL, Open Computing Language*. URL: https://www.khronos.org/opencl//.

[59] NVIDIA. *CUDA, Compute Unified Device Architecture*. URL: http://www.nvidia.com/object/cuda_home_new.html.

[60] Wikipedia. *CUDA Processing Flow, Wikipedia*. URL: http://en.wikipedia.org/wiki/CUDA.

[61] NVIDIA. *Parallel Thread Execution Architecture*. URL: http://docs.nvidia.com/cuda/parallel-thread-execution/.

[62] Maxeler Technologies. *Multiscale Dataflow Programming*. 2013.

[63] Maxeler Technologies. *MaxCompiler*. URL: http://www.maxeler.com/products/software/maxcompiler/.

[64] J. Arram, W. Luk, and P. Jiang. "Ramethy: Reconfigurable Acceleration of Bisulfite Sequence Alignment". In: *FPGA* (2015), pp. 250–259.

[65] A. Chacon et al. “n-step FM-index for faster pattern matching.” In: *Procedia Computer Science 18* (2013), pp. 70–79.

[66] T.W. Lam et al. “High throughput short read alignment via bi-directional BWT”. In: *The IEEE International Conference on Bioinformatics and Biomedicine (BIBM 2009)* (2009), pp. 31–36.

[67] Babraham Bioinformatics Institute. *Sherman - bisulfite-treated Read FastQ Simulator*. URL: http://www.bioinformatics.babraham.ac.uk/projects/sherman/.

[68] E. Fernandez, W. Najjar, and S. Lonardi. “String Matching in Hardware Using the FM-Index”. In: *Field-Programmable Custom Computing Machines (FCCM)* (2011), pp. 218–225.

[69] E. Fernandez et al. “Multithreaded FPGA acceleration of DNA sequence mapping.” In: *HPEC* (2012), pp. 1–6.

[70] C.B. Olson et al. “Hardware acceleration of short read mapping”. In: *Field-Programmable Custom Computing* (2012), pp. 161–168.

[71] C. W. Yu et al. “A Smith-Waterman Systolic Cell”. In: *Field Programmable Logic and Applications* (2003), pp. 375–384.

[72] Maxeler Technologies. *MPC-X Series*. URL: https://www.maxeler.com/products/mpc-xseries/.

[73] B. Langmead and S.L. Salzberg. “Fast gapped-read alignment with Bowtie 2”. In: *Nature Methods* (2012), pp. 357–359.

[74] H. Li and R. Durbin. “Fast and accurate short read alignment with Burrows-Wheeler transform.” In: *Bioinformatics* (2009), pp. 1754–60.

[75] *Raw reads from the YH (Homo sapiens) genome (version SOAPdenovo2)*. URL: http://www.ebi.ac.uk/ena/data/view/ERP001652.

[76] Convey. *Convey HC-2 Architectural Overview*. Tech. rep. URL: http://www.conveycomputer.com/files/4113/5394/7097/Convey_HC-2_Architectual_Overview.pdf.

[77] X. Meng and V. Chaudhry. “A High-Performance Heterogeneous Computing Platform for Biological Sequence Analysis”. In: *IEEE Transactions on Parallel and Distributed Systems* 21 (2010), pp. 1267–1280.

[78] Y. Liu, B. Schmidt, and D.L. Maskell. “CUSHAW: a CUDA compatible short read aligner to large genomes based on the BurrowsWheeler transform”. In: *Bioinformatics* (2012).

[79] *Homo sapiens chromosome 22, alternate assembly CHM1_1.1, whole genome shotgun sequence*. URL: http://www.ncbi.nlm.nih.gov/nuccore/528476531.

[80] Valgrind. *Massif*. URL: http://valgrind.org/docs/manual/ms-manual.html.

[81] David. *STL Container Memory Usage when Developing with C++*. 2014. URL: http://info.prelert.com/blog/stl-container-memory-usage.

[82] “MOABS: model based analysis of bisulfite sequencing data”. In: *Genome Biology* (2014).

[83] D.H. Jones et al. “GPU versus FPGA for high productivity computing”. In: *2010 International Conference on Field Programmable Logic and Applications* (2010), pp. 119–124.

[84] “CARAT-GxG: CUDA-Accelerated Regression Analysis Toolkit for Large-Scale Gene-Gene Interaction with GPU Computing System”. In: *Cancer Informatics* (2014), pp. 27–33.

[85] B. Kreuter et al. “Accelerating Genomic Analyses with Parallel Sliding Windows”. 2010.

[86] P. Jiang. *Methy-Pipe Manual*. URL: http://137.189.133.71/methy-pipe/data/Methy-pipe.manual.pdf.

Appendices

Appendix A

Accelerated Genome Compression Algorithm

Algorithm 7 Referential compression with exact match alignment - the key auxiliary functions are detailed for completeness.

Input: Set of sequencing $reads$, reference FM-index F, suffix array intervals SAI and reference suffix array SA
Output: Sequence-to-reference mapping tuples, T

```
step ← 1                                                    ▷ Initially, single tuple per read
T ← generate_input(reads, step)                             ▷ Generate tuples for alignment
exact_align(T, F, SAI)
parse_output(reads, step, T)
reverse_complement(reads)                                   ▷ Get DNA reverse complement of read sequences
while unaligned tuple in T do                               ▷ Keep generating tuples and aligning them until termination
    T ← generate_input(reads, step)
    exact_align(T, F, SAI)
    parse_output(reads, step, T)
    step ← step × 2                                         ▷ Increase step, causing each tuple to be split in half
end while

function PARSE_OUTPUT(reads, step, T)
    for tuple in T do
        i ← get_tuple_index_in_mapping(reads[tuple.id].mapping, tuple)
        if tuple.low ≤ tuple.high then                      ▷ Parse aligned tuple
            reads[tuple.id].mapping[i].pos ← convert_pos_to_reference(tuple.low)
        else                                                ▷ Parse unaligned tuple
            tuple_1 ← (tuple.pos, tuple.len/2)
            tuple_2 ← (tuple.pos + tuple_1.len, tuple.len − tuple_1.len)
            reads[tuple.id].mapping[i] ← tuple_1
            reads[tuple.id].mapping[i + 1] ← tuple_2
        end if
    end for
end function

function GENERATE_INPUT(reads, step)
    T ← []
    for read in reads, where read.discard != true do
        for j ← 0 to read.mapping.size() do
            if read.mapping[j].len ≤ ⌈read.len/step⌉ then
                compressed_read ← (read.id, read.symbols, read.mapping.pos, read.mapping.len)
                T.append(compressed_read)
            end if
        end for
    end for
    return T
end function

function EXACT_ALIGN(read_tuples, F, SAI)
    ...                                                     ▷ FM-index search algorithm, ideally using n-step and oversampling
end function
```

Appendix B

Methy-Pipe Manual

On the following pages we have included the Methy-Pipe manual (v2.02)[86]. This details the Methy-Pipe directory structure, provides some usage instructions and gives examples of output files.

Methy-Pipe Manual (v2.02)

The structure of directory hierarchy for Methy-Pipe and analysis output.

(1) Methy-Pipe pipeline structure.

methy-pipe2

```
|-methy-pipe2.pl          # main script used to run Methy-Pipe
|-cpp_prog/               # binary programs compiled from C++
|-perl_prog/              # perl programs
|-R_prog/                 # R scripts (requires ggplot2 and gridExtra packages)
|-bed_files/              # bed files such as TSS regions frequently used to calculate
                          # the regional methylation density
|-utils/                  # extra utilities such as DMRs mining etc.
  |-2bwt-builder/         # scripts can be used to build the BWT index for a reference genome
  |-split_meth_call/      # scripts can be used to split the methylation call files based on
                          # each chromosome.
  |-DMR/                  # programs can be used to identify the differentially methylated regions
                          # based on above split methylation call files
  |-bed_files/            # containing regions of interest in bed format such as TSS regions,
                          # gene regions, etc
  |-database/             # containing reference genome
```

(2) Methy-Pipe output structure.

Methy-Pipe_output

```
|-outprefix_alignment/    # BS-Seq alignment results
|-outprefix_meth_call/    # methylation calling results
|-outprefix_meth_density/ # methylation density profiling by using a fixed window size
|-outprefix_summary/      # summary for the Methy-Pipe
|-outprefix_logs/         # intermediate results that can be deleted by users
|-DMRs/                   # analyzing the differentially methylated regions
```

How to use Meth-Pipe

I. If you want to quickly start Methy-Pipe, please use the following shell scripts in dataset folder. It will show some key instructions of how to run it. For the detailed implementation, please refer to the following section and manuscript.

```
#IMPORTANT:
#Please install the ggplot2 (http://ggplot2.org/) and gridExtra
(http://cran.r-project.org/web/packages/gridExtra/) libraries for R before
starting Methy-Pipe.
#uncompress the files:
tar xjf methy-pipe2.full.tar.bz2
tar xjf dataset_light.tar.bz2
cd dataset_light
#create a makefile for BS-Seq alignment and methylation calling:
./wk.sh
#change to the output result folder:
cd Methy-Pipe_output
#Use makefile to run the Methy-Pipe:
make
#if users want to use 2 parallel computing nodes to run
#Methy-Pipe based on SGE platform, run the following command:
#./qsub.sh 2 makefile
#Split the methylation call by each chromosome:
./demo_split_call.sh
#change to DMRs identification folder
mkdir DMRs
cd DMRs
./demo_DMRs.sh
```

II. Two test datasets are accompanied with the released Methy-Pipe software:

(1) dataset_light.tar.bz2:

This folder contains the raw data in fastq format, which can be used to test the Methy-Pipe. Since the depth of this dataset is very low (less than 5 fold on average), the DMR detection probably is not very accurate to offer biological significance. Nevertheless, it is a good example to illustrate how to use Methy-Pipe for the following purposes:

- trimming the raw reads with low-quality bases or sequencing adaptors.
- aligning the BS-seq reads.
- calculating the mappability and sequencing coverage.
- summarizing the results in a "summary.html" file.

(2) dataset_full.tar.bz2:

This dataset contains raw data in fastq format, which can be used to test Methy-Pipe as said in (1). In additional, users can perform the DMR mining.

III. The configuration file for the Methy-Pipe is illustrated in ***CONF***. It is easy to modify this standard ***CONF*** to analyze new dataset. The following is one example of CONF (Please modify the path in blue accordingly).

```
#FOMART: KEY<TAB>VALUE
# path to the statistics program R
R       /path-binary-R/R
#reference genome index for the BSAaligner
BS_INDEX       /path-to-BSAlignerIndex/hg19
#mismatch allowed for each end
MISMATCH 2
#minimal insert size allowed for paired-end reads
MIN_INS 0
#maximal insert size allowed for paired-end reads
MAX_INS 600
#each chromosome length
LIST_CHR_LEN   /path-to-BSAlignerIndex/hg19.size
#Watson strand reference (fasta)
GENOME_W_FA    /path-to-BSAlignerIndex/hg19.W.ori.fa
#Crick strand reference (fasta)
GENOME_C_FA    /path-to-BSAlignerIndex/hg19.C.ori.fa
#frequency for each 3mer in reference genome
HG_3MER /path-to-BSAlignerIndex/hg19.3mer
#windows around TSS (ucsc reference gene)
TSS     /path-to-BSAlignerIndex/TSS.win.bed
#sequencing data in fastq format
SEQ_FORMAT     fq
#prefix for each output result
OUT_PREFIX     test
#sequencing mode in a paired-end manner (PE) or single-end manner (SE)
SEQ_MODE       PE
#how many first cycles supposed to be used
#for example, 75 means the cycles after 75th would be omitted
USED_CYCLES    75
#how many threads supposed to be used for the BSAligner
THREAD 20
#whether to merge the all of alignments in this batch
#0 mean don't merge; 1 means merge
MERGE   0
#window size to profile the methylation density across the genome
#only the CpG sites are considered
BIN_SIZE_CPG   100e3
#window size to profile the methylation density across the genome
#only the CpG sites are considered
BIN_SIZE_NONCPG        100e3
#how many total cycles expected to be used (read1+read2).
SEQUENC_TOT_CYCLE      150
#a separated file recording the path of fastq files as well as the
#sample names to be analyzed (see Part IV)
INFO    ./info
```

IV. The ***info*** file is required for ***CONF***. It records the location of raw data as well as sample information.

```
#sample lane    description           path-to-read1.fq       [path-to-read2.fq
for paired-end reads]
PW396w  7       PW396w  /path-to/PW396w.read1.fq      /path-to/PW396w.read2.fq
CVS396  8       CVS396  /path-to/CVS396.read1.fq      /path-to/CVS396.read2.fq
```

V. If you need to perform DMR identification, you should first split the methylation call by each chromosome using the following commands.

```
#Please change to Methy-Pipe output directory,
#then type in the following commands:

../../methy-pipe2/utils/anno/split_call.sh \
       test.CVS396_8.W.call test.CVS396_8.C.rev.call CVS396_8_split CVS396_8
../../methy-pipe2/utils/anno/split_call.sh \
       test.PW396w_7.W.call test.PW396w_7.C.rev.call PW396_7_split PW396w_7

#or users can directly run the demo_split_call.sh in Methy-Pipe output
directory.
```

VI. You can use the following commands to identify DMRs:

```
#Please change to Methy-Pipe output directory,
#then type in the following commands:

mkdir DMRs
cd DMRs
../../../methy-pipe2/utils/DMR_calling/auto_DMR.biomarker.sh \
       ../CVS396_8_split ../PW396_7_split CVS396_refto_PW396w

#or users can directly run the demo_DMRs.sh in Methy-Pipe output directory.
```

VII. You can further annotate DMRs to closest genes using the following script in DMRs directory by:

```
perl ../../../methy-pipe2/utils/DMR_anno/dmr_anno.pl \
       ../../../methy-pipe2/bed_files/iGenome.hg19.revised.gff3 \
      all.hyper.call.filtered > all.hyper.call.filtered.anno2gene.xls
```

VIII. The methylation in any arbitrary region can be calculated by following script:

```
perl utils/regional_meth_density/calc_regional_met_density.pl \
      region.bed sample.W.CpG.call sample.C.rev.CpG.call > output
```

IX. Examples of Methy-Pipe output.

a. Alignment results (*.bsalign)

Read ID	Read	Quality	No. of hits	Read 1/2	Read length	Strand	Chr	Position	No. of mismatches	mismatch tracking	CIGAR string	alignment tracking	Watson /Crick
HWI-ST328:7:1101:12	AGAATTTATGTTA	BJPJ`Z^S`b	1	a	62	+	chr4	1.2E+08	0		62M	62	C
HWI-ST328:7:1101:12	AATAGTTTATGAT	fehdba^N)	1	b	62	-	chr4	1.2E+08	1	G->9T24	62M	9G52	C
HWI-ST328:7:1101:11	GGATATTGTATG	JJ\c`cSc^\	1	a	62	+	chr1	2.9E+07	0		62M	62	W
HWI-ST328:7:1101:11	GGATATTGTATG	^[a[fff^^X<	1	b	62	-	chr1	2.9E+07	0		62M	62	W

b. Methylation calling (*.call)

chr	Position	Base in reference	Total depth	Cytosine counts	Thymine counts	Sequence context
chr1	11310	c	1	0	1	c:a:g
chr1	11315	c	1	0	1	c:c:c
chr1	11316	c	1	0	1	c:c:t
chr1	11317	c	1	0	1	c:t:c
chr1	11319	c	1	0	1	c:t:t

c. Methylation density (*.density)

Chr	start	end	Cytosine counts	Thymine counts	Methylation density (%)
chr1	0	1000000	1008	556	64.5
chr1	1000000	2000000	4166	1459	74.1
chr1	2000000	3000000	2384	966	71.2
chr1	3000000	4000000	2101	502	80.7
chr1	4000000	5000000	2164	660	76.6
chr1	5000000	6000000	2143	453	82.6
chr1	6000000	7000000	3577	1045	77.4
chr1	7000000	8000000	2692	649	80.6

d. Regional methylation density calculation

Chr	Start	End	Description	Cytosine counts	Thymine counts	Methylation density
chr3	184375797	184376100	AluSx,SINE/Alu	7	1	87.5
chr7	29320335	29320416	MIRb,SINE/MIR	0	1	0.0
chr14	45211549	45211841	AluJb,SINE/Alu	6	1	85.7
chr12	28050879	28051088	MIR3,SINE/MIR	0	2	0.0
chr22	23351240	23351896	PABL_A,LTR/ERV1	3	1	75.0

e. DMR identification (DMRs/all.hypo.filtered)

				Test		Control		Methylation			CpG number	
Chr	Start	End	hypo/hyper	Cytosine counts	Thymine counts	Cytosine counts	Thymine counts	Test	Control	P-value	Test	Control
chr10	38816800	38818300	hypo	132	191	279	52	40.87	84.29	2.91E-08	30	25
chr10	42383000	42396900	hypo	33927	28610	46262	8388	54.25	84.65	0.00E+00	474	434
chr10	42596500	42598500	hypo	8944	6579	11626	1476	57.62	88.73	0.00E+00	61	57
chr10	42598700	42600700	hypo	13163	11847	18054	4509	52.63	80.02	0.00E+00	69	60
chr10	1.28E+08	1.28E+08	hypo	8	118	82	85	6.35	49.1	2.93E-09	21	29
chr14	19640700	19642900	hypo	62	434	195	92	12.5	67.94	7.36E-28	68	48
chr11	65779200	65779700	hyper	17	18	0	35	48.57	0	1.63E-04	7	7
chr9	66455500	66456000	hyper	20	33	0	41	37.74	0	2.23E-04	9	6

f. DMR annotation

				Test		Control		Methylation			CpG number		
Chr	Start	End	hypo/hyper	Cytosine counts	Thymine counts	Cytosine counts	Thymine counts	Test	Control	P-value	Test	Control	Gene associated regions
chr10	5093000	5093500	hypo	20	37	51	12	35.09	80.95	8.49E-03	6	5	AKR1C3:intron
chr10	5094500	5096400	hypo	182	111	322	39	62.12	89.2	2.88E-03	27	28	AKR1C3:intron
chr10	5117500	5118000	hypo	8	27	29	2	22.86	93.55	1.93E-03	5	5	AKR1C3:intron
chr10	5210900	5211400	hypo	18	47	59	9	27.69	86.76	2.71E-04	5	6	AKR1CL1:intron
chr10	5237400	5237900	hypo	29	39	52	3	42.65	94.55	6.41E-03	6	6	AKR1C4:promoter
chr10	5241000	5242000	hypo	62	94	221	36	39.74	85.99	9.31E-06	15	17	AKR1C4:intron
chr10	5405100	5408700	hypo	280	408	640	90	40.7	87.67	4.22E-18	70	66	UCN3:5UTR
chr10	5410400	5417300	hypo	596	640	1279	172	48.22	88.15	5.70E-22	126	125	UCN3:5UTR
chr10	5436800	5437800	hypo	146	125	276	19	53.87	93.56	2.89E-05	19	19	TUBAL3:CDS:3
chr10	5441300	5441800	hypo	9	27	53	9	25	85.48	2.32E-03	6	6	TUBAL3:intron
chr10	5486800	5487300	hypo	54	46	91	3	54	96.81	8.87E-03	7	5	NET1:promoter

X. If you have any other question, please contact jiangpeiyong@cuhk.edu.hk