UiO : Department of Informatics
University of Oslo

A Journey to the core

of man and machine alike

Reidar André Brenna

Master's Thesis Autumn 2015

A Journey to the core

Reidar André Brenna

August 18, 2015

Abstract

This thesis tries to determine if the Xeon Phi co-processor released in 2013 is a suitable hardware tool to perform DNA sequence alignments on. Starting off with an exploratory dive into the hardware itself before tackling some of available optimization techniques and previously implemented tools. As there are no tools currently on the market that utilizes the Xeon Phi hardware, the only comparisons made are to a regular Xeon processor to determine the outcome. Two programs were made in the course of this thesis, CRSbuild and DNAlign, that builds an FM index over the human genome and aligns sequences to the genome respectively. The results achieved in this thesis are rather disappointing and the thesis shows, to some degree, that native execution of an alignment tool of this magnitude is not suited. The shared memory architecture of the Xeon Phi posses a relatively small combined cache and with the lack of support for smaller data types this is a limitation that the four hardware thread and a 512 bit vector unit per core can not overcome.

Acknowledgments

I would like to thank my supervisors, Torbjørn Rognes and Johan Hjelmervik; without their input I would probably not have made it as far as I did. I will also give a huge thanks to Jorun Ramstad, you kept my motivation up when the work seemed to much and for her help with some of the figures

Finally I would like to thank my loving wife for her understanding and patience during this master thesis.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

1.1 Motivation

Aligning DNA Sequences to a genome is an important and time-consuming task in bioinformatics as it is the first step taken in order to analyze DNA samples. By aligning the sequences to a genome, one may gain several important clues about the health of the donor.
In January 2013, Intel released the Xeon Phi co-processor as a serious contender in the high performance programming field. It is designed to tackle highly parallel problems, and is a promising hardware to implement a sequence alignment tool on. However promising it may be, the challenge to find a suitable technique to take advantage of the unique hardware is highly present as there are no other program that has yet to take advantage of the Xeon Phi when mapping DNA sequences. An examination of the co-processor itself, as well as some of the tools implemented for other kinds of hardware is needed in order to attempt this task.

1.2 Goal

The Goal of this thesis is to determine whether the new Intel Xeon Phi co-processor is suitable to perform DNA sequence alignment, and in the process attempt to implement one such program.

Chapter 2

Background

2.1 Xeon Phi

The Intel Xeon Phi co-processor, although relatively new compared to traditional GPUs, have made a big impact on the TOP500 list of supercomputers and are gaining traction in HPC-systems rapidly <CITE top500?>. The co-processor contain roughly 60 cores clocked at 1GhZ each with a shared memory area and a 512-bit wide vector unit, located on a PCI board for easy assembly and is the chosen hardware for this thesis. The co-processors will be described in short in the following subsections as stated in its Best practice guide[2] and the book *Intel Xeon Phi Coprocessor High Performance Programming*[8]

2.1.1 Specifications

Overview

The co-processor, spotting a full service Linux operating system with support for many of Intels' own development tools in addition to OpenMP, C/C++ and MPI, is designed to work, alone or several together, along side the Intel Xeon processor, connected to the host through the PCI express bus. The operative system, designed for a Many Integrated Core architecture (MIC), allows a user to execute code directly on the co-processor in addition to offloading whole, or parts of, programs from the host CPU. The cores shares the same fundamentals as the original Pentium core and are thusly in-order dual issued x86 cores. Consisting of two processing units; the scalar unit and the expanded vector unit spotting a whole new vector instruction set and the 512-bit wide vector unit. Each core is able to fetch and decode instructions from up to four hardware threads and, due to the double processing unit, execute two of them per cycle. Each core, as shown in figure 2.1, is connected by a high performance on-die bidirectional ring interconnect, the Core Ring Interface (CRI), and has equal access to a shared memory and all I/O devices connected to the host computer. The co-processor also comes with several auto-vectorization possibilities enabled by default, and several pragma compiler directives may be used for further customization of Xeon Phi code.

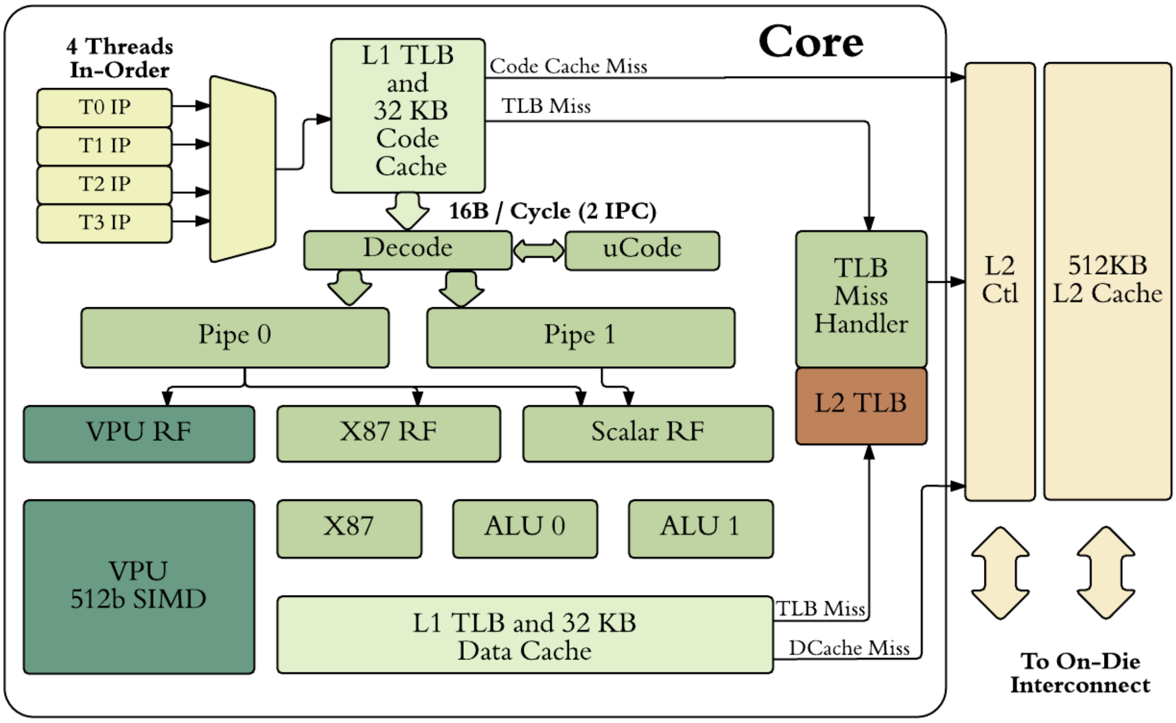


Figure 2.1: Architecture of a single Xeon Phi core drawn from the figure in *Intel Xeon Phi Coprocessor High Performance Programming*[8].

A simplified version of the architecture is depicted in figure 2.2 with only 4 cores visible on the CRI together with the L2 cache, memory ports and tag directory (TD) loosly drawn from *Intel Xeon Phi Coprocessor High Performance Programming*[8].

VPU

The Vector Processing Units (VPU) new vector instruction set provide a plethora of new vector intrinsics to the user, making it possible to vectorize algorithms such as Smith Waterman. As a rule of thumb when utilizing such instructions, regarding execution time, is that they all have a latency to throughput ration of 4 to 1 cycles. As the vector unit requires a 64.byte alignment of its content, non of Intel's previous vector intrinsics are supported, but it comes bundled with an extended math unit (EMU), making it possible to execute 16 single-precision or 8 double-precision operations simultaneously.

Memory

There are eight memory controllers on a Xeon Phi, supporting up to 16 GDDR5 channels. A transfer speed of up to 5.5 GT/s, with each transfer being 4 bytes of data, and a theoretical aggregated bandwidth ranging from 240 to 352 GB/s is provided for fast memory access. The cores access the

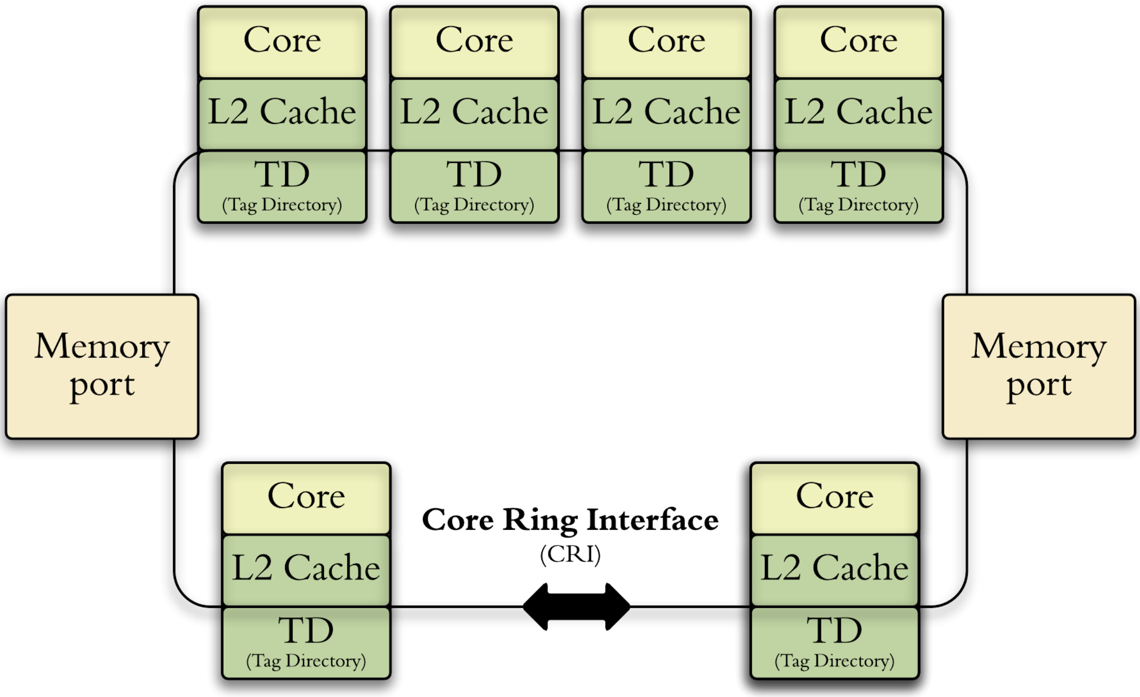


Figure 2.2: Simple architecture sketch based on figure in *Intel Xeon Phi Coprocessor High Performance Programming*.[8]

memory through the CRI with hook memory controllers located on the die and by linking the memory to the ring a smoother memory operation is achieved when all cores are utilized.

Cache

Each core posses a L1 cache consisting of a 32 KB instruction cache and 32 KB data cache with a cache lines of size 64 byte exclusively accessible only to the core itself. The L1 load-to-use latency is 1 cycle, e.g. an integer value loaded can be used on the next cycle by an integer instruction. (Vector instructions operates with a different latency). The cores also contribute 512 KB to the shared global L2 cache storage and if no core shares any code or data the effective size of the L2 cache storage is up to a total of 31 MB. However, as the L2 cache size is highly dependant on the degree of shared code and data among the cores, the size could also be, if every core share the same code and data, closer to 512 KB. Similar to the L1 cache, L2 cache also have a 64 byte cache line but the cycle latency of the L2 cache is higher with its clock cycle latency of 11 cycles. L2 cache also supports ECC correction and come with a streaming hardware prefetcher.

2.1.2 Xeon Phi Software Development

As mentioned in 2.1.1 Overview, the Xeon Phi allows the programmer to develop code that could be executed natively without the need of host CPU involvement in addition to more traditional offloading paradigms.

The first, native programming, are developed to run on the Xeon Phi without involvement from the CPU. These applications need to be highly parallelized with little to none serial dependency to utilize the thread possibilities of the co-processor without having parts of the execution waiting for other threads to be completed. It should also utilize the vector unit to the fullest as this is one of the co-processors biggest strengths, if not the biggest. A Native program has the added benefit of removing the overhead associated with data transferal from host CPU to processing unit. On the other hand, a native program should refrain from overuse of I/O calls as this is significantly slower on Xeon Phi compared to the Xeon CPU in addition to a limited amount of accessible memory.

The second, the offload model, comes in several intensities. An application that have distinct parts of highly parallelize -and vectorize-able code could benefit greatly from offloading those parts to the co processor if the overhead associated with the transferal could be kept as small as possible. Intel has also provided in addition to communication protocols such as OpenMP and MPI compiler directives to allow a software developer to pass information between host CPU and co-processor seamlessly throughout the code with options to both chose whether the co-processor should allocate and or free memory when called or to reuse memory already present in its shared memory.

Optimization

When developing an application that executes, wholly or partially, on the Xeon Phi co-processor there are a few important aspects to take into consideration in order to achieve optimal performance. The book *"Intel Xeon Phi Co-processor High Performance Programming"*[8], written by Jeffers and Reinders, state that there are three important considerations to take into consideration to gain the desired high performance the co-processor is capable of deliver: Memory usage, Scaling and Vectorization.

> *"Trying to use Intel Xeon Phi co-processor without having maximized the use of parallelism on the Intel Xeon processor will almost certainly be a disappointment"*

If the memory usage of the applications is less then the maximum available on a Xeon Processor, you would probably not gain to much by changing to the co-processor.

High scaling on multiple cores is advisable for any applications that intends to utilize the co-processor to the fullest. This may be easily gaged for an application by allowing the application to run several times with increasingly more threads allowed. By tracking the thread affinity we'll be able to detect any significant changes in performance and by some small modifications one may be able to increase performance with more cores.

The last consideration is the ability to vectorize properly. By executing the application with and without vectorization, the execution without should have a significant decrease in performance as the co-processor does its best work when as many cores as possible execute a vector operation every cycle.[8]

As stated above, a co-processor core is clocked at only 1GhZ and are thusly slower than a Xeon core without the luxury of branch predictions and out-of-order execution. To fully utilize each clock cycle to its max capabilities precautions must be made to avoid unnecessary operations. Due to the fact that all memory read and writes on the co-processor are performed on 64-bit aligned data, unaligned data would slow down the execution drastically since the compiler would have to realign the data each time the code requires it. In some cases where the compiler is unable to catch unaligned data the application could in worst case terminate itself prematurely due to memory access errors. Too combat this, it's advisable to use Intels' provided aligned allocation functions _mm_malloc and _mm_free instead of regualar malloc and free. This would ensure that the data is 63-bit aligned and handleable by the co-processor. To further increase performance one could in addition to the aligned allocation explicitly tell the compiler that arrays used by the program is aligned with #pragma vector aligned to prevent the compiler to include alignment checks on already aligned data structures.
Another useful tip is to actively try to arrange data in such a way that data used in succession of each other lies after one another in memory. This could significantly reduce cache misses and unnecessary cache reads since the L2 cache is both slow and relatively small as previously mentioned. This comes into play when you execute a highly threaded application running heavy vector operations.
Following is an extracted checklist for Xeon Phi development:

- 64-bit aligned memory
- Memory arrangement (cache)
- Vectorization
- Good usage of compiler directives.

2.2 DNA sequence mapping

The process of mapping or assembling DNA sequences into a genome is a vital addition to DNA sequence technology as the sequencers themselves are unable to process the entire genome in one go. Varying on the sequencer used, genome piece size processed have a size ranging from only 20 bases up to 30000 bases. However, the sequencing of a genome is not without problems and pitfalls should one not take proper precautions. *"The problem* *of sequence assembly can be compared to taking many copies of a book, passing each of them through a shredder with a different cutter, and piecing the text of the book back together just by looking at the shredded pieces."* Other than the high difficulty of the task, staying with the book analogy, there are some additional problems that complicates this work significantly. The original book could have several repeated paragraphs and some of the shredded pieces may have been modified after the fact, essentially giving the text typos that were not present before shredding. The container of the shredder are not guaranteed to be emptied between each shredding resulting in the possibility of pieces from another book being mixed in with the other shreds or a shredded pieces may have been mangled beyond recognition.[19]

There are two main branches when one should align sequences: de-novo assembly and mapping assembly, one trying to match the pieces to each other while the other relying on a reference to match the pieces to respectively. De-novo assembly pieces each shred together, matching the ends to each other in order to find the best fit, until the book is finished. By doing it this way there is no way of knowing what kind of book we end up with, it being a scientific book or a cartoon. Mapping assembly on the other hand searches the entire reference to find the best suitable location for each shredded piece, tossing away any shred that does not match the reference.

2.2.1 Illumina

The most commonly used sequencing technology today is provided by Illumina [7]. The technology used is based on the two Cambridge scientists Shankar Balasubramanian, Ph.D. and David Klenerman, Ph.D.. In the late 90's they formed the company Solexa and ultimately created the Solexa sequencer in 2006, able to sequence 1 gigabase in one run. Solexa were acquired in 2007 by Illumina which have taken the sequencer to new hights. Their next generation sequencing data output have surpassed more than a doubling each year and the current Illumina sequencer are able to generate more than a terrabase of data each run.

2.2.2 File format

Output

When aligning reads during a DNA sequence mapping process, it is preferable that the final product conforms to some standard in order for other programs, e.g. tools that analyze the output, to extract the content. Heng Li created a format called Sequence Alignment/Map (SAM) which is *"a generic format for storing large nucleotide sequence alignments"* [6]. Aiming to be flexible enough to handle all of the alignment information created by various alignment tools and simple enough to generate by these tools or converted into from tools using other formats. SAM also provide the possibility to allow most operations to be performed on the alignment as a stream of data instead of having to load the whole alignment into memory.

It also allows the file to be indexed by genomic position to retrieve all reads aligned to a locus while boasting a compact file size that remains human readable [5]. There are two main components to SAM formated output. The header section and the alignment section.

The header section of the output is not required, but must be located at the beginning of the output file if present. This thesis will not provide any header output and thusly the header information given in table 2.1 is only for informational purposes. The alignment section, on the other hand, is mandatory and contains information about one or more alignments separated by a newline. Each individual alignment information is tab separated just like the information in each header line. As the alignment section contains 11 mandatory fields, as presented in table 2.2. This thesis does not prioritize to provide all the required information in the mandatory fields as the scope of the thesis is to gage whether a Xeon Phi is suitable to perform alignments and there are far more hardware strenuous activities that will weigh far heavier on the final decision. Thusly, several of the default values will be utilized where possible without loosing the alignment score and location itself. For information regarding the individual fields see table 2.3. In addition to the 11 mandatory fields, there are several optional additions to the alignment line if one wish to utilize them. This thesis takes advantage of one of these additions in order to replace the MAPQ field with its default value and use another score representation. The additional filed used in this thesis is "AS:i:". Broken down, AS states that the following value is the aligners own alignment score and i states that the value is an integer.

There is a trade off due to the human readability of the SAM file, it being more strenuous to work with compared to BAM files, the binary equivalent to SAM. SAMtools is an open source tool made to make life easier for humans and computers alike, also created by Heng Li and freely downloadable from github[6]. It allows a user to freely convert SAM to BAM or vice versa.

Sequence Input

FASTQ[17] *"has recently become the* de facto *standard for storing the output of high throughput sequencing instruments such as the Illumina Genome Analyzer."* The FASTQ format, originally intended for to be a bundle for the preexisting FASTA format, usually contain four line of information for each sequence:

- the sequence identifier, preceded by a '@' character.
- the raw sequence letters
- the third line is preceded by a '+' character and is optionally followed by the same sequence identifier and any additional description
- the last line contains the quality value for the sequence, in line two, and must have the same amount of characters as the sequence

@HD	Begins the header line, if present		
	VN*	-	Format version accepted format: / ^ [0-9] + . [0-9] + $ /
	SO	-	Sorting order of alignments. Valid values: *unknown*(default), *unsorted*, *queryname* and *coordinate*
	GO	-	Grouping of alignments. Indicating that similar alignments are grouped together
@SQ	Reference sequence dictionary		
	SN*	-	Reference sequence name Unique for each present @SQ line.
	LN*	-	Reference sequence length Range: $[1, 2^{31} - 1]$
	AS	-	Genome assembly identifier
	M5	-	MD5 checksum
	SP	-	Species
	UR	-	Sequence url eg. http: or ftp:. If not, file-system path assumed.
@RG	Read group. Unordered multiple RG lines allowed.		
	ID*	-	Read group identifier Unique for each present @RG line.
	CN	-	Name of sequence center producing the read
	DS	-	Description
	DT	-	Date the run was produced
	FO	-	Flow order
	KS	-	Nucleotide array corresponding to key sequence of each read
	LB	-	Library
	PG	-	Program used for processing
	PI	-	Predicted median insert size
	PL	-	Platform/Technology used
	PM	-	Platform model
	PU	-	Platform unit
	SM	-	Sample (Pool name for pool sequencing)
@PG	Program		
	ID*	-	Program record identifier Unique for each present @RG line.
	PN	-	Program name
	CL	-	Command line
	PP	-	Previous @PG-ID
	DS	-	Description
	VN	-	Program version
@CO	One-line text comment. Unordered multiple CO lines allowed.		

Table 2.1: SAM file format: header section

The quality value characters, sorted with increasing quality:

```
!"#$%&'()*+,-./0123456789:;<=>?@
```

Col	Field	Type	Regexp/Range	Description
1	QNAME	String	[!-?A-],{1,255}	Query template name
2	FLAG	Int	$[0, 2^{16} - 1]$	Bitwise flag
3	RNAME	String	| [!-()+-<>-][!-]*	Reference sequence name
4	POS	Int	$[0, 2^{31} - 1]$	1-based leftmost mapping position
5	MAPQ	Int	$[0, 2^{8} - 1]$	Mapping quality
6	CIGAR	String	| ([0-9]+[MIDNSHPX=])+	CIGAR string
7	RNEXT	String	| = | [!-()+-<>-][!-]*	Ref. name of next/mate read.
8	PNEXT	Int	$[0, 2^{31} - 1]$	Position of next/mate read
9	TLEN	Int	$[-2^{31}, 2^{31} - 1]$	Observed template length
10	SEQ	String	| [A-Za-z=.]+	Segment sequence
11	QUAL	String	[!-]+	ASCII of ASCII of Phred-scaled base quality+33

Table 2.2: SAM file format: alignment section - overview

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_‘
abcdefghijklmnopqrstuvwxyz{|}~
```

An example FASTQ file containing one sequence with the Illumina sequence identifiers is presented in figure 2.3.

Figure 2.3: FASTQ with Illumina sequence identifier

```
@HWUSI-EAS100R:6:73:941:1973#0/1
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!’’*((((***+))%%%++)(%%%%).1***-+*’’))**55CCF>>>>>>CCCCCCC65
```

The Illumina sequence identifier breakdown:

- HWUSI-EAS100R - the unique instrument name
- 6 - flowcell lane
- 73 - tile number within the flowcell lane
- 941 - ’x’-coordinate of the cluster within the tile
- 1973 - ’y’-coordinate of the cluster within the tile
- #0 - index number for a multiplexed sample (0 for no indexing)
- /1 - the member of a pair, /1 or /2 (paired-end or mate-pair reads only)

QNAME	Identical names are regarded as originating from the same template
FLAG	0x1 - template have multiple segments in sequencing 0x2 - all segments properly aligned (according to aligner) 0x4 - segment unmapped 0x80 - next segment unmapped 0x10 - SEQ being reverse-complimented 0x20 - next SEQ being reverse-complimented 0x40 - first segment in template 0x80 - last segment in template 0x100 - secondary alignment 0x200 - not passing quality controls 0x400 - PCR or optical duplicate 0x800 - supplementary alignment FLAG & 0x900 = 0 - primary line of read only one read per SAM file should satisfy this
RNAME	Reference sequence NAME of the alignment. For unmapped segments, set to *
POS	The first base in a reference sequence has coordinate 1. Set as 0 for an unmapped read. If POS is 0, no assumptions can be made about RNAME and CIGAR.
MAPQ	Mapping Quality, equals $10\log_{10}\mathrm{Pr}\{\text{mapping position is wrong}\}$, rounded to the nearest integer. A value 255 indicates that the mapping quality is not available.
CIGAR	Mapping string of the same length as the Query. Op - Description M - Alignment match (match or mismatch) I - Insertion to the reference D - Deletion from the reference N - Skipped region from the reference S - Soft clipping (clipped sequences present in SEQ) May only have H between itself and closest end of CIGAR string H - Hard clipping (clipped sequences not present in SEQ) May only be at either end of the CIGAR string P - padding (silent deletion from padded reference) = - sequence match X - sequence mismatch
RNEXT	RNAME of the next read in template. Set to * when information is unavailable
PNEXT	POS of the next read in template. Set to 0 when information is unavailable
TLEN	signed observed Template length. Set as 0 for single-segment template or when the information is unavailable.
SEQ	Segment sequence, may be * is segment is not stored.
QUAL	ASCII representation of MAPQ, set to * if unavailable.

Table 2.3: SAM file format: alignment section - detailed

Genome Input

Even though the FASTQ format would be suited to describe a reference genome, the extra information provided by the two last lines is unnecessary as the genome does not require the extra information to be useful. Thusly

the FASTA format, which FASTQ extends, that retains the two first lines of an entry without the optional second retail of the sequence identifier or the that it extends is suitable for describing a reference genome.
In addition, FASTA is not as strictly line bound as its extender and is divided into two parts; The first line, preceeded with the greater than character '>', describing what the entry contains, such as the Chromosome name and number, and the rest of the entry containing the sequence itself over multiple lines if required.

```
>gi|31563518|ref|NP_852610.1| microtubule-associated proteins 1A/1B light.....
MKMRFFSSPCGKAAVDPADRCKEVQQIRDQHPSKIPVIIERYKGEKQLPVLDKTKFLVPDHVNMSELVKI
IRRRLQLNPTQAFFLLVNQHSMVSVSTPIADIYEQEKDEDGFLYMVYASQETFGFIRENE
```

2.3 Optimization approaches

2.3.1 Dynamic Programming

Dynamic programing involves splitting a problem into smaller and more manageable problems, solving those and then using the answers gained to answer the main problem. Keep in mind that if a problem is to be solved by dynamic programming, the subproblems needs to be similar to each other or else the problem would be more suitable to solve with a method called divide and conquer.
One of the most famous algorithms, utilizing dynamic programming, in bioinformatics is the Smith-Waterman One of the famous dynamic programming solutions in bioinformatic is the Smith–Waterman (SW) [15] algorithm, first proposed by Temple F. Smith and Michael S. Waterman in 1981, that performs local sequence alignment.

2.3.2 Smith-Waterman

Instead of looking at the total sequence, the Smith–Waterman algorithm compares segments of all possible lengths and optimizes the similarity measure to determine similar regions between two strings. Smith-Waterman is decended from the Needleman–Wunsch(NW) algorithm[13], one of the earliest dynamic programming algorithms utilized in bioinformatics. Being a dynamic programming algorithm, it provides the highly sought after property, a guarantee to find the optimal local alignment based on a given scoring system. The scoring systems usually consist of a substitution matrix and a gap-scoring scheme to properly allow SW to find the optimal alignment. The main difference between SW and NW is that NW focuses on global alignment, and is thus not highly suited in bioinformatics, and operates with zero as the minimum value of any given cell, not allowing negative cell values in its calculation matrix. This allows the local alignments to become visible in preference to a global one. Smith-Waterman have a complexity of $O(m^2n)$, which later have been improved significantly by Gotoh [4] to run at just $O(mn)$ by adding a gap extention test, increasing the gap if found to be true, rather than testing for each possible gap length every time. A more detailed description of the changed made by Gotoh can be found in section 2.3.2.

Calculating Smith-Waterman is a two part problem; first a scoring matrix (H) is calculated based on the algorithm seen in figure 2.4, the scoring system (s), and previously calculated cell values, and secondly a traversal of the calculated matrix, starting at the highest scoring cell and backtracking until a 0 is found. The calculation matrix is initialized to zero, and any negative value is set to 0 as well since SW does not allow negative values as stated previously.

Figure 2.4: Smith-Waterman algorithm

$$H(i,0) = 0,\ 0 \leq i \leq m.$$

$$H(0,j) = 0,\ 0 \leq j \leq n.$$

$$H(i,j) = \max \left\{ \begin{array}{cc} 0 & \\ H(i-1,j-1) +\ s(a_i,b_j) & \text{Match/Mismatch} \\ \max_{k\geq 1}\{H(i-k,j) +\ W_k\} & \text{Deletion} \\ \max_{l\geq 1}\{H(i,j-l) +\ W_l\} & \text{Insertion} \end{array} \right\},$$

$$\texttt{Where}\ \ 1 \leq i \leq m\ \ \texttt{and}\ \ 1 \leq j \leq n.$$

An example that descries SW from beginning to end, taken from Wikipedia [20], are depicted in below.

Smith-Waterman example

For this example, the following sequences will be used as reference and query respectively:

$$\begin{array}{cccccccc} \text{A} & \text{C} & \text{A} & \text{C} & \text{A} & \text{C} & \text{T} & \text{A} \\ \text{A} & \text{G} & \text{C} & \text{A} & \text{C} & \text{A} & \text{C} & \text{A} \end{array}$$

A simple scoring system will be used:
$s(a,b) = +2$ if $a = b$ (match), -1 if $a \neq b$ (mismatch).

The resulting calculation matrix are then:

$$H = \left(\begin{array}{cccccccccc}
 & - & A & C & A & C & A & C & T & A \\
- & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
A & 0 & \mathbf{2} & 1 & 2 & 1 & 2 & 1 & 0 & 2 \\
G & 0 & \mathbf{1} & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
C & 0 & 0 & \mathbf{3} & 2 & 3 & 2 & 3 & 2 & 1 \\
A & 0 & 2 & 2 & \mathbf{5} & 4 & 5 & 4 & 3 & 4 \\
C & 0 & 1 & 4 & 4 & \mathbf{7} & 6 & 7 & 6 & 5 \\
A & 0 & 2 & 3 & 6 & 6 & \mathbf{9} & 8 & 7 & 8 \\
C & 0 & 1 & 4 & 5 & 8 & 8 & \mathbf{11} & \mathbf{10} & 9 \\
A & 0 & 2 & 3 & 6 & 7 & 10 & 10 & 10 & \mathbf{12}
\end{array}\right).$$

The local alignment is found by going backward from the highest score, always following the highest cell value above, to the left, or diagonally up to the left until all 3 are 0.

To better depict this backtracking path, a new matrix is shown below with the values all changed to arrows to indicate the direction.

$$T = \left(\begin{array}{cccccccccc}
 & - & A & C & A & C & A & C & T & A \\
- & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
A & 0 & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\
G & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \nwarrow & \uparrow \\
C & 0 & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow \\
A & 0 & \nwarrow & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\
C & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow \\
A & 0 & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\
C & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \leftarrow \\
A & 0 & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \nwarrow
\end{array}\right).$$

The backtracking results in the following alignment:

Database sequence:	A	-	C	A	C	A	C	T	A
Query sequence:	A	G	C	A	C	A	C	-	A

Gotohs modification of SW

As stated in section 2.3.2, Gotohs[4] modifications consisted of changing the way gap penalties were calculated in SW. The new function set is presented in figure 2.5

By adding E and F matrices of the same size as H, keeping track of the score for the corresponding cells in H as if a gap in the query or database were found instead. Q represents a gap opening and a single extension, while R is the gap extension penalty by itself. s is the same here as in the Smith-Waterman algorithm, see 2.4 and score always keep track of the value in the highest scoring cell.

Figure 2.5: Gotohs improved Smith-Waterman algorithm

$$H_{i,j} = \begin{cases} \max \begin{cases} H_{i-1,j-1} + SM[q_i, d_j] \\ E_{i,j} \\ F_{i,j} \\ 0 \end{cases} & \left| \begin{array}{c} i > 0 \\ \cap \\ j > 0 \end{array} \right. \\ 0 & \left| \begin{array}{c} i = 0 \\ \cup \\ j = 0 \end{array} \right. \end{cases}$$

$$E_{i,j} = \begin{cases} \max \begin{cases} H_{i,j-1} - Q \\ E_{i,j-1} - R \end{cases} & \left| j > 0 \right. \\ 0 & |j = 0 \end{cases}$$

$$F_{i,j} = \begin{cases} \max \begin{cases} H_{i-1,j} - Q \\ F_{i-1,j} - R \end{cases} & \left| i > 0 \right. \\ 0 & |i = 0 \end{cases}$$

$$score = \max_{1 \leq i \leq m \cap 1 \leq j \leq n} H_{i,j}$$

2.3.3 Parallelization

A widely used optimization technique is parallelization, a technique that is based upon simultaneous execution of non dependant code. There are several ways to achieve parallel performance; either by utilizing Single Instruction Multiple Data (SIMD) vector operations, OpenMP and MPI, and cluster computing, among others. A short summary on some of these techniques are presented in the following subsections.
There are also specialized computer architectures that may be utilized to gain an even higher level of parallelization, such as the Intel Xeon Phi where several cores take advantages of a shared memory pool and L2 cache.

According to Amdahl's law[1], see figure 2.6, the possible maximum speed gain achievable by a program computing in parallel is highly limited by the time needed to perform the sequential fractions of the program, and have a ceiling of $\frac{1}{B}$ as demonstrated by the example below, when n goes towards infinity.

Amdahl's law example

For a given program that is able to parallel compute 90% of its calculations, $1 - B = .90$, the maximum speed gained, according to Amdahl's law, will be:

$$S(n) = \frac{1}{.10 + \frac{.90}{n}}$$

Calculating the runtime improvement of the program with varying amounts of threads we get the following speed improvement factors:

$$\begin{array}{rcccccccc}
S(n=1) & = & \frac{1}{.10+\frac{.90}{1}} & = & \frac{1}{.10+.90} & = & \frac{1}{1} & = & 1 \\
S(n=2) & = & \frac{1}{.10+\frac{.90}{2}} & = & \frac{1}{.10+.45} & = & \frac{1}{.55} & \approx & 1.81 \\
S(n=5) & = & \frac{1}{.10+\frac{.90}{5}} & = & \frac{1}{.10+.18} & = & \frac{1}{.28} & \approx & 3.57 \\
S(n=6) & = & \frac{1}{.10+\frac{.90}{6}} & = & \frac{1}{.10+.15} & = & \frac{1}{.25} & = & 4 \\
S(n \to \infty) & = & \frac{1}{.10+\frac{.90}{n \to \infty}} & = & \frac{1}{.10+0} & = & \frac{1}{.1} & = & 10
\end{array}$$

According to the calculations above we can see that a program that executes 90% of its calculations in parallel, regardless of threads, will never achieve a greater speed increase than 10 times single thread execution. More importantly however, the speed gain is less and less effective for each thread added as seen when comparing going from 1 to 2 threads as opposed to 5 to 6 threads.

Thread achieved parallelization

Thread level parallelization achieved by executing parts, or the entirety, of the calculations simultaneously in multiple threads as depicted in figure 2.7. Libraries like OpenMP and MPI allows a programmer to chose how and what parts of the code that should be executed in parallel. OpenMP spawns new threads, from the initial program process during runtime, running parts of the calculations before merging back into one thread again. This may occur as many times as the programmer wants during the execution as dictated by compiler directives given to the compiler. The amount of threads used in each parallelized section of the code may either be hardcoded before compilation, set by environmental flags or a combination of the two. OpenMP, On other hand, works in a slightly different way by spawning multiple threads from the beginning running a single program process each. Providing a shared memory to all of its process threads. When utilizing threads its important to consider thread synchronization to avoid race conditions and unpredicted results.

Vectorization

Vectorization, another method of doing several calculations at once, is to utilize vector operations by bundling multiple, equal, calculations together and performing them as one. There are mainly two ways for a programmer

Figure 2.6: Amdahl's Law

Given

$n \in \mathbb{N}$	the number of threads
$B \in [0, 1]$	the fraction of the algorithm that is strictly serial

The time $T(n)$ an algorithm takes to finish when executed with n thread(s) corresponds to:

$$T(n) = T(1)\left(B + \frac{1}{n}(1 - B)\right)$$

Therefore, the theoretical speedup $S(n)$ of executing a given algorithm on a system capable of executing n threads is:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1-B)\right)} = \frac{1}{B + \frac{1}{n}(1-B)}$$

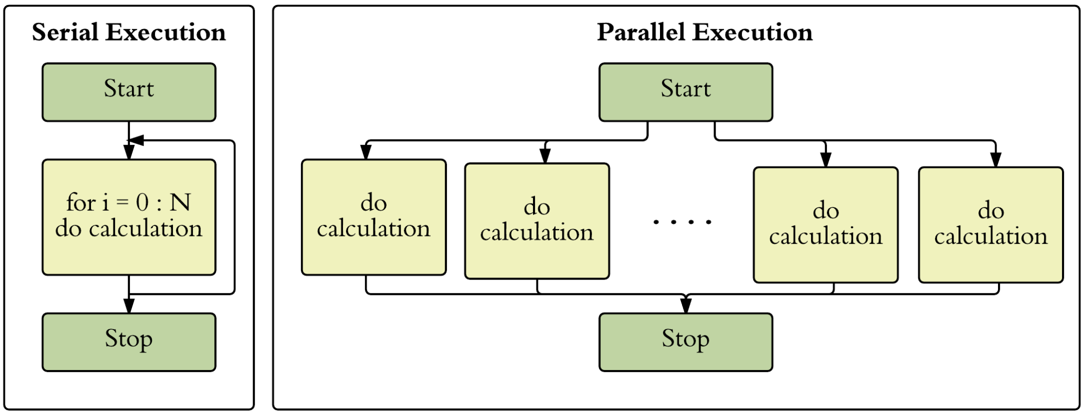
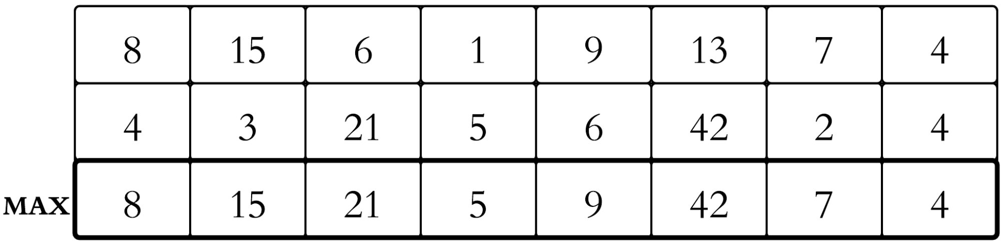


Figure 2.7: Single vs multiple threaded execution

to facilitate vector operations; intrinsics and assembly programing. In C, assembly programing may either be small parts of assembly code snippets embedded in to the C source code, inline assembly, or writing the assembly code in a different area and then making said code available as an external feature. All though, possible to implement as assembly, it is advisable to utilize provided intrinsics which makes the programmers job much simpler. An interactive intrinsic list, provided by Intel, can be found here: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ Almost all of the intrinsics take some sort of vector as both input and output. A vector example is depicted in figure 2.8.



	8	15	6	1	9	13	7	4
	4	3	21	5	6	42	2	4
MAX	8	15	21	5	9	42	7	4

Figure 2.8: Vectorized calculations

Multiple processing units

Without going to much into depth, by using multiple processing units inside a single computer or several computers clustered together a program may divide its workload onto different processing units and thusly perform several operations at the same time. It is noteworthy to mention that an overhead in runtime needs to be taken into consideration, addressing the additional time required to transfer the data from one processing unit to another.

2.4 Tools and Algorithms

There are several methods of aligning DNA sequences on the market today but, as stated previously, non are utilizing the Xeon Phi co-processor to do so. This thesis will focus on one particular approach that will be described as follows together with some of the existing tools and methods already on the market.

2.4.1 BWT

Burrow-Wheeler Transformation[3] is based on a transformation Wheeler discovered, but never published, 11 years prior to the publication of BWT in 1994. It is simple in design and transform a string into a much more compress-able format. Regular compression techniques work by calculating multiple occurring patterns and then encoding those patterns more compactly. What BWT does is that it rearranges a string into patterns of similar characters. This rearrangement of the string provide a much better starting point for a compression but most importantly is that it does not require an additional "codex" to translate the string back into its original arrangement. First of all it creates a table of all of the possible ways of writing the string by bit shifting one space and appending the character the "fell off" on the opposite side of the string. Both the start of and end of character should be included. After the table has been created, its sorted alphabetically. The return string after BWT consist of the last character of each sorted entry in the table. An example can bee seen in table 2.4 where the input sequence is |ANATTACGATCAT$ and the resulting BWT string is T|GNCTACATAAA$T. The example uses | and $ to indicate sequence begining and end respectively[16].

input	rotation table	sorted table	output
|	|ANATTACGATCAT$	ACGATCAT$|ANATT	T
A	ANATTACGATCAT$|	ANATTACGATCAT$|	|
N	NATTACGATCAT$|A	ATCAT$|ANATTACG	G
A	ATTACGATCAT$|AN	ATTACGATCAT$|AN	N
T	TTACGATCAT$|ANA	AT$|ANATTACGATC	C
T	TACGATCAT$|ANAT	CAT$|ANATTACGAT	T
A	ACGATCAT$|ANATT	CGATCAT$|ANATTA	A
C	CGATCAT$|ANATTA	GATCAT$|ANATTAC	C
G	GATCAT$|ANATTAC	NATTACGATCAT$|A	A
A	ATCAT$|ANATTACG	TACGATCAT$|ANAT	T
T	TCAT$|ANATTACGA	TCAT$|ANATTACGA	A
C	CAT$|ANATTACGAT	TTACGATCAT$|ANA	A
A	AT$|ANATTACGATC	T$|ANATTACGATCA	A
T	T$|ANATTACGATCA	|ANATTACGATCAT$	$
$	$|ANATTACGATCAT	$|ANATTACGATCAT	T

Table 2.4: Burrow-Wheeler Transformation matrix.

2.4.2 FM index

Full-text index in minute space is a variation of BWT and utilizes the fact that the sorted table from BWT can be thought of as a suffix table for all the suffixes of the string. The first column of the sorted table is also similar to suffix arrays and this relation is the main drive force behind the FM index.[18] This relation is ideal when trying to find out whether a string is a substring of another. Each first character in a string in the sorted table, and rotation table for that matter, is preceded by the last char in the same string if we consider the input string as a ring buffer. In addition, the characters are strictly unique in the sense that each character retains its original position in the input string.

The link between characters

Considering the sorted table in table 2.4 and specifically the 4 strings starting with T. From the sorted table we can tell that there are 3 separate substrings in the input string that contains AT and one that contains separate substrings in the input string that contains AT and one that contains TT. More precisely the location of these preceding characters in the first row of the table is given by traversing the last row and counting occurrences of the character in question. For instance, if we are searching for a substring containing AT we know there are three of them and they refer to the 3rd.

4th. and 5th row starting with an A.

When searching for a specific substring one only need to look at on char in the query sequence at a time starting from the back of the query sequence and working our way to the front. If one is able to follow this simple search pattern all the way to the beginning, FM index guarantees that the query is a substring of the indexed string. An example of this search process is described below.

Searching for a CAT

By taking the first and last column from the sorted table in table 2.4 the smaller FM index table is constructed as illustrated below:

A	T
A	|
A	G
A	N
A	C
C	T
C	A
G	C
N	A
T	T
T	A
T	A
T	A
|	$
$	T

When searching a FM index there are two actions that are done; the initial location of the last char in the query sequence and the iterative steps following to locate the preceding character in the query sequence.

In this example, the query sequence CAT are to be found and the first action is thusly to locate all strings beginning with the letter T. Due to the alphabetical sorting done to create the sorted table this is the easiest task to perform:

A	T
A	|
A	G
A	N
A	C
C	T
C	A
G	C
N	A
T	T
T	A
T	A
T	A
|	$
$	T

The two next steps are to find A and C in the last column sequentially and if found the coresponding location in the first column. As mentioned above, a T may be precedded by three different A's, the 3rd. 4th. and 5th A in the first column:

A	T
A	|
A	G
A	N
A	C
C	T
C	A
G	C
N	A
T	T
T	A
T	A
T	A
|	$
$	T

And finally we check whether any of these three A's are being preceded by a C, which it is. Out of the three possible AT substrings, only one of which are preceded by a C and that C is also the first of the two C's located in the first column.

A	T
A	|
A	G
A	N
A	C
C	T
C	A
G	C
N	A
T	T
T	A
T	A
T	A
|	$
$	T

After completing the query, it is guaranteed that it is in fact a substring, but the location of this substring is still up for grabs. There is, however, a simple solution in order to locate the exact location of the substring after it has been confirmed in the original string used to create the BWT table: Continuing to move one character forward in the table, by locating the character in the last column in the first column until character is the end of string character. As CAT is located at the end of the original input sequence the example ends here, but the location would have been simple enough to locate.

2.4.3 BWA

Considered MAQ version 2, BWA[11] that was created by Heng Li and Richard Durbin have traded the hash-based index of MAQ into and index based on the Burrow-Wheeler Transformation. Hash-based aligning were previously the only way to align longer reads, >200 bp, but the combined Smith-Waterman and Burrow Wheeler (BWA-SW) aligner utilized in BWA out-performs hash-based software such as BLAT and SSAHA2, while maintaining a small memory footprint, approximately 2.39 GB[9], in addition to being several times faster then them both. This trade lets BWA perform searches at a higher speed than its preceding MAQ while keeping the way of reporting *"a meaningful quality score for the mapping that can be used to discard mappings that are not well supported due to e.g. a high number of mismatches"*.

2.4.4 SOAPv2

SOAPv2[12], created by Li et. al., is an alignment algorithm specifically designed for detecting and genotyping single nucleotide polymorphisms. Just like BWA, SOAPv2 traded the hash-based index used by SOAPv1 for one created by the BWT and then building a hash table to accelerate searching those indexes even more. Designed for finding and genotyping single nucleotide polymorphisms (SNP), SOAPv2 were able to cut down its memory usage to almost a third of SOAPv1, 14.7GB to 5.4GB, while boosting the mapping speed by 20 to 30 times as high. Compatible with paired and single end reads, SOAPv2 also supports several text and compressed file formats and there have also been developed a builder to assemble and detect SNP from mapping of short reads to a reference genome.

2.4.5 BOWTIE

Bowtie[10], developed by Langmead et. al., claims to be *"an ultrafast, memory-efficient alignment program"* if you want to map relatively short reads to a genome, able to align more than 25 million reads every CPU hour when mapping to the human genome. It utilized an index created by the Burrow-Wheeler Transformation, improving previous methods with a novel quality aware backtracking algorithm. While permitting mismatches, it does not allow gapped alignment of unpaired reads. Bowtie can also boast a small memory footprint of only 1.3 GB and is highly configurable to combat the default trade-off of compromising the quality of the read mapping when it can't find exact matches to improve both memory usage and speed. Configuration of Bowtie is a double edged sword, every configuration to improve something may result in a decrease in another aspect, e.g. when configured for maximum speed it is not certain that it will be able to align valid mappings. When it comes to performance on multiple cores, claiming to scale well, it is able to use several cores to increase the overall speed of the alignment process. Bowtie is open source and available for download from http://bowtie.cbcb.umd.edu.

Due to the increasing demands in terms of throughput of sequence aligners, FM indexing has become increasingly more popular to increase speed and memory efficiency. However, this is not optimal to map longer and gapped alignments. Bowtie, re-released as Bowtie2[9], combines *"the strengths of the full-text minute index with the flexibility and speed of hardware-accelerated dynamic programming algorithms to achieve a combination of high speed, sensitivity and accuracy."* The new version have made a small sacrifice of the memory footprint to gain the improvements, going from a footprint of 1.3 GB to almost 2.5 times larger, 3.24. Bowtie is now more than 2.5 times faster then BWA when both are executed on their default settings, and align a greater amount of reads than either BWA or SOAP2. Bowtie 2 source code is downloadable from

2.4.6 SWIPE

SWIPE[14], created by Torbjørn Rognes, is one of the fastest Smith-Waterman algorithms that utilizes SIMD to parallelize local sequence alignment. Rognes proved that SWIPE were twice as fast as the previous ruling algorithm BLAST when using the BLOSUM50 scoring matrix and since it is based in SW it can guarantee the optimal local alignment, something BLAST could not. SWIPE widens the potential of SW searches and proves that SIMD parallelization could be beneficial when opting for improved performance in optimal local alignment applications.

Part II

Implementation

Chapter 3

Startup

3.1 Initial setbacks

This thesis aimed to determine whether a Xeon Phi could be utilized as a contender for the more traditional GPU. Originally, hardware were supposed to be supplied, set up and ready to go, before the thesis work began. Due to unforeseen events and inconvenience this were not the case and making the Xeon Phi work from scratch, short of actually assembling the pieces, became an major part of the early thesis work.

3.1.1 Hardware cooperation

The Xeon Phi were firstly tried to get working alongside a different processor than the Xeon CPU, but when the co-processor refused to even communicate with it, a decision were made to follow Intels' own recommendations on everything regarding cabinet, processor and cooling system.

3.1.2 Software acquisition

The computer containing the Xeon Phi were not utilized by this thesis alone and during the initial phase of the thesis work required by a non Xeon Phi project preferring the CentOS operating system.

Initially, attempts to get the communication going between host CPU and the Xeon Phi were performed. Resulting in some rather ugly hacks and an unstable environment leading to a decision on changing operating system to another more cooperative towards the Xeon Phi. For this thesis, and another running in parallel also utilizing the Xeon Phi, a Linux distribution were still preferable over Windows and thusly Ubuntu were chosen.

There are two main requirements, besides the code itself, to get a program to execute on the co-processor; Intels Manycore Platform Software Stack (MPSS) for the co-processor itself to work, and a compiler that compiles the code for Many Integrated Core architecture (MIC). After the previous hardware problems, emails were sent and phone calls made to Intel in order to acquire software and licenses before the long and tedious installation process begun.

3.1.3 Achieving communication

At this time in the process, a choice of parallelisation had yet to be made and were a stand off between Open Multi-Processing (OpenMP) and Message Passing Interface (MPI). By trying to execute some of the included test programs we found out that both the OpenMP and MPI libraries needed to execute such programs were only provided but not included in the build/installation process. The task of transferring the required libraries into the co-processor, however simple as it may sound, revealed another problem area. Firs of all, root access to the Xeon Phi were needed, in order to copy the libraries to the correct location, and not easily obtained. Several attempts were made to remedy this including an attempt to set the PATH variable. It turns out that during runtime, the co-processor only searches roots /lib64 folder. The other issue at this point were MPI support as the initial student license received from Intel only supported OpenMP.

The final solutions became another phone call to Intel for a new license, and mirroring the users and passwords from host operating system into the local Linux kernel on the Xeon Phi.

Finally all problems seemed to have been solved and the test programs executed perfectly. The thesis could move on to what it were initially intended for, creating an application that mapped multiple input queries to a reference genome, starting off with the Prototype.

Chapter 4

Working with the human genome

4.1 Implementation

The source code developed in this thesis can be viewed in full from the bitbucket repository located at the following address: https://bitbucket.org/reidarbrenna/master/src

As there are two main parts to the alignment suit presented by this thesis, the genome preparation and the alignment itself, there are two chapters detailing implementation process as well. When the implementation process first began, after the decision of utilizing the FM index were taken, the natural start were to begin with the FM index and then move on to the alignment afterwards. Therefore, this chapter will explain the process that starts in several FASTQ formated files, containing individual chromosomes, going through the FM index before ending up in a thesis created format called CRS. Chapter 5 will then continue on with the implementation of the actual search procedure.
Since this thesis aimed to test whether the Xeon Phi were suited to align sequences, the implementation were made with regards to certain limitations.

4.1.1 Referencing the genome

Before the actual implementation, the thought process behind some of the choices will be explained to better the understanding of the code that follows.

In order to work with something as large as the human genome, over 3GB long, there have to be some way of referencing the sheer quantity of characters. Thusly, before the actual implementation, this thesi will first explain the basic though process behind the chosen reference technique before going into the implementation itself. In order to work and manage something as large as the human genome, over 3GB long, there had to be some way of referencing the sheer quantity of characters in a reliable and compressed manner. The next sections will start of by explaining the journey from beginning to a solution before the implementation of the indexer wrap up this chapter.

Thought behind the process

First of all, the choice on whether to just append the whole genome into one string, or, as stated above, implement some sort of referencing technique. The former implies that the program should execute Smith Waterman, a local alignment algorithm, to align a relatively small query sequence of 200 chars to a more than $3 * 1024^3$ char long reference string in one go. On the other hand, the later proposes to do some sort of pre calculation on the genome to make it more manageable when trying to locate the query sequence. The choice were simple, and it were decided that before running any form of alignment on the genome, that it should be easier to reference first. Full-text index in minute space (FM index) were chosen as the desired method of referral. The choice of indexing the genome came with an added benefit of one being needed to execute once for any given genome, as the human genome does not change on a frequent basis. The name of the indexing program became CRSbuild, named after its output file, described further down in this chapter.

CRS - Modifications to the FM index

In order to create, and take advantage of, the FM index when searching while still being able to search efficiently, some changes had to be made. Achieving two things with the same change, the first step were to allow the c++ compiler to sort the end of string character with a lower value than A, thusly always placing that character in the top left position of the sorted BWT table while removing the beginning of string character. This results in one less line in our FM-Index table, a 2xM matrix created by the first and last column in the sorted BWT table where M is the size of the genome + 2, as well as reducing M by 1.

F	L
A	T
A	|
A	G
A	N
A	C
C	T
C	A
G	C
N	A
T	T
T	A
T	A
T	A
|	$
$	T

⇒

F	L
$	T
A	T
A	$
A	C
A	G
A	N
C	T
C	A
G	C
N	A
T	A
T	T
T	A
T	A

The next steps involved the relation between the two columns itself, the characters in the last column refer to it's counterpart in the first according to how many of that given character precede the one in question in the last column, i.e. the A in the last column on row 8 refer to the first A in the first column as no A is above it in the last column while the A on the last row refer to the fifth A in the first column as it has 4 above it. By replacing the whole first row with an index table that tells the index range of the characters in the first row instead, the size were significantly reduced while still retaining the search precision. i.e. by adding the count of similar characters above the character in the last column to the corresponding index in the index table the resulting index is the row number where the character in the last column resides in the first column.

$	T
A	T
A	$
A	C
A	G
A	N
C	T
C	A
G	C
N	A
T	A
T	T
T	A
T	A

⇒

T
T
$
C
G
N
T
A
C
A
A
T
A
A

	A	C	G	N	T
start idx	1	6	8	9	10
end idx	5	7	8	9	13

The third step in the FM index modifications were to create a table to hold the cumulative occurrence count for each of the characters occurrence in the remaining L column. The rank array, as it became called, is an $5 * N$

matrix where N is the length of the original input sequence. The matrix holds the character count from the beginning of L up to and including the character in the corresponding row, e.g. for any given row in the matrix, the content would be the total count of each character in L up to this point.

L
T
T
$
C
G
N
T
A
C
A
A
T
A
A

A	C	G	N	T
0	0	0	0	1
0	0	0	0	2
0	0	0	0	2
0	1	0	0	2
0	1	1	0	2
0	1	1	1	2
0	1	1	1	3
1	1	1	1	3
1	2	1	1	3
2	2	1	1	3
3	2	1	1	3
3	2	1	1	4
4	2	1	1	4
5	2	1	1	4

	A	C	G	N	T
start idx	1	6	8	9	10
end idx	5	7	8	9	13

After implementing the search procedure, mentioned later in this chapter, it became clear that in order to rule out possible matches that were illegal due to cross chromosome match, the individual chromosome locations needed to be stored. The resulting array contained the following; at index 0 the number of individual chromosomes appended together to create the genome, and the rest of the entries giving the start index in the appended genome where each chromosome begins. In addition, to ease the creation of reference strings to send the Smith-Waterman implementation, the original reference sequence needed to be kept as well as an additional row in the rank array containing the original location of the character in the genome. These two changes in addition to the rank array added several times the genome length in bytes, and some changes had to be made. The solution became to compress both the remaining L column as well as the genome. As there is impossible to compress 5 letters down to only 2 bits, and 3 bits does not allow for easy byte distribution, a choice to separate the N out into its own bitmap and the remaining 4 letters down into 2 bits compression.

$$\begin{aligned} A &= 0b00 \\ C &= 0b01 \\ G &= 0b10 \\ T &= 0b11 \end{aligned}$$

The rank array were also compressed to reduce the size of the datastructure that the modified FM index turned into. This were done by only keeping every 64th row of the rank array, a reduction that only affected size and runtime, while keeping the search potential intact. Before the rank array were created, all the previous characters in L would have to be traversed in order to find the corresponding letter in F. After the size

reduction, a maximum of 63 iterations upwards is required instead.
After all of these modifications, the resulting data-structure were named Compressed Reference Sequence due to it's content. Figure 4.1 shows the content as a whole in addition to the equation for calculating the size.

Figure 4.1: Compressed Reference Sequence

The CRS struct		
char *	ref	# Compressed A,C,G,T of genome
char *	ref_bm	# Bitmap of N locations in genome
char *	L	# Compressed A,C,G,T of FM index
char *	L_bm	# Bitmap of N locations in genome
unsigned int *	location	# Chromosome location array
unsigned int **	rank	# Rank array
unsigned int **	idx	# Index table

Size of struct:

$$\begin{aligned} N &= \text{Length of genome} \\ M &= \text{Number of chromosomes} \end{aligned}$$

$$\begin{aligned}
R &= ceil((N+1)/4) && \text{size of ref} \\
Rb &= ceil((N+1)/8) && \text{size of ref\_bm} \\
L &= ceil((N+1)/4) && \text{size of L} \\
Lb &= ceil((N+1)/8) && \text{size of L\_bm} \\
l &= M * sizeof(\text{unsigned int}) && \text{size of location} \\
r &= (6*(N+1))/64 && \text{size of rank array} \\
i &= 10 * sizeof(\text{unsigned int}) && \text{size of Index table}
\end{aligned}$$

Hardware limitation

The Xeon Phi used throughout most of this thesis were the Intel Xeon Phi 3120, and as such only supported 6 GB of memory. This were one of the most limiting factors that came into play during the implementation phase of this thesis. The choice to only keep every 64th line of the rank table as well as the one to compress the reference and L column down to 3 bits per character were made due to this memory limitation in order to have enough memory left for calculations. The relatively small cache size that also accompany the 3120 series were a poor match to the FM index search method as this method require a lot of jumping through the CRS, and thusly resulting in more page misses than strictly necessary.

Handling the alphabet

The alphabet of a DNA consist of the aforementioned A, C, G and T. After bit compression they got the binary value of 0, 1, 2 and 3 respectively and in order to ease the search process the rank array and index table got reshuffled, N were moved to the back so the value of the compressed letter would function as an index in those arrays. As the sequencers are not perfect and does not always know which character to place at all times they use other characters, such as N, to indicate this uncertainty. To keep the memory footprint of the CRS down, all such characters are handled like they were N and thusly unknown. As the N got its own bitmap to keep track of its location throughout the CRS, they would also be encoded as A in the regular compressed string, as A is equal to $0b00$. This allows the search for N to only happen when an A is encountered instead of every time a character are to be determined. Assuming an equal equilibrium among the 4 primary characters and an insignificant amount of N, the extra search would only be required every 4th lookup.

4.1.2 From FASTQ to CRS

The prototype created to transform all of the chromosome inputs into a whole genome and then compressing consist of a series of linear computations and read operations.

Firstly all of the different FASTQ files are read into one single string, appended one after another and the index of each of starting points are recorded in the location array. To speed things up, the FASTQ input files are assumed to come the order; chromosome 1-22, MT, X, Y and then uncertain sequences. This order is hard-coded into the program and the program treats the input as if it were given in this order.

After creation the genome string a sorting algorithm that consist of an parallelized for loop, that executes the c function qsort to sort the genome by comparing all suffix strings with every other suffix string and then merging the thread results into a single array before creating the L column. This method can be seen in figure 4.2 After the CRS structure have been filled it is written to a file along with the size of the struct so that the search program may easily read the whole CRS structure into memory in one go and thusly removing the need for indexing to be executed every time before aligning.

Modification options

The character compression is hard-coded into the software and would require some major changes to the source code in order to change, although not impossible it is not advisable at this stage. However, the rank array compression, is easily changed by changing the defined value rov in common.h. This value dictates the interval for which lines that should be included, and setting the value to 1 would result in keeping the full array.

Figure 4.2: Creating the L column

```
create L column:
#pragma omp parallel for
    for (i = 0; i < len; i++)
        Create index table to not destroy original sequence

#pragma omp parallel for
    for (i = 0; i < threads; i++) {
        each thread sorts sections of the original sequence
        until all individual sequences are sorted.
    }

    merge_multiple() to sort the sorted lists together.

#pragma omp parallel for
    for (i = 0; i < len; i++) {
        fill L column by following the sorted index table.
    }
}

merge multiple:
    while(threads used > 1)
#pragma omp parallel for
        for(i = 0; i < N; i += 2) {
            merge()
        }
        half number of threads
    }
}

merge:
    while (i & j < size) {
        if (first input <= second input)
            take from first;
        else
            take from second;
    }
    if (i != j)
        take the rest of the non empty input.
```

Due to the scope of this thesis, and the fact that this should only be executed once in order to build the CRS file, this program remained in its prototype stage as explained above.

Chapter 5

Searching for a needle

This chapter explains the other half of the program duo mention in the begining of the previous chapter, the alignment process.

5.1 Aligning the query

The process of aligning a DNA sequence to a genome is somewhat tedious, as most of the steps required are strictly linear, all depending on previously made computations. Thusly, both vectorization and parallelization were kept in mind together with cache limitations when the prototype were created.

5.1.1 Searching for the best location

The first thing that were established were the general flow of the program, in order to analyze the potential for parallelization. As seen from figure **??**, there are 3 main parts to the program, finding potential matches, aligning them with the Smith-Waterman algorithm, and then providing output in the form of a SAM file.

The alignment program created in this thesis had to utilize the Xeon Phi to the fullest and in order to do so, a new program flow were thought to provide the most balanced outcome while still also easily amended if the balance are found to be skewed. An initial threading, splitting the program in two, allows both reading and process two queries at the same time. After the first search procedure is complete, one thread will read and process the next query, if any, and the other will calculate Smith-Waterman to determine the best location before writing the results to an output file. Additionally, if the initial search to find possible matches results in a perfect match, there will be no need to calculate SW and thusly skipping ahead directly to the output section.

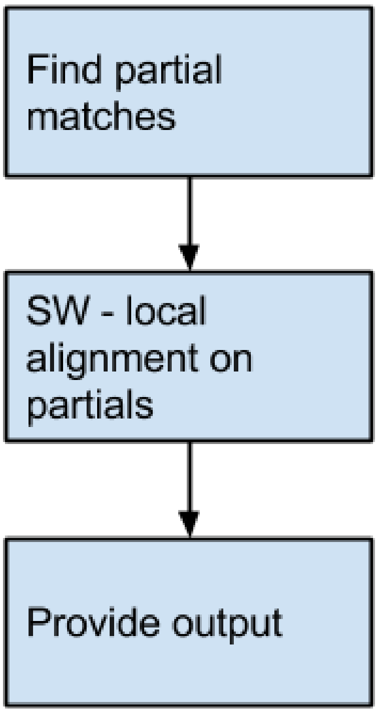


Figure 5.1: General program flow

Finding possible match locations

The first step when processing the query is to compress the query in the same manner as the L column as well as creating a compressed version of the reverse-complimented query as well. The reverse-compliment is created to combat the fact that a DNA string itself is originally paired with its reverse complimented. An example; the reverse-compliment of *CANTGAT* by following the conversion table, see table 5.1, is *ATCANTG*. The process involves starting at the back of the query and taking the complimented and placing it first, and so on until the whole original query have been changed. After creating the two compressed query strings, both are processed in turn by beginning at the end of the string and matching sequential characters forward in the CRS by following the formula, presented in figure 5.2, until no match is possible or the end of the query have been reached. The resulting (sub)string is stored in a list sorted by match length and if there are any characters left in the query, the process starts anew from the location where it left of. If an N or the end of

A	C	G	T	N
T	G	C	A	N

Table 5.1: Reverse-Compliment table
Conversion from top to bottom.

line character is encountered, an automatic mismatch occur.

Figure 5.2: FM index - search equation

```
search range given by:
    (tmp_)St = start index
    (tmp_)Ed = end index

length = how many characters matched

for (each char in query, reversed order)
    if (char == N || end_of_string)
        if (length > 0)
            store_found(St, Ed, length)

    if (first)
        St = index_table[char][0]
        Ed = index_table[char][1]
    else
        tmp_St = index_table[char][0] + rank_table[char][St-1]
        tmp_Ed = index_table[char][0] + rank_table[char][Ed] - 1

    if (tmp_St > tmp_Ed)
        \\missmatch found
        if (length > 0)
            store_found(St, Ed, length)
            make sure for loop starts again at current char.
            length = 0
    else
        St = tmp_St
        Ed = tmp_Ed
        length++

After for loop compleation
if (length > 0)
    store_found(St, Ed, length)
```

When storing only the 1600 best matches are kept for further processing, and if a match is found to be crossing over from one chromosome to another the match is dropped altogether.

Speed vs. Accuracy

When calculating the possible matches for the two queries, a choice had to be made on how through the matching process should be. One could start a new match sequence for each character in the query, resulting in a small possible increase in accuracy while suffering speed loss and several almost equal matches. Another option were to thread the search procedure, starting going through the FM-index on an individual basis. After the initial step of setting St and Ed when length is 0 from figure 5.2, spawning a thread that would utilize the search procedure explained in section 2.4.2 for each possible start location, and then spawn a new thread for each possible forked road encountered. This were quickly discarded as, if assumed equal distribution of A, C, G and T, a potential of 750000 threads could be spawned from the initial thread spawn alone.

5.1.2 Calculating Smith-Waterman

Even though Smith-Waterman is a strictly serial calculation equation, by utilizing the vector unit on the Xeon Phi, 16 simultaneously calculations were achieved, where the calculations occur on a column basis of the Smith-Waterman matrix. This were inspired from the Rognes' SWIPE[14] implementation and it also portrayed a way of only keeping two columns in memory at the same time. As the 1600 best matches were kept, the whole Smith-Waterman calculation could also be calculated in parallel, spawning 100 threads to calculate the match score of 16 possible locations each, all at the same time. The slightly modified SWIPE Smith-Waterman implementation can be seen in figure 5.3. The function requires several input to function properly;

- A pointer to the memory area (HE) for storing two columns worth of data to keep track of the calculated values and the cumulative row gap penalties of the SW calculation matrix.
- A pointer to the memory area (sp) that holds information on whether the reference match is from the regular or the reverse-complimented query. This memory area should also holds the results after calculation.
- A char array (q) holding the query bundle, as explained by figure **??**.
- The length of the query itself (q_len).
- An indication as to how many different reference positions it should match the query to (multiple), ranging from 1 to 16 depending on native vs host execution and or uneven results from the previous step to find possible locations.
- An index (si) to let the function know where in sp the current execution should located and then store information to/from.
- A pointer to the memory area that holds the uncompressed reference sequences it should match the query against.

The last input parameter were not required in the first version of the function as it then read the references from the compressed reference in the CRS from locations found in sp. The choice to change the algorithm were made to combat the poor cache capabilities of the Xeon Phi and will be explained later in section 5.1.3.

There are also several local variables created for each instance of the function;

- int i and j to loop through the columns and rows of the SW calculation matrix, respectively.
- int t to iterate through the different reference sequences when acquiring the cell score.
- and unsigned int r_len to hold the length of the reference sequences to match against. This length is the same as the length of the query with an extra few characters added before and after the match to allow for some adjustment to the initial partial match found.
- vector variables S and SM to hold the highest score found for each reference as well as the score for the currently calculated cell, respectively.
- the H vector variable, initially holds the diagonally cell value before moving on to become the current cell after added with the scores (SM).
- lastly, the vector variables E, F and N each hold the values of the cell to the left, cells above and the cells diagonally up to the left, respectively. The E and F variable holds the same purpose as described in Gotohs improved SW algorithm, section 2.3.2, seen in figure 2.5. The N variable help H to fulfill its role according to the Gotohs improved SW algorithm by holding the value of the cell to the left.

First thing that the function does is to make sure that both the HE array and the max score found is set to 0 as this is the initial values of all SW calculations. It then proceeds to the double for loop to do the actual calculations. Before staring to calculate a new column, both H and F are set to zero to fulfill initial values as if the calculations had been done utilizing the whole matrix. Then before each cell calculation, N and E are updated according to the HE array before storing the cell scores for each reference match into the SM vector. The latter had to be done in a for loop as there were no intrinsic support on the Xeon Phi to load a 512 vector with 16 individual integers, before casting the results into the vector. The first set of intrinsic calls are taken straight from the algorithm itself, first adding the score to the diagonal cell value and then finding the highest value among the result, E, F or 0 to store in H. If the new H values exceeds any of the values in S, they replace those values in S to always have the highest cell score stored in S. The final step performed when calculating a cell is to prepare for the next cell by updating the E and F values before updating the HE array. After all cells have been updated a simple for loop withdraws the maximum scores for each reference sequence and stores them in their respective slots in the storage area.

5.1.3 Repacking the reference genome

In order to decrease the amount of cache lines needed to be read in the Smith-Waterman implementation, the possible locations used in one

QNAME	A maximum of 20 of the first character in the query ID given as input.
FLAG	Only the reverse-complimented bit will be used.
RNAME	Will contain the chromosome name of the alignment; 1-22, X, Y, MT or Non.
POS	1-based index of the leftmost mapped character.
MAPQ	The default value of 255 will be used for this field.
CIGAR	The CIGAR string contain the following characters: M, D, I and S. They will be written in a compressed manner, such as MMM will be 3M.
RNEXT	Default value * will be used.
PNEXT	Default value of 0 will be used.
TLEN	Default value of 0 will be used.
SEQ	The sequence mapped to.
QUAL	Default value of * will be used.
AS:i:	The actual score returned by SW.

Table 5.2: Expected values in SAM output

run of the algorithm were repacked into a single string. The initial implementation relied on the whole reference string and indexes to retrieve the information stored within. This resulted in a new cache line being read almost every time, up to 16 times per round in the inner loop of the implementation, as the sections in the reference string being mapped are most likely positioned in different locations. The new implementation were to withdraw the information before the calculation and then merging those into a single char array that the algorithm could use instead. The merging happens one section at a time to prevent all of the cache line reads to move outside instead of being removed. Each character in the section to be merged, are placed at an equal interval in the new string where the starting location increases once for each section added. An example of this merge can be seen in figure 5.4, and resulted in far less cache lines being read as all the data for the reference sequences were now aligned in memory with the correct order.

5.1.4 Providing output

For each query provided, only the best match found will be written to the SAM file. The SAM file output for each query, provided by this program, will only contain the mandatory 11 fields as well as a 12th field to provide the aligners own alignment score. The filed data presented in table 5.2, are meant to illustrate what data to expect from the aligner.

Figure 5.3: Modified Smith-Waterman implementation

```
void sw_simd(...) {
    int score_results[16];
    unsigned int i, j, t;
    unsigned int r_len = q_len + (2*SW_REF_PADDING);

    __mxxxi H, E, F, N, SM, S;

    S = *(__mxxxi *)ZERO;

    for (i = 0; i < r_len; i++) {
        H = *(__mxxxi*)ZERO;
        F = *(__mxxxi*)ZERO;

        for (j = 0; j < q_len; j++) {
            N = HE[j*2 + 0];
            E = HE[j*2 + 1];

            for (t = 0; t < multiple; t++)
                fill score_results with matching scores.

            SM = *(__mxxxi*)score_results;

            H = _simd_add_epi32(H,SM);
            H = _simd_max_epi32(H,F);
            H = _simd_max_epi32(H,E);
            H = _simd_max_epi32(H,*(__mxxxi*)ZERO);
            S = _simd_max_epi32(H,S);

            HE[j*2 + 0] = H;

            E = _simd_sub_epi32(E,*(__mxxxi*)GE);
            F = _simd_sub_epi32(F,*(__mxxxi*)GE);
            H = _simd_sub_epi32(H,*(__mxxxi*)GOE);
            E = _simd_max_epi32(H,E);
            F = _simd_max_epi32(H,F);

            HE[j*2 + 1] = E;
            H = N;
        }
    }

    for (i = 0; i < multiple; i++) {
        sp[si+i].score = ((int *)&S)[i];
    }
}
```

Figure 5.4: Reference merge repacking

An example showcasing the merging of four sequences
into a single string:

$$\begin{array}{rcccccc}
s_1(GNAT) & - & G_{1,1} & N_{1,2} & A_{1,3} & T_{1,4} \\
s_2(ACAT) & - & A_{2,1} & C_{2,2} & A_{2,3} & T_{2,4} \\
s_3(CANT) & - & C_{3,1} & A_{3,2} & N_{3,3} & T_{3,4} \\
s_4(AGAT) & - & A_{4,1} & G_{4,2} & A_{4,3} & T_{4,4}
\end{array}$$

$$\Downarrow$$

$$G_{1,1}\ A_{2,1}\ C_{3,1}\ A_{4,1}\ N_{1,2}\ C_{2,2}\ A_{3,2}\ G_{4,2}\ A_{1,3}\ A_{2,3}\ N_{3,3}\ A_{4,3}\ T_{1,4}\ T_{2,4}\ T_{3,4}\ T_{4,4}$$

Where the lower case numbers symbolizes originating string and
index within that string, respectively.

Chapter 6

Tuning the application

There are a few options available to tune the alignment program, and will be explained in this chapter.

6.1 Runtime tuning

There are several command line parameters available when executing DNAlign, the alignment program as seen in table 6.1 In addition, there are

<ref>	the compressed reference file (.crs) created by CRSBuild one reference file only		
[query]	$[0 - N]$ query files All parameters given after <ref> are assumed to be query files that follows the FASTQ format		
flags:	(provided before <ref> if any)		
	-h		Prints the help menu and terminates the execution
	-p		Prints the content of the .crs file to debug.out - not recommended
	-n	<int>	Number of threads to use - Default value: 1
	-m	<int>	Number of matches to write to SAM file - Default value: 1
	-l	<int>	Minimum length of partial match to keep - Default value: 20 - Higher value results in longer matches - Lower value results in shorter matches

Table 6.1: DNAlign command line overview

also possible to change the accepted query length which is set to 200 that requires recompilation to function. There are also provided a Makefile that have two main features; make all and make phi, which builds the program to execute on a Xeon Processor or the Xeon Phi processor respectively.

6.2 Memory usage

As the available memory is quite limited on a Xeon Phi, a higher query length than 200 is not advisable on the low end Xeon Phi co-processor, and the program have not been tested on such. The program aim to utilize all available memory when executed with 200+ threads.

6.3 Threading

The thread option available to the user as a command line argument is the total thread count used by the program. Thusly, if more than one thread is given, one of these will be taken by the main program to run two query processing processes simultaneously.

6.4 Vectorization

Vectorization is automatically changed to match the vector capabilities of a Xeon processor or the Xeon Phi co-processors when built. As these two hardware units does not share any vector intrinsics, the Xeon processor is thusly only capable of performing 4 Smith-Waterman calculations at the same time while the co-processor handles 16 simultaneously.

Part III

Conclusion

Chapter 7

Results

7.1 Test conditions

As there are no other program that aligns DNA sequences to a reference genome developed for a Xeon Phi, the testing and comparison were done on a Xeon processor and the co-processor to gage the effectiveness of the Xeon Phi implementation. Due to the fact that a lot of time went into getting the Xeon Phi to function properly in the first half of this thesis, and the results yielded, accuracy have not been compared to other programs that does not utilizes the Xeon Phi. The tests were performed 10 times each to properly calculate a mean time for comparison.

Note: Due to unforeseen events and hardware malfunction in the testing phase, the tests were only performed with a single thread and two queries against 5 relatively small chromosomes instead of the whole genome to save time.

7.2 Testing

Native

These are the following test times achieved on the Xeon phi when aligning 5 almost perfect matches to chromosome 15.

5.951	0.842	0.769	0.746	0.772
0.778	0.734	0.731	0.742	0.727

Host

These are the following test times achieved on the Xeon phi when aligning 5 almost perfect matches to chromosome 15.

0.155	0.141	0.121	0.168	0.200
0.170	0.121	0.207	0.124	0.141

7.2.1 Comparison

Even though with relative small test samples it is quite clear that the host execution were significantly faster than the native execution.
Even though the Xeon Phi executes the Smith-Waterman calculation 4 times

less than the Xeon processor it is about 5 times slower than the Xeon in runtime. As the thesis discovered that every time a series of calculations were performed on the Xeon Phi, the first time were always much higher than the rest, the following stats have been calculated when disregarding the highest and lowest runtime of both hardwares.
The mean execution time for the Xeon Phi co-processor were 0.7655 seconds while the mean time for execution on a Xeon processor were only 0.1525 seconds. Thusly, it becomes clear that without any threads the Xeon Phi is the tortoise to the Xeon Hare.

Chapter 8

Future work

There are several things that this thesis did not get a chance to test properly and this chapter is dedicated to things that could be done in the future.

8.1 Hardware

Intel have stated that they are releasing a newer version of the Xeon Phi with more internal memory and cache as well as support for smaller unit sizes in the vector unit. However, the first iterations of the new Xeon Phi will not have the opportunity to communicate with the host CPU and thusly and offloading implementation on the new hardware will have to wait on the next release after the that. The hardware does however propose the opportunity to test this program again with more memory and less compression to remove several steps in the process when searching CRS structure.

8.2 Application improvements

This thesis did not get the opportunity to test an Offloading approach on the implementation, and that could be a viable solution to take the advantage of the host processor for the tedious linear work and only offload the Smith-Waterman calculations to the co-processor. This could provide beneficial as long as the overhead associated with the offload does not take to much time.

The general idea behind the offload approach is to allocate the memory space needed on the co-processor in the very beginning of the execution and keeping the memory allocated until program termination. The general Flow of the program would then be that one or more threads on the host CPU read input, found possible matches and created the repacked reference sequence before shipping it and the query to the co-processor to calculate the Smith-Waterman scores. After the scores had been calculated, they would be returned to the host processor that writes the findings to the SAM output file.

Regarding thread synchronization, the initial thought would be to only allow one of the host CPU threads to write to the file at the time by wrapping that part in a critical section. This would also have to be done when reading the query, to prevent multiple threads to ruin queries for each other. Based on the total thread, cache and memory availability on the co-processor, only two offloads to the co-processor at the same time would be optimal. This could of course be proven wrong by an implementation and proper testing.

Chapter 9

Final verdict

As the co-processor is today, the low end at least, this thesis have found it to be less than preferable when aligning DNA sequences to a reference genome in Native mode. With more testing and perhaps some fine tuning and an offload approach to the implementation it could have proven better than the native execution and perhaps also better than some of the other tools on the market today. Although the latter is less likely.

Bibliography

[1] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485.

[2] Michaela Barth, KTH Sweden, Mikko Byckling, CSC Finland, Nevena Ilieva, NCSA Bulgaria, Sami Saarinen, Michael Schliephake, Volker Weinberg, and LRZ Germany. *Best Practice Guide Intel Xeon Phi*. v1.1. Intel. Feb. 2014.

[3] Michael Burrows and David J Wheeler. “A block-sorting lossless data compression algorithm”. In: (1994).

[4] Osamu Gotoh. “An improved algorithm for matching biological sequences”. In: *Journal of molecular biology* 162.3 (1982), pp. 705–708.

[5] Li Heng. *SAMtools*. Sept. 9, 2009. URL: http://samtools.sourceforge.net/.

[6] Li Heng. *Sequence Alignment/map Format specification*. Feb. 28, 2014. URL: http://samtools.github.io/hts-specs/SAMv1.pdf.

[7] *History of Illumina Sequencing*. 2014. URL: http://www.illumina.com/technology/next-generation-sequencing/solexa-technology.ilmn.

[8] Jim Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Elsevier Science, 2013. ISBN: 9780124104143.

[9] Ben Langmead and Steven L Salzberg. “Fast gapped-read alignment with Bowtie 2”. In: *Nature methods* 9.4 (2012), pp. 357–359.

[10] Ben Langmead, Cole Trapnell, Mihai Pop, Steven L Salzberg, et al. “Ultrafast and memory-efficient alignment of short DNA sequences to the human genome”. In: *Genome Biol* 10.3 (2009), R25.

[11] Heng Li and Richard Durbin. “Fast and accurate long-read alignment with Burrows–Wheeler transform”. In: *Bioinformatics* 26.5 (2010), pp. 589–595.

[12] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. “SOAP2: an improved ultrafast tool for short read alignment”. In: *Bioinformatics* 25.15 (2009), pp. 1966–1967.

[13] Saul B Needleman and Christian D Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of molecular biology* 48.3 (1970), pp. 443–453.

[14] Torbjørn Rognes. "Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation". In: *BMC bioinformatics* 12.1 (2011), p. 221.

[15] Temple F Smith and Michael S Waterman. "Identification of common molecular subsequences". In: *Journal of molecular biology* 147.1 (1981), pp. 195–197.

[16] Wikipedia. *Burrows–Wheeler transform*. May 13, 2014. URL: http://en.wikipedia.org/wiki/Burrows-Wheeler_transform.

[17] Wikipedia. *FASTQ format*. May 20, 2014. URL: http://en.wikipedia.org/wiki/FASTQ_format.

[18] Wikipedia. *FM-index*. Jan. 28, 2014. URL: http://en.wikipedia.org/wiki/FM-index.

[19] Wikipedia. *Sequence assembly*. May 7, 2014. URL: http://en.wikipedia.org/wiki/Sequence_assembly.

[20] Wikipedia. *Smith-Waterman example*. Apr. 28, 2014. URL: http://en.wikipedia.org/wiki/Smith-Waterman%5C_algorithm.