MSc THESIS

Acceleration of read alignment with coherent attached FPGA coprocessors

Michael Johannes Jaspers

Abstract

With the advent of Next Generation Sequencing (NGS), the cost of sequencing human DNA has decreased significantly over the past decade. This decrease in cost has attracted a great deal of attention from medical research and is now transitioning to clinical practice. Precision medicine, tailored to a person’s genetic profile, is becoming a viable option in the battle against cancer and rare genetic diseases. NGS sequencers produce millions of small fragments of DNA called reads. Mapping those reads to a reference genome proves to be a tremendous computational task and forms the bottleneck in current DNA analysis flows. BWA-MEM, a state-of-the-art alignment tool, applies the seed and extend paradigm to rapidly align the reads with a reference genome. The Smith-Waterman (S-W) algorithm is widely adopted by these state-of-the-art aligners. We have identified the S-W algorithm to be the main computational bottleneck on IBM’s POWER8 processor. This recently released processor includes a new Coherent Accelerator Processor Interface (CAPI), that provides cache coherent access to shared memory for heterogeneous processors. We present a highly parallel FPGA-based accelerator that offloads the Smith-Waterman task. We have integrated the accelerator with software, relying on CAPI, in a tightly coupled fashion. This integrated heterogeneous system is able to achieve a speedup of 1.6X over purely software-based multithreaded execution of BWA-MEM. Furthermore, we propose a more general framework for dividing workload between the processor and accelerator in a fine-grained manner. We expect to achieve the maximum obtainable speedup (bounded by Amdahl’s law) of 2X with the proposed framework.
Acceleration of read alignment with coherent attached FPGA coprocessors

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Michael Johannes Jaspers
born in Oosterhout, The Netherlands
Acceleration of read alignment with coherent attached FPGA coprocessors

by Michael Johannes Jaspers

Abstract

With the advent of Next Generation Sequencing (NGS), the cost of sequencing human DNA has decreased significantly over the past decade. This decrease in cost has attracted a great deal of attention from medical research and is now transitioning to clinical practice. Precision medicine, tailored to a person’s genetic profile, is becoming a viable option in the battle against cancer and rare genetic diseases. NGS sequencers produce millions of small fragments of DNA called reads. Mapping those reads to a reference genome proves to be a tremendous computational task and forms the bottleneck in current DNA analysis flows. BWA-MEM, a state-of-the-art alignment tool, applies the seed and extend paradigm to rapidly align the reads with a reference genome. The Smith-Waterman (S-W) algorithm is widely adopted by these state-of-the-art aligners. We have identified the S-W algorithm to be the main computational bottleneck on IBM’s POWER8 processor. This recently released processor includes a new Coherent Accelerator Processor Interface (CAPI), that provides cache coherent access to shared memory for heterogeneous processors. We present a highly parallel FPGA-based accelerator that offloads the Smith-Waterman task. We have integrated the accelerator with software, relying on CAPI, in a tightly coupled fashion. This integrated heterogeneous system is able to achieve a speedup of 1.6X over purely software-based multithreaded execution of BWA-MEM. Furthermore, we propose a more general framework for dividing workload between the processor and accelerator in a fine-grained manner. We expect to achieve the maximum obtainable speedup (bounded by Amdahl’s law) of 2X with the proposed framework.

Laboratory : Computer Engineering
Codenumber : CE-MS-2015-04

Committee Members :

Advisor: Assistant prof. dr. ir. Z. Al-Ars, CE, TU Delft
Chairperson: Prof. dr. ir. H. P. Hofstee, CE, TU Delft
Member: Associate prof. dr. ir. R. van Leuken, CAS, TU Delft
Dedicated to my family and friends
# Contents

**List of Figures** vii

**List of Tables** ix

**Acknowledgements** xi

## 1 Introduction

1.1 Outline of this thesis 2

## 2 Background

2.1 DNA sequencing 3

2.2 NGS analysis 5

2.3 Sequence alignment 6

2.3.1 State-of-the-art sequence aligners for NGS data 6

2.3.2 Smith-Waterman 7

2.4 Smith-Waterman accelerated prior art 9

2.4.1 GPGPU 9

2.4.2 FPGA 11

## 3 Power8 and CAPI infrastructure

3.1 Introduction to Power8 and CAPI 13

3.2 CAPI architecture 13

3.2.1 Accelerator side 14

3.2.2 Data movement 15

3.3 Specialized hardware 15

3.4 Properties of CAPI 16

3.4.1 Low latency, high bandwidth data access 16

3.4.2 Shared address space programming 17

3.4.3 Coherent cache 18

3.5 CAPI applications 18

3.5.1 Low overhead storage 18

3.5.2 Acceleration of computation 18

## 4 Analysis of BWA-MEM

4.1 BWA-MEM mapping algorithm 21

4.1.1 SMEM seeds 21

4.1.2 Seed extension 22

4.1.3 Output generation 23

4.1.4 Dividing workload amongst threads 24
List of Figures

2.1 Cost of sequencing the human genome as estimated by NHGRI .......................... 4
2.2 Steps in the modern genomic processing pipeline GATK ............................... 6
2.3 Wavefront parallelism mapped to a Systolic PE Array ......................................... 10
3.1 CAPI architecture ............................................................................................... 14
3.2 Data flow of PCIe attached accelerator on traditional platform and CAPI ............... 17
4.1 Seed extension concept in BWA-MEM ............................................................... 22
4.2 Data dependencies in seed extension ................................................................... 23
4.3 Multi-threaded organization of BWA-MEM ...................................................... 25
4.4 Histogram of the lengths of input strings for 1M 100 bp reads ......................... 26
4.5 Profile BWA-MEM synthetic single-ended 250 bp reads .................................... 26
4.6 Profile bwa-mem synthetic single-ended 100 bp reads ........................................ 27
5.1 Smith-Waterman accelerator ............................................................................. 31
5.2 Shared memory constructs with accelerator and Power8 ................................. 36
5.3 Blocking seed extension on FPGA ..................................................................... 39
5.4 Non-blocking seed extension on FPGA ............................................................. 39
5.5 Work flow on heterogeneous system using a Task Queue ................................. 41
6.1 Smith-Waterman kernel scaling on 20-core POWER8 ........................................ 45
6.2 Relative FPGA performance of S-W kernel for 100 bp reads ......................... 46
6.3 Relative FPGA performance of S-W kernel for 250 bp reads ......................... 46
6.4 Average latency of S-W kernel .......................................................................... 48
6.5 Performance comparison of FPGA configurations and software solution of BWA-MEM for 100 bp reads ................................................................. 49
6.6 Performance comparison of FPGA configurations and software solution of BWA-MEM for 250 bp reads ................................................................. 50
6.7 Profile of FPGA accelerated BWA-MEM with synthetic single-ended 100 bp reads .............................................................................................................. 51
List of Tables

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Smith-Waterman similarity matrix</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Optimal alignment between two sequences</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Performance metrics of CAPI on POWER8 from an accelerator’s perspective</td>
<td>17</td>
</tr>
<tr>
<td>5.1</td>
<td>PE array configuration utilization</td>
<td>33</td>
</tr>
<tr>
<td>5.2</td>
<td>FPGA resource utilization of 100 element PE array configurations</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>FPGA resource utilization of 250 element PE array configurations</td>
<td>35</td>
</tr>
<tr>
<td>6.1</td>
<td>Accuracy comparison of FPGA accelerated and pure software BWA-MEM</td>
<td>44</td>
</tr>
<tr>
<td>6.2</td>
<td>S-W core utilization on FPGA for 100 bp reads</td>
<td>47</td>
</tr>
<tr>
<td>6.3</td>
<td>Bandwidth utilization of CAPI attached FPGA while running accelerated BWA-MEM</td>
<td>51</td>
</tr>
</tbody>
</table>
The research for this thesis has been a collaboration between BlueBee and IBM. Both parties have helped me greatly and provided me with the possibility to carry out the research. I would like to thank Professor Zaid Al-Ars for sparking my interest in bioinformatics, and guiding me through every step in the research. I would also like to thank Professor Peter Hofstee for welcoming me in Austin, Texas and making me feel at home. His daily guidance and insights have been invaluable. Furthermore, I would like to thank the people at IBM Austin, in particular Bart Blaner, JT Kellington, Frank Liu and Christopher Durham for their help and support during my stay in Austin. Of BlueBee, I would like to thank Ernst Houtgast and Vlad Mihai Sina for their help. Last but not least, a word of thanks to my friends and family who have supported me unconditionally.

Michael Johannes Jaspers
Delft, The Netherlands
August 19, 2015
Because a number of diseases are a consequence of a person’s genetic makeup, researchers and clinicians are often interested in detecting variations in a person’s genetic profile.

With the introduction of Next Generation Sequencing, sequencers are able produce large volumes of reads, representing the human DNA, in a small amount of time. These reads are characterized by their shorter length and higher error rate compared to the traditional reads generated by Sanger-based sequencers [3]. Mapping these reads to a reference genome, in order to reconstruct the original DNA sequence, is a complex task. Since variations with respect to the reference genome must be allowed (since they are of interest), mapping reads becomes a task of inexact string matching, where the characters in the strings are the nucleotides Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). A reference genome consists of around 3.2 Gbp (base pairs, a pair of nucleotides), and modern NGS sequencers produce around 100 Gbp of data, at 30X coverage.

Typically, the Smith-Waterman (S-W) algorithm is applied to find similarity or edit distance between two biological sequences [4]. This algorithm runs in quadratic time, and no faster algorithm exists [5]. Due to the large volume of available data, this algorithm has been supplemented with various heuristics in NGS analysis flows. Heuristics have been applied to reduce the search space of the dynamic programming algorithm, at a loss of accuracy. Modern alignment tools, such as BWA-MEM, apply a seed and extend paradigm [6] to find the best candidate location for short reads. Seeds, short exact matches, are found using two approaches. Hash-based methods hash the reference genome, and can find an exact match of length $l$ in $O(1)$ time. Other approaches use a backward search algorithm based on the Burrows-Wheeler Transform, taking only $O(l)$ to find a seed of length $l$ [7]. In a next stage, the S-W algorithm is applied locally to extend the candidate seeds and find the region with greatest similarity in the reference sequence. Still, the S-W algorithm is the bottleneck in modern alignment tools.

The S-W algorithm has been accelerated on platforms such as GPUs [8] and FP-GAs [2]. However, the Smith-Waterman task in BWA-MEM operates on very small sequences, with lengths between 100 and 500 base-pairs. It turns out that the communication and control overhead for such fine-grained acceleration tasks is detrimental on such platforms [2]. Traditionally only when dealing with large blocks of data, amortizing the communication overhead, can acceleration be achieved on those platforms.

The POWER8 includes CAPI, and provides coherent access to shared memory for an accelerator. Communication and control overhead is reduced compared to a traditional attached PCIe device, that requires device drivers and OS involvement for acceleration. As such, by attaching an FPGA to the POWER8 system, we should be able to accelerate the dynamic programming present in modern aligners that are based on the seed and extend paradigm.

In this thesis we address three main questions:
1. How should modern alignment tools, in particular BWA-MEM, be partitioned on a reconfigurable system with a processor and an FPGA for best performance?

2. How can we accelerate the S-W workload of BWA-MEM on an FPGA?

3. How can we use shared memory for control and communication of a CAPI attached accelerator, for the acceleration of fine-grained tasks such as sequence alignment?

The main contributions to this thesis are as follows:

1. Implementation of BWA-MEM on a POWER8.

2. Analysis of BWA-MEM and identification of bottlenecks for different workloads on a POWER8 processor.

3. The design and implementation of a high-throughput Smith-Waterman accelerator on a FPGA for small DNA sequences, leveraging a previously designed Smith-Waterman macro [9].


5. Measuring characteristics of CAPI.

6. Integration of a Smith-Waterman accelerator and BWA-MEM.

7. General models for integration of a tightly coupled accelerator with a CPU through shared memory.

1.1 Outline of this thesis

This thesis is organized as follows. In Chapter 2, we introduce the reader to Next Generation Sequencing (NGS). We discuss modern analysis flows for NGS data, and briefly address the Smith-Waterman algorithm, widely used in these analysis flows. Finally we discuss the state of the art in sequence alignment. Chapter 3 introduces the new Coherent Accelerator Processor Interface (CAPI), providing coherent shared memory access for an accelerator on the POWER8 processor. We briefly describe the architecture of the POWER8 processor, and address the potential applications of CAPI. In Chapter 4, we perform an in-depth analysis of sequence aligner BWA-MEM, and based on this analysis and properties of CAPI, we map portions of BWA-MEM to either the processor or accelerator. We discuss the hardware design of the accelerator and coupling with BWA-MEM in Chapter 5. The results on performance of the Smith-Waterman acceleration and integrated solution are presented in Chapter 6. Finally Chapter 7 draws conclusions and gives an outlook for future work.
This chapter serves as a short introduction to genomics, a discipline that analyses the function and structure of the genome. In Section 2.1 we discuss the relevance of DNA sequencing and we describe its technology advances of the past decades. These advances in technology have lead to a revolution in analysis of DNA, and we discuss state-of-the-art analysis tools for current DNA analysis flows in Section 2.2. Sequence alignment or mapping is a first step in modern DNA analysis flows and therefore we briefly discuss it in Section 2.3. In Section 2.3.2 we discuss Smith-Waterman, a dynamic programming algorithm that is most widely used for sequence alignment. This compute-intensive algorithm has received a lot of attention in literature, and we discuss the most common approaches for acceleration of this algorithm in Section 2.4.

2.1 DNA sequencing

DNA sequencing is the task of determining the order of nucleotides within a DNA molecule, the four nucleotides being adenine (A), guanine (G), cytosine (C), and thymine (T). It can be used to detect the sequence of single genes, chromosomes and even full genomes, or just the exome, the region of the genome that contains the genes. Sequencing allows researchers to study the genome, identify variants and this information is valuable to various disciplines, including but not limited to biology, forensics, diagnostics, and medicine.

The human genome project (HGP) was the first project to successfully sequence the human genome. It was started in 1988, and only finished in 2003, when it was announced that 99% of the genome was known with 99.99% accuracy. One of the remarkable outcomes of this project was that the human genome contains only 20,000 to 25,000 genes. It is estimated that costs of the HGP were around 2.7 billion dollar, and various countries contributed to the project.

One of the first methods for rapidly determining the sequence of DNA is attributed to Sanger [3], awarded with a Nobel prize for chemistry in 1980. His methods are based on chain termination, and can deliver short sequences between 500 and 700 nucleotides in length. It was considered the gold standard for nearly two and a half decade. Since 2005, the method of sequencing DNA has shifted from traditional Sanger-based methods to newer techniques referred to as Next Generation Sequencing (NGS) [10], driven by the demand for higher throughput sequencing and lower costs. NGS methods are based on massively parallel sequencing. This greatly improves the throughput of sequencing, and allows a full human genome to be sequenced within a day.

Figure 2.1 shows the decline in costs of sequencing the human genome as measured by the NHGRI and it is projected against the costs of computing power on silicon predicted by Moore’s law [11]. Note that the scale on the y-axis is logarithmic. The costs of
sequencing have gone down from $100M at the beginning of the millennium to just $10,000 at the start of 2011, and continues to decline in this decade. The emergence of NGS platforms is clearly visible during mid 2007, where the sequencing costs are dropping at an extreme rate. Illumina, one of the companies providing sequencing machines, has recently announced their HiSeq x Ten System, claiming to sequence a human genome for under $1000.

The decline in cost has lead to a number of innovative projects, including the 1000 Genomes Project, completed in 2012 when 1092 genomes were sequenced worldwide [12]. In order to improve the ability to diagnose, treat and prevent cancer, The Cancer Genome Atlas (TCGA) project was launched in 2005, and it aims to catalogue genetic mutations responsible for cancer [13].

More recently, sequencing genomes has received support from the US government. A news article states that US president Barack Obama announced to finance the sequencing of over one million volunteers [14]. The project aims to support studies of cancer and rare diseases, and is a step forward into the development of precision medicine.

A recent study expects that between 100 million and 2 billion genomes will be sequenced by 2025 [15]. The article compares genomics and various other Big Data areas, and expects that storage of genetic data will outscale the storage of companies as Youtube and Twitter.
2.2 NGS ANALYSIS

Not only storage of the data will pose a challenge, the computational power required for analysis of the genetic data is immense. The analysis is considered the rate-limiting factor in genomic research of today [16].

2.2 NGS analysis

A major interest of genomic research is in variant calling, finding variations in the DNA associated with a wide variety of diseases. Those variants are generally classified into three categories:

1. Single nucleotide polymorphisms (SNP). A variation in a single nucleotide. For example, a SNP might replace a cytosine (C) nucleotide with thymine (T) in a certain location in the DNA.

2. Insertions / Deletions (INDEL). An INDEL is a characterized by a single nucleotide that has been inserted in / deleted from the genome.

3. Structural Variation (SV). SV is defined as a region of over 1k nucleotides characterized by a translocation between chromosomes, deletions, and inversions of a large region.

Current NGS techniques produce large volumes of data in the form of small fragments of DNA, referred to as short reads. Those short reads range in length between 35 and 500 nucleotides, depending on the sequencing platform [17]. Traditionally, Sanger-based methods produced long reads of 500 nucleotides or more and with higher quality. Downstream analysis of these short reads, such as the variant calling described above, requires knowledge about the origin of the reads obtained by sequencing and their relation to each other, i.e. the sequenced DNA needs to be reconstructed to its original form. Reconstruction methods exist in two forms:

1. De novo assembly, the overlapping short reads are assembled and form longer sequences, called contigs [18].

2. Resequencing, the short reads are mapped to a reference sequence. This builds a larger sequence that is similar but not identical to the reference sequence.

Numerous frameworks have emerged in the last decade for variant calling on NGS data, using resequencing as a construction method. These frameworks are often referred to as pipelines, as they contain multiple processing steps. One of the more prominent and widely used pipeline is the Genome Analysis Toolkit (GATK), developed at the Broad Institute of Harvard and MIT [1]. Figure 2.2 depicts the typical steps in modern genomic processing pipelines. These steps are advocated by GATK as best practices. The first step in this pipeline after acquisition of the NGS data is mapping the short reads to their corresponding location in a reference genome. This step is termed read alignment or sequence mapping and will be the focus of acceleration for this thesis and is explained further in section 2.3. Subsequent steps apply ordering, filtering, removing of duplicates of the mapped reads.
2.3 Sequence alignment

The genome of a human and a reference genome show great similarity. However since NGS sequencing is error-prone, and the DNA sequence may be subject to the variations such as SNPs and INDELs, the reads obtained from NGS may be inherently different from the reference genome. Since we are interested in finding those variations, mismatches between nucleotides and gaps should be allowed in the mapping of the reads to the reference. The process of mapping reads to a reference is referred to as sequence alignment. The most popular traditional algorithms for sequence alignment are the Smith-Waterman algorithm and BLAST. These aligners require copious amount of time to compute accurate alignments. As a result, numerous of aligners have been developed lately specifically targeted towards NGS data. These aligners are orders of magnitude faster than the traditional algorithms. In section 2.3.1 state-of-the-art sequence aligners will be discussed briefly.

2.3.1 State-of-the-art sequence aligners for NGS data

Due to the long runtime of traditional sequence aligners, numerous modern aligners have been developed specifically targeting NGS data. These tools have been designed to cope with large volumes of short reads (between 30 and 500 bp), with relatively high error margins in the sequenced data. The short reads are typically compared against a large reference genome. For example, the human genome contains around 3.2 billion base pairs (bp).

Modern aligners can be categorized by their method of detecting exact matches in a reference sequence. Most aligners utilize either hashing algorithms or the Burrows-Wheeler Transform (BWT). The BWT was originally used as a compression algo-
2.3. SEQUENCE ALIGNMENT

In practice, hash-based aligners can achieve better performance but have a larger memory footprint for whole human genomes [20]. The aligners MAQ [21], and ZOOM [22] hash the reads and fit these hashes to the reference genome. Other aligners such as SOAP [23], SNAP [24], Novoalign [25] hash the reference genome, and can quickly look up at what location in the reference the reads have exact matches. Popular BWT-based aligners include BWA [26], Bowtie [27] and SOAP2 [20]. BWA-MEM is our focus of study; It is one of the most popular open-source aligners, advocated in the GATK pipeline and well documented. A more thorough analysis of BWA-MEM is presented in Chapter 4. Most modern aligners still rely on more accurate fine-grained alignment after identifying exact matches in a reference sequence. The dynamic programming Smith-Waterman algorithm is often used.

2.3.2 Smith-Waterman

The Smith-Waterman algorithm performs local sequence alignment. It attempts to find the most optimal alignment based on a certain scoring system. When comparing the output of Smith-Waterman over multiple sequences, the highest scoring alignment will have the highest similarity and the pair of sequences are most likely related. Given two strings $a$ and $b$ with respective lengths $N$ and $M$, similarity in this case is expressed as the edit distance between $a$ and $b$, i.e. the number of insertions, deletions or substitutions of characters to transform $a$ into $b$ (or vice versa). DNA sequences are represented as strings over the alphabet $\Sigma = A, C, G, T$. The algorithm is a dynamic programming algorithm, a method for solving complex problems by breaking them down into smaller subproblems. The following equations describe how to solve the problem, by filling out similarity matrix $H$.

**Initialization**: 

$$
\begin{align*}
H(i, 0) &= 0 \\
H(0, j) &= 0
\end{align*}
\quad 0 \leq i \leq N \quad 0 \leq j \leq M \tag{2.1}
$$

$$
H(i, j) = \max \left\{ \begin{array}{l}
0 \\
H(i-1, j) + \alpha, \\
H(i, j-1) + \beta, \\
H(i-1, j-1) + s(a_i, b_j)
\end{array} \right\}, \quad i > 0, j > 0. \tag{2.2}
$$

In this equation, $\alpha$ denotes the score of an insertion penalty, $\beta$ of an deletion penalty and $s(a_i, b_j)$ describes the score of a match when the characters are equal, and the score of a mismatch when the two characters are not equal. For each cell in the matrix, the equation determines whether an insertion, a deletion or a match/mismatch leads to the optimal alignment. After all cells have in the matrix have been calculated, the optimal alignment is known. The highest occurring value in matrix $H$, referred to as the similarity score, determines the end of the local alignment. The optimal alignment can be obtained by storing a pointer to the previous cell in the similarity matrix, depending on the direction of movement, and backtracing the path. In practice, most applications
use the S-W algorithm to compute merely the similarity score in a first round, and trace
the corresponding alignment in a second round.

In the following example we show the alignment of sequences $a = AGTAC$ and
$b = AGTC$, when $\alpha = \beta = -1$ and

$$s(a_i, b_j) = \begin{cases} 
-2 & \text{if } a_i \neq b_j (\text{mismatch}) \\
+1 & \text{if } a_i = b_j (\text{match}) 
\end{cases}$$

(2.3)

From the example, it follows that the similarity score is 3, and two optimal
alignments exist. By tracing back the path in the score matrix, the alignment(s) can be
found. One of the optimal alignments is shown in Table 2.2. The resulting path through
the matrix for one of the sequences is referred to as cigar. E.g. the cigar for sequence $b$
is $3M1D1M$, this cigar string indicates three matches, followed by one deletion and one
match.

The most widely used mechanism for scoring alignments uses an affine gap penalty,
introduced by Gotoh in 1982 [28]. This mechanism introduces a penalty for opening a gap
(insertion / deletion) and a separate penalty for extending a gap. This mechanism allows
for longer insertions and deletions, that are commonly found in two related biological
sequences; long gaps can be caused by a single mutational event. Additional score
matrices are used to select between inserting a gap or extending one.

Filling out the similarity matrix has runtime $O(NM)$, where $N$ and $M$ are the
lengths of a pair of input sequences to the algorithm. It has been proven that this is
indeed the most optimal algorithm for finding the edit-distance (or similarity) between
two sequences [5]. For NGS data, short reads are typically between 35 and 500 bp, while
a human reference genome is around 3.2 Gbp. Aligning a sequence to a human genome

---

Table 2.1: Smith-Waterman similarity matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>T</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>−</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

$H = \begin{pmatrix} 
- & A & G & T & A & C \\
- & 0 & 0 & 0 & 0 & 0 \\
A & 0 & 1 & 0 & 0 & 1 \\
G & 0 & 0 & 2 & 1 & 0 \\
T & 0 & 0 & 1 & 3 & 2 \\
C & 0 & 0 & 0 & 2 & 1 
\end{pmatrix}$

Table 2.2: Optimal alignment between two sequences

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>T</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>A</td>
<td>G</td>
<td>T</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>b</td>
<td>A</td>
<td>G</td>
<td>T</td>
<td>-</td>
<td>C</td>
</tr>
</tbody>
</table>

score: +1 +1 +1 -1 +1 = 3
takes enormous amounts of time. E.g. if we align a read of 100 bp with a reference genome of length 3.2 Gbp, even on a fast specialized platform (RIVYERA S3-5000, with 128 FPGAs), it would take a number of days to align 10M reads with the naive S-W algorithm. For those reasons, it’s not feasible to directly apply this algorithm to align NGS data. Most often, heuristics are supplemented to limit the search space of the dynamic programming.

### 2.4 Smith-Waterman accelerated prior art

Attempts for accelerating the Smith-Waterman algorithm generally spend effort into parallelizing computation. Two types of parallelism are often exploited in accelerated platforms. With data parallelism, multiple Smith-Waterman computations are carried out in parallel on independent data. Inter-task parallelism or wavefront parallelism is applied to accelerate the computation of Smith-Waterman by exploiting parallelism in the computation of the similarity matrix $H$, i.e. multiple cells in the similarity matrix can be computed in parallel. The performance of a platform is measured in terms of the number of cells in the matrix that are calculated per unit of time. E.g. 10 GCUPS denotes 10 billion cell updates per second. This measurement is independent of clock frequency or number of instructions and is therefore the figure of merit to compare different platforms. We’ll review the acceleration of the Smith-Waterman algorithm on different platforms, such as the GPGPU, FPGA and the CPU.

#### 2.4.1 GPGPU

In 2011, Hasan et al. accelerated the search of finding the highest ranking matching proteins in a database on a CUDA enabled GPU with their framework titled DOPA [29]. Since their database of proteins consists of multiple sequences, data parallelism is exploited by employing the Single Instruction Multiple Threads (SIMT) model: multiple threads on a GPU execute concurrently using a single instruction. The input sequences are distributed among the multiple streaming processors on a GPU, making full use of the hardware’s capabilities.

To the best of knowledge, CudaSW++ 3.0 is the fastest implementation on a GPU for searching protein databases to date [8], with a reported performance of 119 GCUPS. It is a hybrid solution, in that it concurrently puts the CPU and the GPU to work. Just like DOPA, this framework employs SIMT to distribute the workload amongst the multiple processors on a GPU. Furthermore, new CUDA SIMD instructions are used to exploit more parallelism. Known SIMD acceleration techniques are also used on the CPU.

#### 2.4.2 FPGA

Due to the dependencies shown in Equation [2.2], the cells in score matrix $H$ cannot all be computed in parallel. A common method referred to as wavefront is used to compute cells that show no dependency in parallel. Figure [2.3] depicts the Smith-Waterman similarity matrix [2]. Each iteration, cells on the anti-diagonal line can be computed in parallel
with respecting the data dependencies. E.g. in the fifth iteration, cells marked with a (5) can be computed, and in the next iteration cells marked with (6) can be computed.

This wavefront parallelism can be easily implemented on a FPGA using a structure referred to as Systolic PE Array. Processing Elements (PEs) are cascaded into an array, and each PE is responsible for executing the equations in 2.2 and filling out one cell of the similarity matrix at a time. Each PE is mapped to one of the characters of sequence $a$. All of the characters of sequence $b$ are shifted through the PE array. Systolic refers to the wave-like propagation through the score matrix, in the same way as the heart pumps blood through the human circulatory system. The PE array suffers from startup costs, i.e. not all PEs are utilized in the first and last iterations. As a result, this system is more effective for longer sequences, where the startup costs are negligible.

Yamaguchi et al. claim that the FPGA platforms are twenty-fold more energy efficient than GPUs or CPUs in computing the S-W algorithm, with 16 GCUPS/W for FPGAs and 0.08 GCUPS/W on a GTX295. Therefore we claim FPGAs are superior to GPUs in terms of energy efficiency for the S-W algorithm.

Very recently, Chen et al. have proposed a novel Smith-Waterman acceleration en-
2.4. SMITH-WATERMAN ACCELERATED PRIOR ART

gine for NGS sequence alignment [31]. Instead of using a systolic array for exploiting parallelism in the computation of the similarity matrix, they apply a GPU-like approach of instantiating a large number of single PEs. They claim a speedup of 6x over traditional systolic array approaches, and a speedup of 26.4x over a 24-thread Intel Haswell Xeon server. However, in contrast to our work, they only report on the acceleration of the S-W application kernel. Prior to the S-W kernel, large number of sequence pairs are collected and streamed via the PCIe bus.

2.4.3 CPU

With the Pentium III, Intel offered the first support for Single Instructions Multiple Data (SIMD) instructions, with its Streaming SIMD Extension (SSE). It includes multiple ALUs in the pipeline of the processor, working on narrow operands in wide registers (128 bits). With the SSE instruction set, the processor is able to operate on these narrow operands (8 bits or 16 bits) in parallel. Efforts in literature have focused on adopting SIMD instructions for acceleration of the S-W algorithm [32][33][34], and in particular the mapping of cells in the similarity matrix to the SIMD registers.
3.1 Introduction to Power8 and CAPI

IBM’s latest processor, the POWER8 CPU, is targeted towards big data and analytics applications. These types of applications are often characterized by copious amounts of data, and the challenge to structure the data and extract useful information places high demand on the bandwidth of the memory but also on the single threaded performance. The big data industry in general is moving towards commodity servers and scale-out, adding more computing power to an application by adding cheap hardware instead of scaling-up, by upgrading the CPU or adding more memory. IBM delivers a high-end processor with a new set of features with their latest POWER8 processor.

As one of the new features, POWER8 introduces Coherent Accelerator Processor Interface (CAPI), that allows specialized hardware to act as an accelerator to the POWER8 system. The attached accelerator has coherent access to the shared memory of the system, and is a peer to the POWER cores.

Computational tasks can easily be offloaded to an accelerator. Since accelerators are optimized for a specific computational task, they can be more energy efficient than a general CPU. Thus the additional accelerator can greatly improve the performance of applications running on the POWER8, while marginally increasing the energy consumption.

CAPI has been made available through the OpenPower community as of October 2014, and is supported by the FPGA vendors Altera and Xilinx.

3.2 CAPI architecture

Figure 3.1 depicts the architecture of the Power8 CPU and the CAPI architecture. Power8 is a SMP processor, containing up to 12 processor cores per chip, and with SMT8, supports up to 96 threads per chip. All cores have access to shared memory through the SMP bus. The accelerator has the same virtual address space as normal cores, and has full access to it. It operates on memory in the same way as a Power8 core (load/stores), and can therefore be considered as a peer to the cores.

CAPI is the interface, and describes the interaction between the processor and the accelerator. A dedicated area of the Power8 chip, The Coherent Attached Processor Proxy (CAPP), and a service layer on the accelerator (PSL) enable the interface.

The CAPP maintains a directory of cache lines for the accelerator and provides coherency by snooping the bus on the POWER8 CPU [35]. Data between the accelerator and the processor is transferred with the CAPI protocol. The physical layer in this protocol is a PCIe Gen3 link. The PHB provides connectivity between the CAPP and the PCIe link, and decodes the CAPI protocol messages and forwards it to the CAPP. The
CAPP in conjunction with the PHB handles the virtual to physical address translation, forwards data across the SMP bus, provides interrupts to the processor and provides coherency.

### 3.2.1 Accelerator side

The CAPI configuration on the accelerator consists of a Power Service Layer (PSL) and one or more Accelerator Function Units (AFUs). The AFU contains the custom acceleration logic. The PSL exposes a predefined interface to the AFU, and it contains an MMU to handle address translation (on the accelerator side). Furthermore it contains a copy of the cache directory (of 256 kB). Coherency of this cache is handled by the CAPP and the PSL.

The PSL provides a number of interfaces to the AFU that each work independently. AFUs can issue memory operations through the command interface \[30\]. The command interface offers various read/write commands, allowing for various options of coherency, or bypassing the caches in the PSL and CAPP entirely. Each write or read command is labeled with a tag. The buffer interface handles the movement of the data. The PSL reorders commands in a high-performance order, and responses to these commands might not arrive in the same order as issued. The response interface is notified whenever a command completes, where the tag is used to identify the command. The control interface can be used to control the state of the AFU. And finally the MMIO interface allows to write or read registers inside the AFU, but can be used for other purposes.
3.2.2 Data movement

During a read operation, the following steps describe the course of action for various elements in the CAPI architecture. The AFU issues a read command, where it specifies the address and the number of bytes to read. In case the read hits the cache in the PSL, the PSL immediately returns the data on the buffer interface and notifies the AFU. In case of a cache miss, the PSL attempts the virtual to physical address translation, and proceeds to signal the operation on the PCIe link. The PHB decodes the PCIe command, forwards it to the CAPP, which in turn issues the command on the memory bus. This is picked up by source that contains the cache line and returned to the AFU through the same path.

If the AFU attempts to write to a cache line, the PSL must have ownership of the line. It checks in the cache directory whether ownership is present and in modified state, if this is the case then the write command is completed directly and responded to appropriately. If it’s not present, the CAPP access the SMP bus and gains ownership of the line, the store from the AFU is then allowed to complete.

3.3 Specialized hardware

CAPI allows specialized hardware to act as a coprocessor to the POWER8. Specialized hardware exists in the form of custom design circuits, referred to as ASICs (Application Specific Integration Circuit). ASICs are hardwired to perform a certain task, and can greatly improve performance and energy efficiency over a CPU. However, ASICs have not seen widespread application in data centers due to the non-recurrent engineering costs of development and production. Furthermore, ASICs cannot be reprogrammed and can be out-dated whenever new knowledge about the computational task is available.

A Field Programmable Gate Array (FPGA) also offers customization, and can implement specific algorithms directly on hardware but one step less directly than an ASIC. However it provides flexibility through reprogrammability. Most FPGAs contain logic blocks and routing channels between those blocks. Each block typically contains a number of Look-Up Tables (LUTs), adders and D-type flip-flops, depending on the vendor. These logic blocks are capable of implementing any logical function. The reprogrammability of the FPGA comes at the cost of reduced clock frequencies and lower logic density compared to an ASIC.

FPGAs are typically programmed with a Hardware Description Language (HDL) describing the RTL design. This is a time-consuming, error-prone and complex engineering task. Major FPGA vendors have recently released C-to-gates tools, allowing automated development of hardware using the C programming language. Often, these tools can generate designs with comparable or superior performance compared to RTL designs [37].

The reconfigurability and energy efficiency turn an FPGA into an excellent specialized accelerator. Recently FPGAs have shown their value in a number of fields. Traditionally, FPGAs perform well in computationally heavy tasks that exhibit high arithmetic intensity, where parallelism can be applied to accelerate computation. Examples include encryption, compression, image processing, and analytics.
With heterogeneous computing systems, we indicate systems containing more than one kind of processor. Specifically, reconfigurable computing systems refers to heterogeneous systems combining one or more reconfigurable accelerators (such as an FPGA) with a microprocessor.

Reconfigurable computing systems, e.g. FPGA+CPU, have been deployed successfully for accelerating workloads in large-scale systems in various projects in the industry. Microsoft has attached an array of FPGAs (connected in a torus) to their high-end servers to accelerate parts of the Bing Web Search Engine [38]. IBM incorporated FPGAs to accelerate the throughput of the GZIP compression / decompression algorithm [39]. Memcached, a distributed memory caching system, has been accelerated by enabling the FPGA as an intermediate between the Network Interface Controller (NIC) and the CPU [40].

3.4 Properties of CAPI

In this section we will describe the advantages that CAPI offers with respect to a regular PCIe-attached accelerator.

3.4.1 Low latency, high bandwidth data access

In the context of an accelerator, three types of memory exists.

1. Device Memory, memory in the form of DRAM or SRAM on the accelerator

2. Pageable Host Memory, memory on the host device, can be paged out by the OS. Allocated with malloc, calloc, new etc.

3. Pinned Host Memory, cannot be paged out by the OS.

Pinning the memory tells the OS to keep the allocated space in physical memory, while pageable host memory can be swapped out. The advantage of Pinned Host Memory is that it can be tunneled over the PCIe link by DMA engine without OS involvement, while Pageable Host Memory normally requires OS involvement.

In Figure 3.2 we see the typical workflow of invocation a kernel on a PCIe attached device. The device driver first copies to the pinned memory, or the region of memory where the data resides is pinned. Then the kernel on the accelerator is invoked using a device driver call. The latencies of the device drivers instructions and copying the memory is platform dependent. Recently released frameworks for PCIe attached FPGAs report latencies of up to 20 $\mu$s for transferring data on Windows 7 [41]. Overhead for synchronous kernel invocation on GPGPUs with Cuda has been reported to be between 10 and 20 $\mu$s [42]. A number of the latest Intel high performance CPU’s such as the Intel Xeon E3 also feature on-chip PCIe support. A recent study has shown remarkably improved latencies (sub $\mu$s) for the latency of PCIe attached I/O devices [43].

The latency reported in Table 3.1 is the time between initiating a data movement by the AFU, and completion of this operation reported to the AFU, or round-trip latency to shared memory as seen from the accelerator. We have taken the average over a large number of memory transactions where the data requested was not cached. If we
3.4. PROPERTIES OF CAPI

Table 3.1: Performance metrics of CAPI on POWER8 from an accelerator’s perspective

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average read latency to shared memory</td>
<td>0.86 µs</td>
</tr>
<tr>
<td>Average write latency to shared memory</td>
<td>0.84 µs</td>
</tr>
<tr>
<td>Latency to PSL cache</td>
<td>120 ns</td>
</tr>
<tr>
<td>Maximum Bandwidth to shared memory</td>
<td>3.9 GB/s</td>
</tr>
</tbody>
</table>

use constructs in shared memory to signal the accelerator, we obtain a latency that is an order of magnitude smaller than those reported above, where OS involvement and device drivers are needed. This opens up the possibility for finer-grained acceleration, algorithms or applications with low arithmetic intensity.

CAPI enabled accelerators are connected through PCIe 3.0, a full-duplex bus that offers 985 MB/s per lane. The effective bandwidth of the PCIe 3.0 x8 is 7.8 GB/s. About 50% of that bandwidth is provided to the AFU. Communication between PCIe devices involves the transfer of packets, packet overhead contributes to the reduced attainable bandwidth. Additionally, providing coherency and address translation by the PSL also causes traffic over the PCIe link.

3.4.2 Shared address space programming

The host processor and accelerator in a CAPI environment share the address space, i.e. both the processor and the accelerator have coherent access to host memory. The PSL in conjunction with the PHB provide the address translation for the accelerator. CAPI allows the accelerator to issue memory operations similar to a thread running on a POWER8 core.

Instead of copying data structures to a predefined address in memory, the accelerator can dereference the pointer and read in the data structure without OS involvement, this is commonly referred to as zero copy. Pointer dereferencing facilitates programming the accelerator. The accelerator has direct access to dynamic data structures typically
found in software, such as linked lists. With a traditional PCIe accelerator, these typical software structures would need to be converted into a continuous data structure suitable for an accelerator.

3.4.3 Coherent cache

As explained previously, a small cache is available to the accelerator, and coherency is provided by a dedicated unit on the processor. An accelerator can benefit from this in the same way as a regular core benefits from a cache. Applications that show strong locality of reference are able to have lower latency access to data. As shown in Table 3.1, for the accelerator, latency to the PSL cache is significantly lower than to shared memory on the host.

(1) The cache is only beneficial to the accelerator if the AFU displays temporal locality. An AFU that only requires data once, such as a streaming application, does not benefit from a cache and possibly causes more bus traffic for maintaining coherency.

(2) The coherency is only required when during runtime, the cores and AFU access the same region of memory. I.e., if the AFU works on an isolated region of shared memory, and the cores work on a different region, there is no need for coherency of the cache. If no coherency is required, the AFU could make use of custom storage (e.g. registers) on the accelerator to exploit temporal locality.

3.5 CAPI applications

In the following sections we describe applications where CAPI proves beneficial. Note that these are just examples, and more applications might exist. In this thesis, we will demonstrate acceleration of computation as a possible application.

3.5.1 Low overhead storage

CAPI has already been proven successful as an accelerator for IBM’s storage system for Big Data applications. The IBM Data Engine for NoSQL leverages IBM’s FlashSystem 840. This Flash storage system is capable of delivering high IOPS, while being cheaper than DRAM. A CAPI attached FPGA is used to reduce the OS overhead involved with reading from the flash disk. Through CAPI, the Flash array proves a low latency and high performance data storage.

3.5.2 Acceleration of computation

CAPI can be employed to attach an accelerator to the CPU. We classify the interaction between an accelerator and the host / processor in the following two ways:

1. Off-load: the complete algorithm or application is executed by the accelerator.

2. Hybrid: a portion of the algorithm or application is executed on the host, and the another portion is executed on the accelerator.
In applications where latency to the accelerator is critical, CAPI proves to be beneficial. Making use of shared memory concepts, the overhead for synchronization and signalling between an accelerator and the CPU is significantly reduced. Since the processor is able to read from the accelerator at a low latency, tightly coupled off-loading becomes an option.

Due to the overhead associated with traditional PCIe transfers (copying data or pinning memory), a common technique to amortize the overhead is to aggregate small transfers into a batch. In certain applications, aggregating small transfers is not a possibility, or not efficient, due to the nature of the application. CAPI significantly reduces overhead of transfers and lowers the need for batching of data, allowing for more tightly coupled hardware and software. In order to achieve acceleration for finer-grained kernels, low latency communication is paramount.

In general, we are still limited in the speedup we are able to achieve in a hybrid system if we only accelerate an application kernel. This is expressed by Amdahl’s Law. Given:

1. $k$, the acceleration of a kernel: $k = \frac{\text{runtime}_{\text{onhost}}}{\text{runtime}_{\text{onaccelerator}}}$
2. $\alpha$, the fraction of the application that can be accelerated

The speedup $S$ we are able to achieve is given by:

$$S = \frac{1}{\alpha \cdot \frac{1}{k} + (1 - \alpha)} \quad (3.1)$$

The maximum speedup that is attainable is limited by $\frac{1}{1 - \alpha}$, as we let the runtime of the accelerated kernel approach 0. E.g. to get a speedup of an order of magnitude (10x), we need to accelerate the application kernel that takes up at least 90% of the application’s runtime.
4.1 BWA-MEM mapping algorithm

The Burrows Wheeler Aligner (BWA) has been released in 2009, and consists of numerous alignment tools. It is designed to run on commodity hardware, with a small memory footprint, and is open-source software. It has seen widespread usage in the bioinformatics industry.

It employs three algorithms for aligning reads, BWA-backtrack (or BWA-aln), BWA-SW and the most recent BWA-MEM. The first algorithm is specifically targeted towards short reads, of up to 100 bp in length. Short reads are characterized by a low amount of gaps and mismatches, and short read aligners exploit this by applying exact match or allowing limited gaps. Performance degrades quickly when allowing longer gaps, since the number of steps required to align a read increases exponentially with the number of mismatches permitted.

Long gaps occur more frequently in longer reads, and BWA-SW is permissive about gaps. BWA-SW supports reads of up to a few kilo bp in length. BWA-MEM succeeds both algorithms, and is recommended for longer reads, due to performance enhancements, better accuracy and additional features [6].

BWA-MEM is based on the seed and extend paradigm found in BLAST [19]. In the first stage, exact matches between the reads and the reference sequence are found. The length of the exact matches depends on a runtime parameter of BWA-MEM. This stage is referred to as seeding, and is further described in Section 4.1.1. In Section 4.1.3, we describe how BWA-MEM reports its results. The relation between the two stages, and the execution on a multi-threaded system is described in Section 4.1.4.

We argue that seeding is a memory intensive process, while seed extension is computationally intensive and therefore a good candidate for acceleration on a FPGA.

4.1.1 SMEM seeds

BWA is named after the search algorithm based on the Burrows Wheeler Transform (BWT) [7]. The BWT compressed files have a limited footprint on memory, requiring only about 3 GB for a full human genome. This is well within the bounds of commodity hardware RAM storage. This state-of-the art aligner carries out read mapping in two stages. Seeds are generated by finding Super Maximal Exact Matches (SMEM) between a read and a reference sequence. A Maximal Exact Match (MEM) is an exact match between two strings, that cannot be extended in a forward or backward direction. A SMEM is a MEM that is not contained in any other MEM [44]. Using the BWT,
an SMEM of length $l$ can be found in $O(l)$, independent of the length of the reference sequence. Only seeds that exceed a minimum length (19 by default) are processed further. BWA-MEM uses the FM-index \cite{15}, a data structure that allows for fast string matching on BWT compressed strings. Exact matching of a string is done by a backward search on the transformed string, which essentially performs a top-down traversal of the prefix tree of the string \cite{15}. In the case of BWA-MEM, the reference genome is transformed with the BWT. This backward search has a very irregular memory access pattern and it shows very poor locality. Therefore the backward search suffers very high cache and TLB misses, and is not able to make use of prefetching by the CPU \cite{16}.

Each read may contain multiple SMEMs, and each SMEM may have hits at multiple locations in the reference sequence. Some filtering is applied to reduce the number of seeds, as it reduces the computational work of the next stage.

### 4.1.2 Seed extension

In the next stage, seeds are extended in both directions of the seed. The sequences to the left and to the right of the seed in the read, named *query* are aligned with the respective matches in the reference sequence, named *target*. Figure 4.1 depicts such an extension.

Smith-Waterman, a dynamic programming algorithm is used to rank these alignments between *query* and *target* in this seed extension. Only the alignments that have a high enough ranking are recorded into the output. During seed extension, BWA-MEM also keeps track of the alignment score reaching the end of the *query*. If this score is within a certain margin of the best local score, this end-to-end extension is selected above the local alignment.

Figure 4.2 depicts the alignment process for one read. Generally multiple SMEMs are found per read, and each SMEM can have a potential match at multiple locations in the reference sequence. The collection of seeds for a read is referred to as a chain of seeds \cite{6}. In BWA-MEM software, a higher level function titled *chain to alignment* performs seed extension for all seeds in the chain. The operation *chain to alignment* selects a seed from the chain, and checks whether the seed is already contained in a previous alignment.

The initial similarity score in the Smith-Waterman algorithm $H(0, 0)$ is not zero as in Equation \cite{2.1}, but takes on the score of the exact match according to the score mechanism. E.g. if we start with an exact match of length $l$ and the score for a match
is $m$, the starting score $H_0$ would be $l \ast m$.

$H_0$ serves as an input score to the left seed extension. Subsequently, the resulting similarity score of the left seed extension is passed as an input for right seed extension. The best alignment for a read is selected, and recorded to the output.

Seed extension in software applies a number of heuristics to limit the search space of the Smith-Waterman algorithm. In practice, only around 45% of all cells in the similarity matrix are calculated. BWA-MEM applies banded Smith-Waterman, only cells within a certain distance from the diagonal are calculated. Cells outside the band have a low probability for a high score. Furthermore BWA-MEM employs a heuristics referred to as Z-dropoff. If the similarity score of current cell $H(i, j)$ is below a certain margin from the best score encountered, seed extension is stopped prematurely.

### 4.1.3 Output generation

The resulting alignment for a read is found by another Smith-Waterman operation, tracing back the path for the highest scoring alignment. The output of BWA-MEM is a Sequence Alignment Map (SAM) file, containing the reads that have been aligned, the
position they have been aligned to, and the resulting alignment including the mapping quality \[47\]. This output file can be processed in further steps of the NGS analysis pipeline.

### 4.1.4 Dividing workload amongst threads

The typical size of modern day FASTQ files, that serve as an input to BWA-MEM, of a whole human genome (WGS) with 30x coverage is roughly 200 GB in size. Coverage refers to the average number of times a single base is read during a sequencing run. The FASTQ format isn’t very space efficient, since it uses human readable text and stores the nucleotides and mapping quality in 8-bit format. The input data set is normally provided from disk and is processed in chunks in order to reduce memory footprint.

BWA-MEM supports multi-threaded execution, and does this by dividing the workload amongst threads. Each chunk is further divided in a subset of reads, and each subset is assigned to a thread. Figure 4.3 shows the assignment of the different stages to the threads. Each thread executes the SMEM generation and the aligning of seeds directly after each other for a single read, before processing the next read.

After all reads have been extended by all threads, threads are synchronized by a barrier operation. For clarity, the figure has been simplified. During a run, each thread handles thousands of seed extensions and SMEM generations per chunk.

### 4.2 Analysis and profiling

In this section we will present the results of system-wide profiling of BWA-MEM on a Power8 system. BWA-MEM 0.7.10 for Power8 is available at biobuilds \[48\], and is distributed as a binary. We use a profiling tool termed *perf* to collect CPU cycle information while running BWA-MEM. *perf* collects its data on a per-process basis, and does not require instrumentation in the binary or recompilation of code. Instead it samples the call stack at a frequency of 99 Hz, and has little impact on the execution.

We profile the application for input sizes of various length between 100 bp and 250 bp on a Power8 system. Reads shorter than 100 bp are typically aligned with the specific short read aligner BWA-aln \[26\]. BWA-MEM is designed for longer reads, and since the industry is slowly moving towards longer reads, we also profile for 250 bp. Both reads are generated with *wgsim*. Real data might show a slightly different profile.

Sequence alignment is an embarrassingly parallel process with no data dependencies between individual reads. Straight-forward data parallelism can be applied, and each thread can be assigned to each of the sequenced reads. BWA-MEM is a multithreaded application making use of POSIX threads. The Power8 system that has been used in the experiment contains 20 physical cores. With SMT8, 160 logical cores are presented to the OS. We profile the application with a number of threads ranging from 1 to 160.

Figure 4.6 and 4.5 show the profile for 100 bp and 250 bp reads respectively. The top 5 functions in terms of CPU cycles are shown. The y axis denotes the fraction of CPU cycles that are spent in a certain function. All functions prefixed with *ksw* are Smith-Waterman related functions. The function *ksw_extend2* denotes the seed extension in BWA-MEM, and *ksw_global* is used to generate the alignments. The functions prefixed
4.2. ANALYSIS AND PROFILING

Scaling with respect to input characteristics When we compare Figure 4.5 with Figure 4.6, it stands out that \textit{ksw\_extend} (seed extension) has a more prominent role in the longer reads. For 100 bp reads, the processor spends 35% of its cycles in \textit{ksw\_extend}, and 43% while dealing with 250 bp reads, on a single-threaded basis. Also the other S-W related function, \textit{ksw\_global} occupies a larger fraction of CPU cycles. We can directly relate this to the scalability of the S-W algorithm, the runtime is \(O(nm)\) when the two lengths of the pair of input strings are \(n\) and \(m\). Since both input strings scale linearly with increasing read lengths, the runtime of

Figure 4.3: Multi-threaded organization of BWA-MEM
the S-W algorithm scales quadratic in the worst case. Due to the use of heuristics (see section 5.1.1), seed extension scales slightly better. Empirically this is shown in Figure 6.4. SMEM generation scales linearly with the read length, due to the efficient search of exact matches. Thus we observe a difference between Figure 4.6 and 4.5.

**Scaling with respect to number of threads** The Power8 system can be configured to support multi-threading, known as *simultaneous multi-threading* (SMT). With mode SMT8, each core can support up to 8 threads simultaneously, and to the OS 8 virtual cores are presented.

(build 0.18)
4.2. ANALYSIS AND PROFILING

As we scale up the number of threads, we can see an increase in the portion of ksw-extend2 starting from 16 threads. We believe this is due to limited computational resources on a core, and threads are fighting for the resources. Since the BWT related functions are very memory intensive and show many cache misses, the threads handling BWT can hide latency by switching context and are better able to make use of the multi-threading capabilities of Power8.

We have deduced that the seed extension is becoming the bottleneck on the Power8 when we scale up the number of threads or when we move to larger reads. As we have seen in Section 2.4.2, the FPGA presents itself as a energy-efficient and high-throughput platform for the Smith-Waterman (predominant in seed extension) algorithm.

The FPGA is less suitable for the task of SMEM location. This task uses the FM-index and suffers from irregular memory access patterns and high cache and TLB misses on the CPU. However through simultaneous multi-threading, the CPU is able to hide cache misses. In his Master’s thesis, Padmanabharao ported the SMEM generation of BWA-MEM to four FPGAs on a Convey Hybrid-Core system [49]. Little success has been booked here, as he reported a maximum speedup of 0.1x over a single-threaded execution on the host processor. He states that memory access remains a bottleneck in SMEM generation, and hiding the latency to memory proves a difficult task.

However, a recently released a high-performance reconfigurable computing system has been used to accelerate bisulfite sequence alignment [50]. This accelerated system uses the FM-index to quickly align reads, but only allows up to two mismatches. The reconfigurable architecture has been implemented on a Maxeler MPC-X1000, consisting of 8 Altera Stratix V FPGAs. In their implementation, the random access pattern of the FM-index is the limiting factor of performance [50]. Still, they report an order of magnitude increase in throughput compared to 16 threads on a Intel Xeon E5-2650 running alignment tool soap2.

Figure 4.6: Profile bwa-mem synthetic single-ended 100 bp reads
This chapter will describe the design and implementation of the accelerated S-W engine on the FPGA and the integration of this accelerator with BWA-MEM. The FPGA offloads the execution of seed extension in BWA-MEM. This dynamic programming operation is compute intensive for long DNA sequences and forms a computational bottleneck on modern processors. Seed extension burns a significant portion of the CPU cycles, and offloading it to the FPGA can greatly improve performance of BWA-MEM.

In Section 5.2 we describe the design of the accelerator, that aims to achieve a high throughput and low latency. The implementation of the design and the resource utilization on the FPGA is described in Section 5.3. In Section 5.4 we describe the use of shared memory for integrating BWA-MEM with the acceleration engine on the FPGA, leveraging the advantages of CAPI. Design alternatives, both for FPGA implementation and application integration are discussed in Section 5.5.

5.1 Architecture design

Since we have determined that seed extension is computationally intensive, our focus is on designing a highly parallel accelerator for seed extension, a modified Smith-Waterman (SW) algorithm that uses dynamic programming to find the similarity between two DNA sequences. The accelerator must not form a bottleneck itself, and should have a high enough throughput to keep up with seed generation on the CPU.

5.1.1 Design motivation

In order to achieve this high throughput on the FPGA, the design is made highly parallel, exploiting both data parallelism and parallelism found in the dynamic programming algorithm. The architecture is designed with the following observations from the analysis of BWA-MEM.

Data parallelism BWA-MEM is designed with data parallelism in mind. In current NGS flows, billions of short reads are generated. Each short read can be processed independently, allowing for enormous data parallelism. BWA-MEM software can be executed multithreaded, each thread processing a subset of short reads. An FPGA’s reconfigurability can be utilized for this enormous data parallelism. To facilitate the data parallelism on the FPGA as well, numerous independent units that perform the S-W algorithm can be instantiated, leveraging the parallelism of an FPGA.

Inner-task parallelism and varied-size input Each S-W task can be parallelized as well. As explained in Section 2.4.2 all elements on the anti-diagonal in the S-W

5
matrix can be computed in parallel, often referred to as "wavefront". In hardware, the S-W algorithm is often implemented using a systolic PE array.

**Varied-size input** Figure 4.4 shows the distribution of the length of input sequence when 100k reads of 100 bp are considered. In general, only SMEMs satisfying a minimum length $s$ are reported to the seed extension stage. The remaining part of the read of length $l$ is used for seed extension. Consequently we obtain sequences for seed extension that have length $l - s$. In this figure, SMEMs are generated with a minimum seed length of 19. Thus the maximum occurring length for extension is $100 - 19 = 81$. The occurring lengths are evenly distributed. The accelerator design needs to account for this.

## 5.2 Architectural overview of accelerator

A top-level overview of the architecture is depicted in Figure 5.1.

The Power Service Layer (PSL) is a predefined logic block provided by IBM. It contains an interface to the PCIe link and exposes a number of interfaces to the Accelerator Function Unit (AFU). The AFU is supplied by the author and implements the Smith-Waterman acceleration unit. To provide a high-throughput accelerator, the AFU accommodates multiple S-W cores, that carry out the Smith-Waterman algorithm. With S-W core, we denote a systolic PE array.

The DMA wrapper, wrapping multiple PSL interfaces, facilitates data movement to/from shared memory. It has multiple read and write engines and is capable of reading in or writing multiple cache lines autonomously and in parallel.

The MMIO block accepts all MMIO operations, and forwards all operations to the rest of the design. The operation of accelerated seed extension is described below.

From the host side, the processor populates a Work Element Descriptor (WED) structure in shared memory, containing the parameters for seed extension and the sequences being aligned, along with a pointer where results should be stored. The pointer to this structure is sent to the AFU with an Memory Mapped I/O (MMIO) operation. These MMIO operations are buffered in a FIFO, and the pointer to the WED is forwarded to the DMA wrapper through this FIFO. When the FIFO has a pointer available, an available read engine requests the data from host memory.

Since the rate of incoming might exceed the computational capabilities of the systolic arrays, the incoming data is buffered in another FIFO. A round-robin arbiter inspects the state of each S-W core, and assigns the incoming task to an idle core.

When a S-W core has completed the computations, it raises a flag and presents the output of the seed extend operation. The output includes the maximum similarity score of the alignment, the similarity score of the global alignment and the matching locations in the sequence where the scores are found.

The results are stored in a FIFO supplying the write engines. This FIFO is necessary in case the collection of S-W cores temporarily generates results at a rate exceeding the capabilities of the write engines. The write engines write the results to shared host memory at the address indicated by the pointer in the WED.
Meanwhile, the host processor is polling the status flag in the result structure at the same address.
5.2.1 Smith-Waterman computational units

In this section, we will discuss the impact of the length of a systolic array on the resource utilization on the FPGA, and the number of cycles it takes to complete the Smith-Waterman operation. The input to this algorithm is a pair of input sequences \textit{query} and \textit{target} with respective lengths \(m\) and \(n\), and \(P\) denotes the length of the systolic array. For an explanation on the systolic array, see Section 2.4.2.

Various configurations that map the Smith-Waterman algorithm on a systolic array have been proposed in literature, we distinguish three cases:

1. A systolic array that covers the length of one of the sequences, \(P = \min(m, n)\). This configuration obtains maximum degree of parallelism and leads to the least amount of cycles to compute the algorithm. However, this configuration also leads to the largest area utilization, since generally the area is proportional to the length of the systolic array.

2. A smaller systolic array is used, that does not cover one of the sequences, \(P < \min(m, n)\). The S-W similarity matrix has to be computed in multiple stages. Generally, the similarity matrix is divided into smaller partitions, and each partition is processed by the systolic array [30].

3. A systolic array of length \(P = 1\), consisting of a single PE. Only a single cell of the score matrix can be computed per cycle. The wavefront parallelism in this case is discarded. As we shall see, this configuration requires the least amount of area out of the three cases, but also requires the most cycles for computation.

We will now determine the required area and the latency of the computation for various configurations. For simplicity and without loss of generality, we assume that each PE is able to compute one cell in the Smith-Waterman score matrix \(H\) per cycle. Furthermore, we assume that initializing a PE array of length \(P\), populating it with one of the two sequences, takes \(P\) cycles (due to the cascading of the PEs). A single PE requires no initialization. Since each new S-W computation requires initializing the systolic array, we include this in the total cycles for computation. We assume the required area on the FPGA for the systolic array is proportional with \(P\), in practice control of the array adds considerable area. The effective utilization \(h\) of the PEs is defined as the ratio of PEs in the array that are joining in the calculation and length of the systolic array. This expresses how effectively the resources are used and is computed as:

\[
h = \frac{m \times n}{2 \times P + n} \times \frac{1}{P} \tag{5.1}
\]

After the systolic array has been initialized (taking \(P\) cycles), each PE has been populated with a character of the query sequence with length \(m\) and the PE array can the process the seed extension. Producing the alignment takes \(P + n\) cycles, and not \(m + n\), since the target sequence has to traverse the entire systolic array.

Table 5.1 depicts the cycles needed for computation, and the utilization of the systolic array for various configurations. We consider the S-W operation for various lengths of the query and target sequences, since profiling has shown that length of the query sequence...
Table 5.1: PE array configuration utilization

<table>
<thead>
<tr>
<th>array length</th>
<th>80x100 cycles</th>
<th>area</th>
<th>utilization</th>
<th>40x50 cycles</th>
<th>area</th>
<th>utilization</th>
<th>10x20 area</th>
<th>cycles</th>
<th>utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>260</td>
<td>80</td>
<td>37.5%</td>
<td>210</td>
<td>80</td>
<td>12%</td>
<td>80</td>
<td>170</td>
<td>1.5%</td>
</tr>
<tr>
<td>40</td>
<td>360</td>
<td>40</td>
<td>55%</td>
<td>130</td>
<td>40</td>
<td>38.5%</td>
<td>40</td>
<td>90</td>
<td>5.5%</td>
</tr>
<tr>
<td>1</td>
<td>8000</td>
<td>1</td>
<td>100%</td>
<td>2000</td>
<td>1</td>
<td>100%</td>
<td>1</td>
<td>200</td>
<td>100%</td>
</tr>
</tbody>
</table>

is distributed evenly, as shown in Figure 4.4. The column 80x100 denotes two input sequences with length 80 and 100. In the most optimal case, the largest configuration is only able to achieve a utilization of 37.5%, this is caused by the start-up costs and cool-down costs of the pipelined nature of the systolic array. For example, regardless of the size of the systolic array, in the first and last cycle, only one matrix cell can be computed. Even though the large systolic array is heavily underutilized, it outperforms the single PE by 30X for the 80x100 operation. The utilization for the largest systolic array drops rapidly for smaller sequences, since it can no longer obtain a large degree of parallelism. The single PE always achieves 100% utilization, since it’s always being put to work. It clearly needs more cycles to perform the S-W algorithm, but is 80X smaller in terms of area than the largest configuration. The length of the systolic presents a clear area / performance trade-off.

If we constrain the resources on the FPGA and include multiple systolic PE arrays in the design, a design with multiple single PEs should give the best throughput, since we achieve 100% utilization of resources. However if we want to optimize for latency (number of cycles), this design is no longer optimal. In practice, our simplifications presented above in terms of area are not accurate. For example, a considerable area overhead per PE is incurred for control and data movement. A more sophisticated model is needed to derive the optimal configuration.

The Smith-Waterman core that has been used in our design was developed and described by Houtgast et al. [9]. It is implemented as a Linear Systolic Array, but employs a number of modifications, aiming to cope with the variable length input sequences and suitable for the workload of BWA-MEM.

Houtgast et al. propose early exit points, allowing the systolic array to report its output at intermediate stages in the array. This technique is referred to as Variable Logic Length (VLL) and aims to reduce the latency (in terms of cycles) of the PE array, and address the underutilization of the systolic array. This method prevents smaller queries having to traverse the entire array, leading to a smaller number of cycles for the computation.

Ideally, a PE array of a matching length should be instantiated for each occurring query length, leading to minimal latency. Due to limitations in FPGA resources, this is not feasible. The systolic array in Houtgasts’s design is not capable of computing the similarity matrix in multiple stages, i.e. a systolic array of length \( n \) only supports seed extensions where the length of the input does not surpass \( n \). Furthermore Houtgast describes the use of multiple PE arrays with different lengths, such that shorted reads can be handled by a smaller PE array and longer reads can handled by a larger array, and calls this technique Variable Physical Length (VPL). Clearly, the design space becomes
very large: we can instantiate multiple systolic arrays, each with a different length, and with multiple exit points at intermediate indices.

In our design, we instantiated multiple PE arrays, this allows us to increase the throughput of the design by handling multiple seed extensions in parallel. Furthermore, as will be shown in Chapter 6, this allows us to minimize the latency of the accelerated solution. We have not considered VPL and instantiated PE arrays with constant length. More optimal configurations have been reported in [9].

The size of the reads of the data input set also determines the maximum size of the query sequence in the seed extension operation. The size of the PE array needs to correspond to this maximum query length for functional correctness. An optimal design in the lengths of the systolic arrays is subject to the input set. In order to cope with different data input sets, we have generated multiple FPGA bitstreams with corresponding PE array sizes. Then depending on the input set, the FPGA can be flashed with any of the bitstreams.

5.2.2 DMA wrapper

The Direct Memory Access (DMA) wrapper acts as an intermediate between various PSL interfaces and the AFU and handles all data movement operations to shared host memory. It hides the complex details of the interaction between the command interface, buffer interface and response interface of the PSL. The DMA wrapper houses a variable number of read ports and write ports, and all ports act autonomously. Each port is capable of moving $N$ bytes to/from coherent host memory at an arbitrary address. Upon completion of the data movement, the port is notified by the response interface and the port can be used again.

The PSL only orders commands on a cache-line address basis. Order between commands involving different addresses is unpredictable [35]. Each port therefore reorders the incoming responses, based on the tag of the command.

The memory operations in this design are limited in size. The read operations consist of 3 cache lines (each cache line carries 128 bytes), and the write operation carries one cache line of data. The bandwidth of each port is directly related to the latency to shared memory and the number of bytes moved, since it does not accept new requests until the previous request has been served (by design). In order to make efficient use of CAPI’s maximum bandwidth, multiple ports have therefore been incorporated into the design. The command interface of the PSL only allows 1 command to be issued per cycle, and since write ports and read ports work in parallel, an arbiter decides which port is allowed access to the command interface.

The PSL contains a state machine for read and write operations, allowing to have 32 outstanding reads and 32 outstanding writes. It uses a credit system and whenever an command is issued, a credit is taken from the AFU. Each time a read or write command completes, the credit is returned to the AFU. Thus in total, 64 commands can be outstanding. In order to maintain a high bandwidth, the AFU aims to fully utilize the given credits.
5.3 Implementation

The design has a number of parameters that can be set during synthesis. We can vary the number of S-W computational units, implemented as a PE array, and the length of each PE array. $M \times N$ denotes a design with $M$ arrays with $N$ PEs per array. We have kept the length of each array the same. Houtgast et al. explore an optimal combination of lengths for the PE arrays [9]. The length of all PE arrays is lower bounded by the longest occurring query. Depending on the minimum seed length $k$, and length of the read $l$, this should be $l - k$. We will create a separate bitstream for the FPGA for reads with length 100, and one with an array of 250 PEs for 250 bp reads. In retrospect, as per the discussion above, the length of the arrays could have been reduced to $100 - 19 = 81$ and $250 - 19 = 231$ when considering a minimum seed length of 19.

The design has been described with VHDL, and constitutes of around 10K lines of code, of which around 3K constitute Houtgast’s design.

Altera’s Quartus II tool is used to synthesize the design. The design has been clocked at 250 MHz, without any timing violations in the AFU. We are restricted in the number of arrays that can be placed by the available resources on the FPGA. An Adaptive Logic Module (ALM) is Altera’s basic building block [51], consisting of 2 6-input LUT, two adders and four registers.

Table 5.2: FPGA resource utilization of 100 element PE array configurations

<table>
<thead>
<tr>
<th>Design</th>
<th>Resource utilization (ALMs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x100</td>
<td>38%</td>
</tr>
<tr>
<td>2x100</td>
<td>47%</td>
</tr>
<tr>
<td>4x100</td>
<td>63%</td>
</tr>
<tr>
<td>6x100</td>
<td>81%</td>
</tr>
<tr>
<td>8x100</td>
<td>89%</td>
</tr>
</tbody>
</table>

Table 5.3: FPGA resource utilization of 250 element PE array configurations

<table>
<thead>
<tr>
<th>Design</th>
<th>Resource utilization (ALMs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x250</td>
<td>50%</td>
</tr>
<tr>
<td>2x250</td>
<td>70%</td>
</tr>
<tr>
<td>3x250</td>
<td>89%</td>
</tr>
</tbody>
</table>

As shown in Table 5.2 and 5.3, the maximum amount of 100 element arrays that fit on the FPGA is 8, and this is 3 for 250 element arrays. In this design, the logic utilization scales linearly with both the number of arrays, and the number of PE per array. Quartus was unable to fit a larger design onto the Stratix V. Around 28% of the ALMs are taken up by the PSL block. A PE array of 100 elements utilizes around 9% of the ALMs, while an array of length 250 utilizes about 20%. Furthermore we use about 16% of available block memory bits (8.3M / 52.4M), mainly for implementing the FIFOs described in the design, and BRAM memory used in the DMA wrapper.
5.4 Application integration

We operate the FPGA in the dedicated process programming model, meaning that only one process can have a handle to the FPGA device. Other modes that CAPI provides are the PSL-controlled shared programming and AFU-directed shared programming, where access to the AFU is shared between multiple processes [3]. Through a kernel call, the OS gains access to the PCIe device and initializes the PSL, and resets the AFU and afterwards the FPGA is ready to commence shared memory communication.

The FPGA in the design is the master, and initiates loads and stores to memory through the command interface. There are two ways to notify the FPGA of new work: through a synchronization mechanism in shared memory, or through interrupt-like interfaces such as the MMIO interface or the control interface.

5.4.1 Notification of accelerator

IBM has introduced the term Work Element Descriptor (WED), a user defined structure that contains the parameters and pointers to data to be accelerated. The WED is interpreted by the AFU. Two shared memory constructs have been considered, and are depicted in Figure 5.2.

(1) In the first construct, a collection of WEDs is built, in this case we use a linked list, but similarly a regular array or circular buffer can be used. Initially, the pointer to the first WED element is communicated to the AFU using a MMIO or through the control interface. Whenever a thread wants to offload a seed extension task to the FPGA, the thread gains exclusive control over the linked list and adds its work item to the end of the list, and sets the valid flag to true. The AFU can iterate over the linked list, and whenever it reads in a descriptor that is valid, processes
the item. Whenever the job is not valid (the parameters and valid flag have not been set), the AFU keeps polling the same memory address.

(2) In the second construct, the linked list is not maintained in software. Instead each thread notifies the AFU of a job item through a MMIO request, with a pointer to the WED.

The PowerPC architecture defines a storage model for ordering access to memory that is called weakly consistent. The architecture can reorder instructions (OoO execution) for improved performance. However, the compiler nor processors have a way of knowing that the AFU might access the same shared memory. To cope with the weakly consistent model for the accelerator, correct memory ordering needs to be enforced manually in the same way as two cores dealing with a shared variable. I.e. we need to make sure the WED has been populated with valid data before it is added to the linked list in (1), or before an MMIO request is made in (2). This can be achieved with a synchronization (or barrier) operation. This operation ensures that all memory accesses prior to the synchronization have been completed.

5.4.2 Waiting for results

On the host side, after adding the job to the linked list or firing an MMIO, the thread is polling on the status flag in the result structure, also referred to as busy waiting or spinning. Since the processor is not doing any useful work while spinning, alternatives have been considered to alleviate the processor.

1. Sleep On Linux, nanosleep suspends the execution of the calling thread for a certain duration. This frees up resources on the core, usable by other threads. Since the minimum latency of off-loading the seed extension kernel is known, the thread can be put to sleep for this duration. Unfortunately, the resolution of this high-precision timer on this platform is not small enough (in the order of tens of microseconds), and severely impacts the latency of seed extension. Since latency is very critical, this option is not viable. If the latency of the accelerated kernel would be in the order of milliseconds, this would be an option.

2. Lower thread priority The PowerISA (Power Instruction Set Architecture) allows to change priority of hardware threads with a special NOP instruction. At first inspection, modifying thread priority did not seem to influence execution time of BWA-MEM. Further inspection is needed to identify the effects of lower thread priority.

3. Wait and reserve The PowerISA offers the wait and reserve instructions. The core holds a reservation to a memory address, and waits until this reservation is lost. Whenever an external force modifies this memory address, the reservation is lost and the core is notified and continues the execution of instructions. While the core is waiting for losing the reservation, other threads could take over execution on the core. Unfortunately, the instruction is not implemented in the Power8 microarchitecture and thus could not be evaluated. However, it does seem promising since the OS is not involved.

build 0.18
In conclusion, no better alternative is available than spinning.

The adaptations to the code-base of BWA-MEM needed to integrate the accelerator and the host were extremely limited. Only the calls to the software-based `ksw_extend` have been replaced with a blocking call to the accelerator-based `ksw_extend_capi`. Around 10 lines of code have been modified in the original BWA-MEM software to enable the accelerator. In this function call, the WED is populated with parameters and input data and the accelerator is notified. The corresponding thread is busy waiting for results.

### 5.4.3 Model of performance

Since BWA-MEM is a multi-threaded application, multiple threads request service from the FPGA. Each thread is exposed to a different input set and processes the input independent from the other threads. The threads are not running in lockstep. The processing time for the SMEM generation and seed extension will vary depending on the input, and therefore threads start seed extension at independent and varying points of time, as shown in Figure 4.3. Collectively, these requests arrive independent to the accelerator and are distributed exponentially in time. We can model the integrated solution as a queueing system. Each `ksw_extend_capi` request is buffered in the FPGA in a first in first out (FIFO) structure, and is served by the FPGA in the order in which the requests arrive. The service requests can be modeled as a Poisson process with arrival rate $\lambda$. The mean time for processing the seed extension job is $1/\mu \times (f \times \text{cycle}_{SW})$, and therefore $\mu$ describes the service rate or throughput of one PE array. In general, by increasing the number of threads of the accelerated solution, $\lambda$ is increased.

The behavior of such a queueing system can be described by queueing theory. A system with $c$ systolic arrays with length $P$ is denoted as $cxP$. For simplicity, we assume that all systolic arrays have the same length. A $1xP$ configuration behaves as a M/M/1 queue, and $cxP$ configuration, where $c > 1$, behaves as a M/M/c queue. Following from queueing theory, in a M/M/1 queue, the total time to process the request, including time spent in the queue, can be expressed as:

$$t_{\text{process}} = \frac{1}{\mu - \lambda} \quad (5.2)$$

The utilization of the systolic array in this system is expressed as: $\rho = \lambda/\mu$.

In a M/M/c queue, where we have $c$ PE arrays, the total time to process a seed extension is expressed as:

$$t_{\text{process}} = C(c, \lambda/\mu)/c\mu - \lambda + 1/\mu. \quad (5.3)$$

where $C$ represents Erlang’s C formula. In this case, the utilization rate is represented as: $\rho = \lambda/c\mu$. This equation only holds whenever the utilization rate $\rho = \lambda/\mu$ is smaller than 1. The total latency of the accelerated seed extension then can be defined as:

$$t_{\text{latency}} = t_{\text{process}} + t_{\text{overhead}} \quad (5.4)$$

The overhead consists of sending an MMIO request, populating the WED with parameters and copying the data, reading in the WED by the AFU and writing the results back to shared memory. We assume that the read latency and write latency to shared memory is negligible.
memory, $t_{relatency}$ and $t_{wlatency}$ respectively are constant. If the WED is $M$ bytes, and the results consist of $N$ bytes, and the bandwidth is $B$, the overhead can be expressed as:

$$t_{overhead} = t_{mmio} + t_{relatency} + M \times B + t_{wlatency} + N \times B \quad (5.5)$$

Finally we arrive at the estimated speedup:

$$S = \frac{1}{(t_{latency}/t_o) \times k + (1 - k)} \quad (5.6)$$

Where $k$ is the fraction of CPU cycles spent in $ksw\_extend$ and $t_o$ is the latency of $ksw\_extend$ experienced by a Power8 core. The overall acceleration of BWA-MEM in this model is thus dependent on two factors:

1. Overhead of moving the data to the FPGA. This is dependent on the infrastructure, this is inherent to CAPI.

2. Processing capabilities of the seed extension core.

In Chapter 6, we will measure the speedup achieved under this blocking model. We will show that the speedup closely matches Equation 5.6.

### 5.5 Design alternative: heterogeneous system with a Task Queue

The presented architecture required few changes to the source code to integrate the accelerator with BWA-MEM. However, the implemented design is non-optimal. Figure 5.3 shows how the task are divided between the processor on the left side, and the FPGA on the right side. While the seed extension is carried out on accelerator, the threads on the processor actively poll for the completion of this task. During the polling, no useful work is carried out by the threads, and this leads to underutilization of the processor.

![Figure 5.3: Blocking seed extension on FPGA](image1)

![Figure 5.4: Non-blocking seed extension on FPGA](image2)
To address the underutilization of the processor, we propose a non-blocking integration between FPGA and processor; i.e. an alternative method of scheduling work on the FPGA. Figure 5.4 depicts the work division in this more optimal model. In this model, seed extension is carried out on the FPGA in parallel with seed generation on the processor. As long as the total computation time needed for seed extension is smaller than seed generation ($\lambda > \mu$), the speedup of the application is governed by:

$$S = \frac{1}{1 - k} \quad (5.7)$$

The speedup no longer depends on the latency of the seed extension operation on the FPGA. However we must make sure that the throughput of the FPGA accelerator is high enough and can keep up with seed generation on the processor.

The main culprit in the blocking model is the active polling for results. To address this, we need the following two changes.

1. The task executed on the FPGA should be independent from tasks executed on the processor. A thread should be able to independently process the output of the FPGA, stored in shared memory and should be able to relate it to the input task. With the current design, this was a cumbersome task due to the data dependencies shown in Figure 4.2.

2. The accelerator should autonomously acquire tasks, without MMIO notifications, since an MMIO notification incurs a heavy-weight synchronization instruction on the processor.

These two changes are taken into consideration in the alternative design. The work flow of the heterogeneous design is depicted in Figure 5.5. A task queue is a conceptually simple structure: a shared data structure that contains a list of tasks to be processed. The host processor generates tasks for the accelerator, and stores them into a task queue. At the same time, the FPGA monitors the task queue and executes available tasks. The task queue in this case is known as a producer consumer queue, with multiple producers and a single consumer/worker. Since the task queue is stored in shared memory, both the host processors (producer) and the FPGA (consumer) have access to it.

There are multiple ways to implement a task queue, the main issue being synchronization (to prevent race conditions) between the producers and the consumer. To ensure correctness, concurrent access has to be synchronized. Implementations of queues have been well described in literature for multiprocessor environments [52]. We suggest a way for implementing the task queue based on the lock command provided by CAPI. Actual implementations have not been done and no measurements have been performed and are considered future work.

The queue can be implemented as a circular or linear array, or as a linked list, with head and tail pointers. The producers gain exclusive access to the queue and modify the head pointer (the queue grows from the head) and insert the work items. When a consumer wants to read from the queue, it also gains exclusive access and reads in the work item from the tail pointer and updates the tail pointer. CAPI provides commands to lock cache lines that allow to have atomic access. While the lines are locked, no other read or write accesses to the lines are permitted by any other processor in the system. In
5.5. DESIGN ALTERNATIVE: HETEROGENEOUS SYSTEM WITH A TASK QUEUE

Figure 5.5: Work flow on heterogeneous system using a Task Queue

In this case, a lock can be used on the head and tail pointers. This model can be extended for multiple consumers, e.g., multiple accelerators accessing the same queue, or sharing the same task on a accelerator and a processor.

Traditionally, more advanced techniques are used to efficiently schedule work on a multiprocessor environment, e.g., work stealing [53]. This type of scheduling could also be applied on a heterogeneous system with attached accelerators. Implementation of such a scheduler is not within the scope of this thesis.
In this chapter, various measurements of correctness and performance of the hardware implementation of the Smith-Waterman kernel will be presented. First the functional correctness of the Smith-Waterman implementation on the FPGA is demonstrated in Section 6.2. We do this by comparing the accuracy of the software implementation with the FPGA accelerated implementation. In Section 6.3, we measure the acceleration of the S-W kernel and compare it to a baseline on a Power8 machine. We integrate the accelerated kernel in the sequence aligner BWA-MEM and measure the speedup with respect to the baseline in Section 6.4.

6.1 Experimental setup

Of all measurements, we consider the wall-clock time (the real time that has elapsed), as most important. Other aspects, such as power consumption have not been considered.

In all the experiments, we use data input, reference genomes that are used in current hospitals and research institutions. Our choices for the length of the reads are based on state-of-the-art sequencers, and the advance in sequencers is leading to larger and more accurate reads. Human genome GRCh37 is used as a reference, obtainable from UCSC. We use an input set with about 1M reads, generated with \texttt{wgsim}. See Appendix ?? for an explanation on the tool.

In this experiment a Power8 S284L machine is used. It has 2 10-core Power8 processors, both supporting 8-way multithreading (SMT 8), with around 256 GB of DDR3 memory available. It runs Ubuntu 15.04 LE, with kernel 3.19 that supports CAPI, and uses a 512 GB HDD as a storage system. The accelerator card is a Nallatech 385N, that contains an Altera Statix V FPGA, and contains 16 GB of DDR3 memory on the card.

6.2 Functional verification

The sequence aligner BWA-MEM can be evaluated on throughput and accuracy. Throughput indicates the elapsed time before all reads have been aligned, and accuracy describes the percentage of reads that have been correctly mapped to the reference sequence. In most aligners, there is a trade-off between throughput and accuracy, i.e. more accurate aligners have longer computation times. For accuracy, often two metrics are used in literature:

1. \textit{Aligned reads}: Percentage of reads that are confidently mapped to a location

2. \textit{Error rate}: Percentage of aligned reads that are mapped to a wrong location in the reference genome
The software implementation of seed extension uses a number of heuristics to limit the search space of the dynamic programming algorithm. These techniques have not been employed in the systolic PE array on the FPGA, and therefore output of the seed extension might be slightly different. The output of seed extension is the position where the maximum similarity between two sequences is found, as well as the similarity score that has been given to the alignment based on the scoring system. Depending on the sequences, both implementations can find different best scoring alignments. In the context of BWA-MEM, these different alignments might lead to a different final mapping of the read. We evaluate the effect of the different heuristics in the resulting output of BWA-MEM.

We evaluate the mapping of the reads using simulated reads, for which we know the true location in the genome. We create simulated reads of 100 and 250 bp from the human reference genome hg19 using the \texttt{wgsim} program, distributed in SAMTools [54].

We simulated 100 bp reads, with a 0.1\% mutation rate and a 2\% error rate, representative of sequenced human data and also used in the BWA paper [26]. A mapping quality of 10 is used as a threshold to determine confident mappings, similar to the BWA paper. In Table 6.1 we compare the accuracy of the base implementation of BWA-MEM in software, and the FPGA accelerated version. It shows that difference in accuracy is insignificant.

<table>
<thead>
<tr>
<th>Error</th>
<th>Program</th>
<th>100 bp</th>
<th>250 bp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% Aligned</td>
<td>Error</td>
<td>% Aligned</td>
</tr>
<tr>
<td>2% error</td>
<td>CPU only</td>
<td>92.70%</td>
<td>4.12E-004</td>
</tr>
<tr>
<td></td>
<td>CPU+FPGA</td>
<td>92.70%</td>
<td>4.13E-004</td>
</tr>
<tr>
<td>5% error</td>
<td>CPU only</td>
<td>92.70%</td>
<td>4.11E-004</td>
</tr>
<tr>
<td></td>
<td>CPU+FPGA</td>
<td>92.70%</td>
<td>4.11E-004</td>
</tr>
<tr>
<td>10% error</td>
<td>CPU only</td>
<td>70.60%</td>
<td>8.86E-003</td>
</tr>
<tr>
<td></td>
<td>CPU+FPGA</td>
<td>70.60%</td>
<td>8.84E-003</td>
</tr>
</tbody>
</table>

### 6.3 Smith-Waterman kernel acceleration

We compare the speedup of our accelerated FPGA implementation with the software baseline. As a reference, we run the software on a 20-core Power8 CPU and evaluate the wall-clock time with $N = 1, 2, 4...160$ threads. Power8 supports up to 8-way multi-threading, and thus by setting the number of threads to 160 we map all virtual cores to a physical core and ensure full utilization of all 20 cores. We modify BWA-MEM to first generate all the seeds and collect them in a set. Subsequently we either stream those seeds to the FPGA, or processes them in software in the seed extension step to obtain our two sets of measurements. This way we ensure that only the application kernel time is measured. We use an input with 1M reads, generated with Samtools’s \texttt{wgsim}. We vary the length of the reads between 100 bp and 250 bp, and compare them against the human genome hg19.
To demonstrate the speedup of our accelerator over the original BWA-MEM software, we first determine the performance of the baseline and its scaling with respect to the number of threads. The speedup is defined as the execution time relative to single-thread performance. Since the S-W computation is mainly compute intensive, a speedup proportional to the number of cores in the Power8 system is expected. As shown in Figure 6.1 up until 16 threads the performance scales linearly with the number of threads. We can see that for more than 128 threads, the performance deteriorates.

![Figure 6.1: Smith-Waterman kernel scaling on 20-core POWER8](image)

It must be noted that the BWA-MEM (0.7.10) software implementation of the Smith-Waterman kernel does not make use of SIMD instructions for seed extension, but uses a scalar approach.
In Figure 6.2 and 6.3 we compare the throughput of the software implementation of the Smith-Waterman kernel on a 20-core POWER8 with the FPGA implementation. The throughput is shown relative to the best performing configuration in software. In this case, the runtime of BWA-MEM with 160 threads is taken as a reference, and the other measurements are normalized to this runtime.

In this experiment, all the seeds are collected in a first stage. In the second stage, all the sequences in the seeds are streamed to the FPGA. In general, the throughput of the FPGA design can be limited by two factors in such a streaming application:

1. **Bandwidth limitation**, the interconnect between FPGA and processor is not able to provide the needed bandwidth.

2. **Computational limitation**, the FPGA does not have enough resources to process the incoming data.

With 100 bp reads, we can outperform the 20-core POWER8 (IBM System S2824) with 4 PE arrays. For the 250 bp, we can outperform with a 2 PE array configuration. Increasing the number of PE arrays beyond 6 does not seem to increase the throughput anymore.

To gain further insight in the limitations of the design, we inspect the availability of the S-W cores, and the number of work elements in the queues of the FPGA. A low utilization of the compute cores indicates that the system is limited by bandwidth, while empty queues and high utilization indicate a computational bottleneck.

Utilization of the PE array/core is defined as the non-idle cycles of a core divided by the total cycles needed for the Smith-Waterman computations. A fully utilized core has a 96% utilization rate, since some overhead is caused by moving the sequences into buffers. The mentioned bandwidth is expressed as the data transferred between the PSL...
and the AFU. This excludes all other traffic caused by CAPI such as address translation or coherency.

As discussed in Section 5.2.1, various configurations exists that include multiple systolic arrays in the design, and even the systolic arrays can vary in length. In this chapter, we use a systolic array that matches the length of the reads. For 250 bp reads, we use a systolic array of length 250. This is non-optimal, as discussed in Section 5.2.1.

In Table 6.2, the utilization rates and bandwidth have been summarized. With \( N \times P \), we indicate \( N \) systolic PE arrays, all with length \( P \). The core utilization rate decreases dramatically for more than 6 cores, and we can deduce that the bandwidth is not sufficient for this many cores. Since the bandwidth achieved in other applications, such as the \texttt{memcpy} demo is higher (3.5 GB/s), this indicates a non-optimal implementation of the DMA wrapper.

Furthermore, BWA-MEM encodes the nucleotides in the DNA sequences as an 8-bit character. Since only 4 nucleotides exist in DNA, we only need \( \log(4) = 2 \) bits to encode a nucleotide. This could reduce the required bandwidth by 4, but would require transforming the sequences before sending them. Parameters in the WED could also be more efficiently packed, if bandwidth remains an issue.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Core Utilization</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x100</td>
<td>96.40%</td>
<td>364 MB/s</td>
</tr>
<tr>
<td>2x100</td>
<td>96.40%</td>
<td>727 MB/s</td>
</tr>
<tr>
<td>4x100</td>
<td>94.50%</td>
<td>1425 MB/s</td>
</tr>
<tr>
<td>6x100</td>
<td>88.00%</td>
<td>2004 MB/s</td>
</tr>
<tr>
<td>8x100</td>
<td>68.00%</td>
<td>2091 MB/s</td>
</tr>
</tbody>
</table>

In order to estimate the impact of the acceleration of the S-W kernel in the context of BWA-MEM, we also need to consider the latency of the seed extension step on both software and on the FPGA, according to Equation 5.6. The latency is defined as the time required for a single core to complete the computation of a single seed extension. This is the reciprocal of the throughput on a single core on software, or on a single systolic PE array on hardware. On hardware this latency includes an MMIO request to notify the accelerator, reading in sequences from shared memory, execution on one of the cores and writing the results back to shared memory. The average latency of seed extension for different read lengths can be seen in Figure 6.4. Software has a smaller latency for 50 bp reads, but the FPGA implementation scales much better with respect to the read length. The overhead of the MMIO notification, transferring the data to the FPGA and writing the results back to shared memory amounts to around 3 \( \mu s \).

6.4 Application acceleration

In this section the integrated solution will be evaluated. This solution consists of the integration of the S-W kernel on the FPGA and the BWA-mem software on the Power8 processor. The accelerator is attached through CAPI. To make a fair comparison with
CHAPTER 6. EXPERIMENTAL RESULTS

Figure 6.4: Average latency of S-W kernel

performance of the Power8, the measurements include loading the reference genome from disk, loading the reads from a storage system (disk), and storing the mapped results back to the storage system (disk).

In this accelerated solution, all function invocations to the software-based seed extension \texttt{ksw\_extend} have been replaced with a call through the CAPI stack, this has been further described in Chapter 5. The call to the CAPI stack (\texttt{ksw\_extend\_capi}) does not run in parallel with the rest of the software, threads are busy waiting for the resulting S-W scores to return. Signalling and synchronization between the threads and the coprocessor occurs using shared host memory. In order to achieve any acceleration in this model, the latency of the CAPI solution should be smaller than the latency of a software-based computation.

Figure 6.4 depicts the latency of one S-W computational unit on the FPGA, and compares it with the latency of a seed extension on one Power8 core. The latency on the FPGA is an end-to-end latency, including transferring the input sequences to the FPGA and writing the results back to shared host memory. We see an improvement of about 7-fold decrease in latency for 250 bp reads down to 2.5 times for 100 bp reads. For 50 bp reads, an increase in latency is observed for the FPGA. We see from Equation 5.6, no acceleration is possible in this case.

In Figure 6.4, the speedup of the accelerated solution for reads of length 100 is shown. In a 1x100 configuration, when we execute BWA-MEM with more than 32 threads, \( \lambda > \mu \) and the speedup drops below 1. Intuitively, when we increase the number of PE arrays and keep \( \lambda \) equal, each PE array will experience more idle time and will be able to serve a request faster, reducing the latency. When \( \lambda \) is sufficiently large, i.e. we run BWA-MEM with a large number of threads, the speedup is increasing when more PE arrays are instantiated.

For 250 bp reads, we can observe a similar pattern in Figure 6.6. If we would be
able to place more PE arrays of length 250 on the FPGA, the speedup of the design would approach the optimal speedup. However, not enough resources are available on the FPGA to place more than 3 systolic arrays on it.

The dashed line in Figure 6.5 and 6.6 represents the optimal speedup that we are able to achieve with the computational model of Section 5.4, where progress in software is
blocked by the accelerator call. We assume that there is an infinite number of PE arrays, and thus the wait time in any of the FIFOs is 0. In this case, the process time for one seed extend operation becomes $t_{\text{process}} = 1/\mu$. We fill in the unloaded latency numbers from Figure 6.4 and use $k$ from Figure 4.6 in Equation 5.7 to arrive at the dashed line. An 8x100 configuration approximates the optimal line, as the seed extensions experience virtually no waiting time in the queues.

The dotted line in Figure 6.5 represents the maximum speedup obtainable, when seed extension on the FPGA is running completely parallel with the application on the processor. The call to the FPGA is non-blocking. In this case, this speedup is governed by Equation 5.7. This line represents the maximum speedup achievable, when only the seed extension core is accelerated. We expect the acceleration of BWA-MEM to be close to the optimal line, when integration between CPU and FPGA is implemented with a task queue model, as described in Section 5.5.

In Table 6.3 we show the core utilization numbers and bandwidth numbers when we
run BWA-MEM with the maximum number of threads supported and compare them with the maximum bandwidth obtained when running the isolated S-W kernel. We see that we have full core utilization for a 1x100 configuration, and are not limited by bandwidth for the other configurations. For a 8x100 configuration, the cores are collectively only used 25% of the time.

Table 6.3: Bandwidth utilization of CAPI attached FPGA while running accelerated BWA-MEM

<table>
<thead>
<tr>
<th>Configuration</th>
<th>S-W Core Utilization</th>
<th>Bandwidth (MB/s)</th>
<th>Maximum Bandwidth (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x100</td>
<td>97%</td>
<td>364</td>
<td>364</td>
</tr>
<tr>
<td>2x100</td>
<td>85%</td>
<td>414</td>
<td>727</td>
</tr>
<tr>
<td>4x100</td>
<td>55.8%</td>
<td>848</td>
<td>1425</td>
</tr>
<tr>
<td>6x100</td>
<td>37.15%</td>
<td>841</td>
<td>2004</td>
</tr>
<tr>
<td>8x100</td>
<td>25.35%</td>
<td>763</td>
<td>2091</td>
</tr>
</tbody>
</table>

In Figure 6.7, we show the profile of BWA-MEM of the integrated system. In this diagram, capi-extend indicates the portion of CPU cycles spent in the blocking seed extension kernel, polling for results. Compared to Figure 4.6, the portion of the seed extension kernel is reduced immensely. The seed generation kernel, with over 60% in CPU cycles, has overtaken seed extension as the most time consuming.

![Figure 6.7: Profile of FPGA accelerated BWA-MEM with synthetic single-ended 100 bp reads](image-url)
Conclusion and future work

In our attempt to accelerate BWA-MEM, a popular read alignment tool, we have designed a high-throughput Smith-Waterman accelerator on an FPGA. The design includes multiple systolic arrays, exploiting both data parallelism and wavefront parallelism in the S-W computations. We have integrated the accelerator with BWA-MEM, and offloaded the computationally intensive seed extension to the FPGA.

The seed extension tasks are stored in shared memory, and with a blocking call we have notified the accelerator of the task. Since the latency of the seed extend operation on the FPGA is vastly smaller than on a CPU, we are able to significantly accelerate this operation, even though we are suffering from small communication and control overhead. With the use of shared memory, the FPGA accelerator integrates seamlessly with existing software, in a transparent way to the user. Only small changes to the original source-code of BWA-MEM were required to integrate the accelerator with software. The accelerated system does not sacrifice in accuracy and generates the same alignments as the original BWA-MEM software.

For 100 bp reads and 250 bp reads, we are able to achieve a speedup of respectively 1.3X and 1.6X. However, Amdahl’s Law dictates that the maximum speedup for the reads is 1.7X and 2.1X respectively. Clearly, there is room for improvement.

Under this blocking model, the FPGA accelerator behaves as a queueing system. Seed extension tasks are stored in buffers on the FPGA, and the latency of the task depends on the availability of the systolic arrays, and the process rate of a single systolic array.

The disadvantages of the blocking call, while easy to implement, are twofold.

**Communication overhead** Even though CAPI provides low latency access to shared memory, for very fine-grained tasks communication overhead constitutes about 40% of the total latency of the seed extension task.

**Under-utilization of systolic arrays** Since threads are polling for results to come back from the accelerator, the latency of the accelerated seed extension task determines the speedup. To reduce waiting time in the buffers (and therefore the latency of the operation), a large amount of systolic arrays needs to be instantiated. Our results show that during execution, only about 25% of the systolic arrays are utilized on average. We have shown that bandwidth between FPGA and CPU is sufficient to offload all seed extension tasks on the FPGA.

We propose a non-blocking model, implemented with a Task Queue, that addresses the issues described above. This model removes the seed extension from the critical path on the processor, and allows it to perform other work. The latency of seed extension on the accelerator is no longer critical, as long as the throughput is high enough to
sustain the rest of the system. We could therefore reduce the number of systolic arrays, and preserve area on the FPGA. With the Task Queue in place, we expect to achieve a speedup very close to the maximum 2.1X. By equipping POWER8 processors with an FPGA, we could reduce the physical footprint of Data Centers by two, while maintaining the same throughput for read alignment.

To answer our research questions:

1. How should modern alignment tools, in particular BWA-MEM, be partitioned on a reconfigurable system with a processor and an FPGA for best performance?

Profiling shows that seed extension is the most time-consuming task in BWA-MEM. Because of the limited computational resources on the POWER8 processor, this task doesn’t scale as well as the rest of the system and forms a computational bottleneck. The seed generation part makes use of backtracing by means of the BWT-transform. This part suffers from a high number of cache misses, because of irregular memory access patterns. The POWER8 provides extreme high bandwidth to DDR3 memory, and through SMT provides a way to hide memory latency. We argue accelerating this on an FPGA would be a very cumbersome task.

2. How can we accelerate the S-W workload of BWA-MEM on an FPGA?

The S-W workload represents many small acceleration tasks. We implement a highly parallel design, that is able to read in tasks, perform computations on multiple systolic arrays and write back results to memory concurrently.

3. How can we use shared memory for control and communication of a CAPI attached accelerator, for the acceleration of fine-grained tasks such as sequence alignment?

We show that the traditional I/O attached accelerators require the pinning or copying of memory. The OS has to actively copy the task and required input data to pinned memory. Through CAPI, we have coherent access to the entire virtual address space of the host process. We can notify the accelerator through MMIO requests, or a CAPI attached accelerator can also autonomously read tasks from shared memory without OS involvement.

7.1 Recommendations for future work

7.1.1 Hash-based seed generation

Since seed extension is no longer a bottleneck in the accelerated BWA-MEM, seed generation becomes the performance limiting operation. Memory capacities on today’s servers, such as the IBM Power System S824 with 256 GB RAM, are not leveraged by BWT-based backtrace methods. Only around 8 GB RAM is used during execution of BWA-MEM. Hash-based methods require more memory (39 GB as reported by Zaharia et al. [24]), and could potentially increase the throughput of seed generation.

build 0.18
7.1.2 Implement task queue for multiple accelerators

Work on implementation of the task queue has started, but is not yet finished. It would be interesting to find data structures and methodologies that would lead to minimum synchronization overhead. Up to four FPGAs can be coherently attached to a POWER8 824 system. We would like to extend the task queue model to accommodate multiple accelerators. Scheduling the tasks to the accelerators now becomes an interesting problem.

7.1.3 Reduce FPGA resources

We have seen that single PEs have a higher resource utilization than systolic PE arrays. Applying these single PEs in a design could reduce area required to maintain a specific throughput. The free area could be used to accelerate other application kernels in BWA-MEM, such as output generation or seed location.
Bibliography


[22] H. Lin, Z. Zhang, M. Q. Zhang, B. Ma, and M. Li, “Zoom! zillions of oligos
http://bioinformatics.oxfordjournals.org/content/24/21/2431.abstract


[24] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. A. Patterson, S. Shenker,
I. Stoica, R. M. Karp, and T. Sittler, “Faster and more accurate sequence
http://arxiv.org/abs/1111.5572


http://dx.doi.org/10.1093/bioinformatics/btp324

alignment of short dna sequences to the human genome,” Genome Biology, vol. 10,


database and memory access optimizations,” 2011.

Analysis and novel design.” in ARC, ser. Lecture Notes in Computer Science,
A. K. 0001, R. Krishnamurthy, J. McAllister, R. Woods, and T. A. El-
http://dblp.uni-trier.de/db/conf/arc/arc2011.html


http://bioinformatics.oxfordjournals.org/content/23/2/156.abstract

[33] A. Wozniak, “Using video-oriented instructions to speed up sequence comparison,”

[34] T. Rognes and E. Seeberg, “Six-fold speed-up of Smith-Waterman sequence

build 0.18


[51] [Online]. Available: https://www.altera.com/


[54] [Online]. Available: samtools.sourceforge.net