Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2011

MSc THESIS

Genetic sequence alignment on a supercomputing platform

Erik Vermij

CE-MS-2011-02

Abstract

Genetic sequence alignment is an important tool for researchers. It lets them see the differences and similarities between two genetic sequences. This is used in several fields, like homology research, auto immune disease research and protein shape estimation. There are various algorithms that can perform this task and several hardware platforms suitable to deliver the necessary computation power. Given the large volume of the datasets used, throughput is nowadays the major bottleneck in sequence alignment. In this thesis we discuss some of the existing solutions for high throughput genetic sequence alignment and present a new one.
Our solution implements the well known Smith-Waterman optimal local alignment algorithm on the HC-1 hybrid supercomputer from Convey Computer. This platform features four FPGAs which can be used to accelerate the problem in question. The FPGAs, and the CPU that controls them, live in the same virtual memory space and share one large memory. We developed a hardware description for the FPGAs and a software program for the CPU. Some focus points were: a sustainable peak performance, being able to align sequences of any length, FPGA area efficient computations and the cancellation of unnecessary workload.
The result is a Smith-Waterman FPGA core that can run at 100% utilization for many alignments long. They are packed per six on a FPGA running on 150 MHz, which results in a full system performance of 460 GCUPS (billion elementary operations per second). Our elementary processing element can deliver double the work per clock cycle than a naive implementation, resulting in a better throughput per area ratio. At a system level a notable amount of workload is cancelled. It is the most flexible implementation we are aware of . We re-evaluate the use of FPGAs for accelerating Smith-Waterman and conclude that they will continue to be a good choice per dollar and per watt, as long as we narrow the problem space.

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Genetic sequence alignment on a supercomputing platform

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Erik Vermij
born in Gouda, the Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Genetic sequence alignment on a supercomputing platform

by Erik Vermij

Abstract

Genetic sequence alignment is an important tool for researchers. It lets them see the differences and similarities between two genetic sequences. This is used in several fields, like homology research, auto immune disease research and protein shape estimation. There are various algorithms that can perform this task and several hardware platforms suitable to deliver the necessary computation power. Given the large volume of the datasets used, throughput is nowadays the major bottleneck in sequence alignment. In this thesis we discuss some of the existing solutions for high throughput genetic sequence alignment and present a new one.

Our solution implements the well known Smith-Waterman optimal local alignment algorithm on the HC-1 hybrid supercomputer from Convey Computer. This platform features four FPGAs which can be used to accelerate the problem in question. The FPGAs, and the CPU that controls them, live in the same virtual memory space and share one large memory. We developed a hardware description for the FPGAs and a software program for the CPU. Some focus points were: a sustainable peak performance, being able to align sequences of any length, FPGA area efficient computations and the cancellation of unnecessary workload.

The result is a Smith-Waterman FPGA core that can run at 100% utilization for many alignments long. They are packed per six on a FPGA running on 150 MHz, which results in a full system performance of 460 GCUPS (billion elementary operations per second). Our elementary processing element can deliver double the work per clock cycle than a naive implementation, resulting in a better throughput per area ratio. At a system level a notable amount of workload is cancelled. It is the most flexible implementation we are aware of . We re-evaluate the use of FPGAs for accelerating Smith-Waterman and conclude that they will continue to be a good choice per dollar and per watt, as long as we narrow the problem space.

Laboratory : Computer Engineering
Codenumber : CE-MS-2011-02

Contents

Acknowledgements

First of all, I would like to thank Dr. Zaid Al-Ars for his supervision and support throughout the work. I would like to thank Convey Computer, especially Glen Edwards, for their help on the HC-1 personality development kit. I would like to thank Laiq Hasan and Zubair Nawaz for their help and feedback. From Imperial College Londen, I would like to thank Jose Gabriel de F. Coutinho and Brahim Betkaoui for letting me use their HC-1 and helping me out with several problems. Last but not least, I would like to thank Stephan Wong and Dick de Ridder for being on my graduation committee.

Erik Vermij
Delft, The Netherlands
March 11, 2011

1 Introduction

1.1 Introduction to bioinformatics

Over the last few decades, biologists have made major steps in trying to understand life; humans, animals or plants. One of the largest projects was the Human Genome Project, started in October 1990. Some of the goals of this project where to

- identify all the approximately 20,000-25,000 genes in human DNA,
- determine the sequences of the 3 billion chemical base pairs that make up human DNA,
- store this information in databases.

The project was successfully completed in 2003 [30]. One of the major drives behind research like this is the medical world. Understanding genetic diseases, autoimmune diseases and protein related diseases (Creutzfeld-Jacob, cancer) can lead to (better) medicines and treatment [7, 36].

Nowadays the challenge shifted from the data gathering to the data processing. While tools exist to extract the DNA of a healthy and a sick person, the tools for analyzing the significant differences (and find a remedy) are still largely not in place. Here is where computer science comes in. With the help of algorithms and ever growing computer power, computer scientists and biologists are now working together to crack problems. This is a field that is generally referred to as bioinformatics.

For further understanding we take a look at the structure of the human genome. This can best be explained in layers. A graphical view of the system is given in Figure 1.1.

Chromosomes
Every healthy human being has 23 chromosome pairs. 22 pairs are found in both men and women, 1 pair is sex dependent. Chromosomes are made out of an enormous stretch of DNA and are found in every cell [7].

DNA
DNA (or DeoxyriboNucleic Acid) is a very large double helix shaped molecule. On every backbone you will find a sequence of four types of nucleotides, namely Adenine (A), Thymine (T), Cytosine (C) and Guanine (G). The two backbones are connected by the nucleotides via coupling, A with T and C with G, these couples are called base pairs. This fixed coupling makes it possible to express the DNA with only 1 sequence of nucleotides. There are over 3 billion base pairs in the human DNA [7].

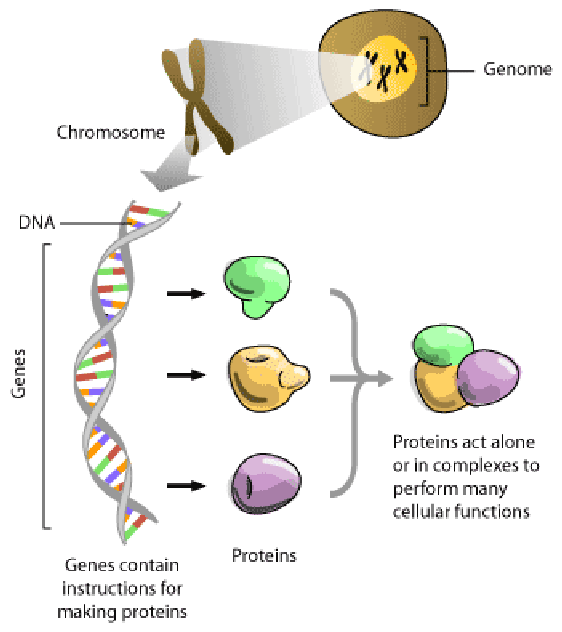

Genome
Chromosome
DNA
Genes
Proteins act alone or in complexes to perform many cellular functions
Genes contain instructions for making proteins
Proteins

Figure 1.1: Schematic view of the human genome from chromosome to protein.

Genes
Genes are continuous subparts of DNA which have a coding function, for example the color of ones eyes. Genes only occupy 1.5% of the total DNA; the function of the remaining part is largely unknown or beyond the scope of this thesis. Humans have somewhere between 20,000 and 25,000 different genes [7].

Proteins
Proteins are created by a complex biological process which is controlled and initialized by a gene. The resulting protein can consist out of 20 types of amino acids and once created, folds into a 3D shape. Together with other proteins and molecules it keeps the cell alive, as well as communicating with its environment [7].

1.2 Genetic sequence alignment

Genetic sequence alignment is the science of aligning 2 or more genetic sequences (DNA, proteins or others) in such a way, that you can extract the maximum amount of information from their similarities or differences. Two possible alignments examples are visible in Listing 1.1. The sequences are written in two rows and similar items are placed in the same column. To optimize the alignment, inserts (-) can be introduced or items can stay mismatched. Inserts are known by the term indel, since they occur due to mutational inserts or deletions.

	A	**T**	**A**	**T**	**C**	**G**	**G**	**C**
A	●		●					
T		●		●				
C					●			●
G						●	●	

Table 1.1: Dot matrix

Listing 1.1: Genetic sequence alignment

```
Two sequences of DNA
S1: ATATCGGC
S2: ATCG

Alignment 1
S1: AtaTCgGc
S2: A—TC–G–

Alignment 2
S1: atATCGgc
S2: ––ATCG—
```

1.2.1 Dot matrix

The dot matrix is a tool that can be used by researchers or algorithms to identify regions of similarity (alignments) in two sequences. Its main concept (pair wise comparison) is used in many other algorithms. When building the dot matrix, one of the sequences defines the rows, the other one the columns. A matrix cell is filled with a dot if the sequence item of its row matches that of its column. Regions of similarity (possible alignments) between the two sequences are recognized by a diagonal line of dots. The dot matrix for the sequences in Listing 1.1 can be seen in Table 1.1, with a region of interest in the central 4×4 submatrix.

A way to filter out random matches is to use a sliding window. Instead of item to item comparison, for example 20 items are compared, and only if the amount of hits is greater than a certain threshold, a dot is drawn in the base of the window. The size of the windows and the threshold can be different for DNA and protein sequences, driven by the fact that a random match is much more likely in the former [36].

1.2.2 Creating useful alignments

The goal of genetic sequence alignment is to make an alignment that is as good as possible. What a ‘good’ alignment is often depends on what the researcher in question is trying to achieve. We will discuss two important factors that can steer the alignment in the right direction.

Scoring scheme

An important aspect of creating alignments and evaluating them is the scoring scheme

[18]. Such a scheme gives a value to a match s(a, a), a mismatch s(a, b) and a gap (indel) s(a, -) or s(-, a). Here is s(,) a function which returns a certain score depending on its two parameters. The summation of all these score values defines the quality of your alignment, and choosing the score scheme is therefore an important aspect of the alignment process.

Genetic sequences can be constructed out of four letters for DNA, or 20 letters for proteins. For DNA, match and mismatch scores can be the same for every letter combination. This does not hold true for protein sequences. Some amino acids mutations are more likely to happen than others, so a more complex scoring scheme is needed. The s(a, a), s(a, b) and s(b, a) values for protein sequences can be found in 20×20 substitution matrices like PAM and BLOSUM. These are families of matrices, each one of them designed for a special biological problem. Some scoring schemes introduce dependencies among the columns. This is mostly seen in creating a gap penalty depending on the length of the gap using an affine gap penalty function. For example, when entering a gap subsequence you get a startup penalty and a penalty for every consecutive gap in the subsequence. This can help making alignment scores more realistic following the biological phenomenon that opening a gap is harder than extending it [18]. The alignments from Listing 1.1 are evaluated given a scoring scheme in Listing 1.2.

Listing 1.2: Evaluation of two alignments with a scoring scheme

```
Scoring scheme:
s(a, a)              = 2
s(a, b) = s(b, a)    = −1
Opening gap          = −3
Extending gap        = −1

Alignment 1:
S1:     A   t   a T C   g G   c
S2:     A   −   − T C   − G   −
Score:  2  −3  −1 2 2  −3 2  −3 = −2

Alignment 2:
S1:      a   t A T C G   g   c
S2:      −   − A T C G   −   −
Score:  −3  −1 2 2 2 2  −3  −1 = 0
```

Local and global alignments
Another important aspect is whether we are going for a local or global alignment. Let us look at the following two DNA sequences and two possible alignments in Listing 1.3.

Listing 1.3: Local and global alignments

```
Two sequences of DNA
S1: TCCCAGTTTGTGTCAGGGGACACGAG
S2: CGCCTCGTTTTCAGCAGTTATGTGCAGATC

Alignment 1:
S1: -----------tccCAGTT-TGTGTCAGgggacacgag
S2: cgcctcgttttcagCAGTTATGTG-CAGatc-------

Alignment 2:
S1: tcCCa-GTTTgt-GtCAGggg-acaC-GA-g
S2: cgCCtcGTTTtcaG-CAGttatgtgCaGAtc
```

Both alignments are valid, but totally different. Alignment 1 is aligned locally, while alignment 2 is aligned globally. When aligning two sequences locally, you try to find long subsequences with the highest possible matching score. This method can indicate similar parts in long, further unrelated, sequences. In contrast, when aligning two sequences globally, you try to maximize the number of matches between the two sequences along their entire lengths. As can be seen in Listing 1.3, global alignments will not give you a single high scoring region, and are therefore not used that much in practice.

1.2.3 Applications of sequence alignment

Here we would like to give some example applications of genetic sequence alignment.

Homology research, genetic diseases

Homology research focuses on creating evolutionary trees. This research can be done by aligning large parts of DNA from various life forms, making the similarities and differences visible. By doing so it can become clear which animals are close or distant relatives. For example, humans share a lot of their DNA with monkeys and other large primates, but not with jellyfish [7].

The same can be done on a human-only scale. This can be helpful for detecting the genes responsible for genetic diseases. Aligning the DNA of a healthy and a sick family will make the differences (and thereby hopefully the defects) visible [7].

Auto immune diseases

Auto immune diseases (AID) are a class of diseases where the body attacks itself. Normally, the defense mechanism of the body searches for cells with a protein sequence fingerprint known to be bad (a virus for example). With an AID, some of the normal and healthy body cells have a protein subsequence matching the fingerprint of a bad cell. The result is an attack by the immune system, followed by degrading functionality of the attacked body part [32].

Some well known AIDs are Multiple Sclerosis and Rheumatoid Arthritis. These, as well as many other AIDs cannot be treated effectively. Sequence alignment can help identify the similarities between the protein sequences in body cells of an AID patient, and known viruses and bacteria. When significant similarities have been found, medicine can be developed that changes the genetic code of the body, rendering the cells immune again [36].

Protein functions
When created, proteins quickly fold into a 3D shape. This shape, together with the amino acid composition defines the behavior and function of the protein. The folding process is extremely complex, and it is therefore as good as impossible to predict the 3D shape of a protein, given its amino acid string. To make an estimation of the final function based on the amino acid sequence, sequence alignment can be used. Comparing the sequence to a database of known proteins and their function, and estimation can be made [7]. This is of course not a fault free method, but it is easy and cheap compared to most other solutions.

1.3 Problem definition

Sequence alignment as described in Section 1.2 is not a difficult problem in itself. The challenge lies in the enormous amount of available sequences. Typical database sizes are in the order of gigabytes, and growing. Searching them by applying alignment algorithms currently takes a lot of time, thereby limiting the amount of practical work researchers can do. There have been several attempts to accelerate the alignment of sequences, each of them choosing its own approach. In this thesis we will discuss a couple of them, and present another one based on our ideas. Hopefully this work is another brick in the wall of knowledge, aimed at making sequence alignment fast en easy available for any researcher.

2 Genetic sequence alignment algorithms

In this chapter a classification of genetic sequence alignment algorithms is made. Three of them are explained in detail, focusing on alignment accuracy and parallelization options. Following their explaination, we discuss serveral selection criteria, to finally choose the most suitable algorithm for acceleration in the conclusion.

2.1 Classification of algorithms

For the classification we will look at two different properties, whether the algorithm is based on dynamic programming or heuristics and whether it produces a local of global alignment. A schematic view including the algorithms to be discussed can be seen in Figure 2.1.

Dynamic programming and heuristic algorithms

Dynamic programming is a method of solving a big problem by breaking it down into smaller parts and solving them individually. From all the solved parts, the final solution can be derived. This results in an optimal solution. As we will see later on, dynamic programming algorithms often require a lot of computations and can therefore be considered as 'slow'. Heuristic algorithms rely on the knowledge of their engineer and user to produce an as good as possible result. The algorithm is steered by educated guesses and parameters, and will not always find the optimal solution (but they may). They do not rely so much on brute force computations as dynamic programming ones and can therefore be considered as being 'fast'. The validity of these statements obviously depends on the problem that is addressed and the implementation used.

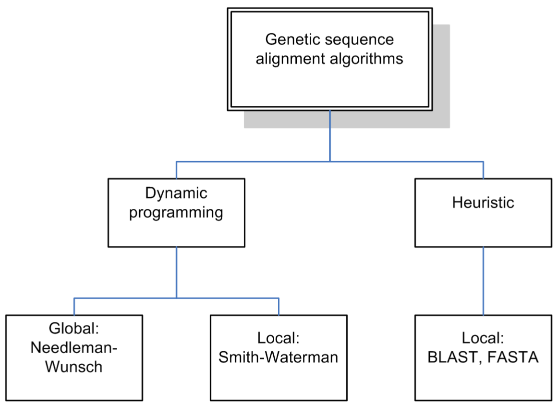


Figure 2.1: Classification of algorithms

2.2 Dynamic programming algorithms

In this section we will take a look at two genetic sequence alignment algorithms that are based on dynamic programming.

2.2.1 Needleman-Wunsch

In 1970 Needleman and Wunsch suggested a general method to search for similarities in two protein sequences [27], known as the Needleman-Wunsch (N-W) algorithm. The algorithm always produces the optimal global alignment between two sequences. Since global alignment algorithms are hardly used, a further analysis of N-W will not be done. Most of the features of the Smith-Waterman algorithm, discussed in the following section, have an analog in N-W.

2.2.2 Smith-Waterman

In 1984 Smith and Waterman suggested a variant on the N-W algorithm, known as the Smith-Waterman (S-W) algorithm [37]. With a small alternation of existing formulas, the algorithm is changed from finding the optimal global alignment into finding the optimal local alignment. It fills a matrix which keeps track of the degree of similarity between the two sequences compared. The definition for S-W with a linear gap model is given in Equation 2.1. Where H is the similarity matrix, $S_{i,j}$ is the similarity score of comparing sequence A_i with sequence B_j and d is the gap penalty. The sizes of the two sequences being compared are n and m. The algorithm can be extended to use an affine gap model, but this will not be shown here.

$$\begin{aligned}
&H_{i,0} = 0, i \leq n \\
&H_{0,j} = 0, j \leq m \\
&H_{i,j} = max \begin{cases} H_{i-1,j-1} + S_{i,j} \\ H_{i-1,j} - d \\ H_{i,j-1} - d \\ 0 \end{cases} \qquad i > 0, j > 0
\end{aligned} \tag{2.1}$$

With Equation 2.1 a similarity matrix can be built. First, it has to be initialized using the first 2 lines the equation, after that; the matrix can be filled using the recurrent part of the equation. The latter is referred to as the matrix fill step. In Table 2.1 an example S-W matrix is given. The optimal local alignment (bold in Table 2.1) can be found as follows:

- Find the maximum value in the similarity matrix; this is the starting point.
- From the starting point move towards the origin always choosing the cell with the highest value. The alignment follows the path, creating indels for vertical and horizontal movement.

	-	**T**	**C**	**C**	**T**	**G**	**T**	**G**	**T**	**C**	**G**
-	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	2	0	2	0	0	2
T	0	2	0	0	**2**	0	2	0	4	2	0
G	0	0	1	0	0	**4**	2	4	2	3	4
T	0	2	0	0	2	2	**6**	4	6	4	2
G	0	0	1	0	0	4	4	**8**	6	5	6
G	0	0	0	0	0	2	3	6	**7**	5	7
T	0	2	0	0	2	0	4	4	**8**	6	5
C	0	0	4	2	0	1	2	3	6	**10**	8
A	0	0	2	3	1	0	0	1	4	8	9

Table 2.1: Smith-Waterman matrix, +2 for a match, -1 for a mismatch and -2 for a gap

- When you encounter a 0, the alignment is finished.

The matrix fill step requires $m \times n$ operations, rendering the process quadratic in time complexity. The matrix itself requires $m \times n$ memory space, rendering the process also quadratic in space.

Alignment accuracy

It is proven that the S-W algorithm finds the optimal local alignment [37]. However, if you have two alignments which are disconnected by a gap, the S-W algorithm only returns the best one.

Parallelization and speedup

The S-W algorithm is very suitable for massive parallelization. We will take a look at some of the possibilities since they are an important aspect for the further evaluation of the algorithm.

- Fine-grained parallelization. Since the matrix cells only depend on their three left-upper neighbors, filling the matrix can be parallelized. Every cell which has its dependency data available can be filled. This is the drive behind many vector processing (like [38]) and FPGA (Field Programmable Gate Array) implementations (like [2]).

- Coarse-grain parallelization. The algorithm is also suitable for coarse-grain parallelization. The matrix can be split into smaller parts which can be processed in parallel; given their three left-upper neighbors are processed. This gives rise to multithreaded (broad sense) implementations running on clusters of CPUs, FPGAs or any other hardware platform.

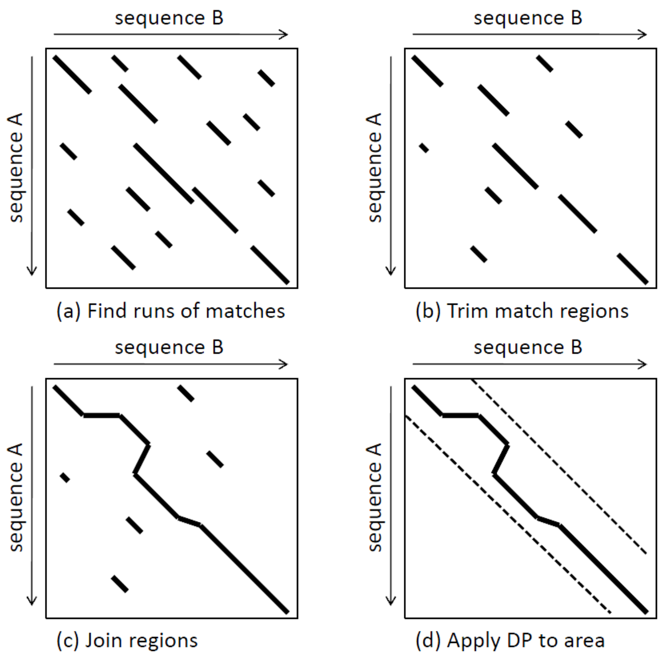


Figure 2.2: The steps of the FASTA algorithm [12, 32]

2.3 Heuristic algorithms

Despite providing the optimal alignment, algorithms entirely based on dynamic programming are not very popular because of the computational demands. There are numerous algorithms proposed which ease the dynamic programming step by using some heuristics. Other algorithms rely entirely on heuristics.

2.3.1 FASTA

In 1987 Pearson and Lipman proposed some new tools for the alignment of genetic sequences [32]. One of them was FASTA. The algorithm consists out of four steps, graphically shown in Figure 2.2. The algorithm first uses heuristic methods to make the alignment problem smaller and then uses a dynamic programming step to find the optimal solution of that sub problem. Each step is explained below.

1. First a dot matrix like plot is made. A setting called *ktup* defines how many consecutive matches are needed for a hit, like the sliding window technique discussed in Section 1.2.1. A result can be seen in Figure 2.2 (a).

2. The found diagonals are rescanned with an actual score scheme, like PAM or BLOSUM for proteins. During this scan, similarity matches smaller than *ktup* may contribute to the score. Every diagonal is assigned an 'initial region', based on the maximal score found. Around 10 of the best diagonals are kept and can be seen in Figure 2.2 (b).

3. Given the location and score of the initial regions, FASTA calculates which diagonals can contribute to an optimal alignment. The rest is discarded. The result can be seen in Figure 2.2 (c).

4. Smith-Waterman is used to calculate an optimal alignment in a restricted space around the selected diagonals. The space restriction can be seen in Figure 2.2 (d) by the dashed lines.

Alignment accuracy
A sensitivity study performed in 1991 [31] showed that FASTA with a *ktup = 1*, is beaten by S-W on 8 out of 34 groups of proteins. The sensitivity becomes worse when the *ktup* is increased.

Parallelization and speedup
Here we will take a look at the parallelization possibilities of the FASTA algorithm.

- Fine-grained parallelization. The first step of FASTA can be done entirely in parallel. Every cell only depends on the sequences (input data), so no data dependencies exist. This opens the door for fine-grained parallelization whether through vector processing or FPGA implementations. No publications have been found considering this approach. In the second step the rescoring of every diagonal could also benefit from vector processing.

- Coarse-grain parallelization. For the first step, the matrix can be split into sub matrices, since there are no data dependencies at all. The second step can be parallelized per diagonal. This, similar to S-W, gives rise to multithreaded (broad sense) implementations running on clusters of CPUs, FPGAs or any other hardware platform.

2.3.2 BLAST

In 1990 Altschul et. al. proposed a new entirely heuristic algorithm for local alignment of sequences, BLAST (Basic Local Alignment Search Tool) [3]. The algorithm can be described in three steps, but the modern implementations are far more complex.

1. Create all possible words of length w from the query sequence L. For every word, find a list of words that, when matched, score above a certain threshold T using a score scheme like PAM. Dependencies between neighbor words can be introduced. All the matching words are stored in an efficient tree like structure.

2. The resulting list of words is compared to every sequence in a database. Sequences with (multiple) word matches are stored, the rest is discarded.

3. The word matches are extended in both directions, to find alignments that score above a certain threshold *S*.

The parameters w, L and S define the behavior of the algorithm. A low value for w increases the sensitivity, but also increases the runtime and memory requirements. A large value for S may result in missing local alignments; a small value may result in a large amount of uninteresting alignments [5].

Alignment accuracy
The nature of BLAST suggests that the algorithm finds a large amount of relative small, disconnected alignments. A comparison against S-W for large sequences was performed in [22] with the conclusion that BLAST omits about 37% of the amino acid positions, against 3% for S-W. In words this means that S-W finds most of the alignments found by BLAST, but BLAST does not find all the alignments found by S-W. This behavior is also visible in a comparison between BLAST and a S-W implementation called z-align [6]. From [22, 6] it is visible that S-W tends to find longer alignments than BLAST, as suggested above.

The original BLAST implementation from the NCBI (National Center for Biotechnology Information) is still under development so improvements can be expected in the future.

Parallelization and speedup
Not much is written about the parallelization and speedup possibilities of BLAST. Most accelerated versions focus on speedup through database or query splitting, which is a form of speedup that can always be done and is of no interest here. Since the algorithm itself is complex, a lot of non-fundamental improvements can be made. The ongoing development of the BLAST tool will result in faster versions.

- Fine-grained parallelization. The first step might possibly be accelerated with vector processing. The remaining steps are too complex to be accelerated with a fine-grained method.
- Coarse-grain parallelization. The first step can be split into coarse blocks for parallel processing. The possible word dependencies must be taken into account. The second step, the match searching process can be done in parallel efficiently. The third and last step can be done in parallel, but is tricky due to the possibilities of connecting alignments.

2.4 Algorithm comparison and selection

From the discussed algorithms we will pick the one which fits our requirements and has the best papers for a fast optimized implementation. First, we take a look at the criteria.

2.4.1 Evaluation criteria

The considered criteria are listed and explained below.

1. Alignment accuracy. The alignment accuracy, or in other words the value of the result, is of major interest.

2. Baseline computational speed. To make a statement about the value of speedup possibilities it is necessary to know what the inherent performance of each algorithm is.

3. Speedup potential. Since this research is aiming at the speedup of one of the algorithms, it is important to look at the possibilities for a speedup. One of the possibilities is coarse-grain parallelization, and enables the algorithm be run in a divide and conquer fashion. Another important speedup can be achieved due to fine-grained parallelization like vector processing, or parallel processing on a FPGA.

4. Memory bottlenecks. Implementation problems considering memory must be taken into account.

2.4.2 Comparison and selection

The criteria discussed in Section 2.4.1 are evaluated for every algorithm and shown in Table 2.2.

Whether we can accelerate an algorithm, the single most important aspect of it is its ability to be cut into smaller pieces which can be solved individually, without any dependencies. We are undoubtedly moving towards a many-core era so a simple, regular algorithmic structure, which is easy to parallelize, is key. The shift towards many-cores goes hand in hand with a move towards vector processing, currently still implemented as rather simple instructions, but in the future possibly as highly specialized high throughput units. Smith-Waterman is the only algorithm which can fully benefit from these paradigm shifts. It is therefore, with an eye on the future, the most promising algorithm for genetic sequence alignment.

2.5 Conclusion

Smith-Waterman outperforms FASTA and BLAST on scalability, alignment accuracy and in some cases already in speed. Furthermore it can be implemented easily on a FPGA for a potential speedup. It is for all these reasons the algorithm of choice for further research.

Alignment accuracy	
Smith-Waterman	- Optimal. - Reports only the best alignment.
FASTA	- Sub-optimal. - Reports only the best alignment.
BLAST	- Sub-optimal. - Reports all alignments.
Baseline computational speed	
Smith-Waterman	- Traditionally the slowest of the three. - Pure software implementations of S-W using vector instructions have beaten BLAST for large (300+ residues) sequences [38]. Whether this advantage scales to much longer sequences is an open question.
FASTA	- Traditionally slower than BLAST.
BLAST	- Traditionally the fastest of the three.
Speedup potential	
Smith-Waterman	- Massive parallelization possible due simplicity and regularity. - Vector processing can boost performance considerably [38]. - FPGA implementations are well researched and can be faster than pure software [2].
FASTA	- Massive parallelization for some steps possible. - Vector processing might boost the performance for some parts of the algorithm.
BLAST	- Massive parallelization possible but complex. - There are only a small number of FPGA implementations available, but considerably faster than pure software [19].
Memory bottlenecks	
Smith-Waterman	- When aligning large (millions of base pairs) sequences, the memory requirements can become infeasible. Nonetheless, implementations like Z-Align solve this problem while maintaining the optimal performance [6].
FASTA	- No difficulties.
BLAST	- No difficulties.

Table 2.2: Algorithm comparison

3 Smith-Waterman performance analyzed

In this chapter we will first take a better look at how the S-W algorithm works and how it can be parallelized. With this in mind we take a look at some of the possible implementations on different platforms for S-W. For every platform, CPUs, FPGAs and GPUs (Graphical Processing Units) some of the current cutting edge implementations will be discussed as well as an estimate of future performance given the development trends of the platform. In the conclusion we will choose a platform that is most promising for a high speed optimized implementation.

3.1 Smith-Waterman performance

In this section we take a better look on how the S-W algorithm behaves with respect to performance, or computational speed. As explained in Section 2.2.2 the S-W algorithm consists out of three steps. First the top row and most left column of the similarity matrix (‘the matrix’) are set to zero. Second, the matrix is filled with the recurrent equations shown in Equation 2.1, followed by the last step, the trace back. The filling of the matrix is by far the most intensive task, so accelerating S-W is mainly done by speeding up this step.

3.1.1 Data depedencies in the S-W algorithm

If we look at the definition of S-W in Equation 2.1, we can see that every matrix cell depends on its three upper-left neighbors. So every matrix cell has three data dependencies which have to be resolved before its value can be calculated. This means we cannot fill the matrix in a random fashion, but have to iterate through the cells so that we never get blocked by an unresolved data dependency. In Figure 3.1 this is made visible for a small matrix. For the left-most matrix, the top row and left-most column have been initialized to zero, and all available data dependencies are visualized with an arrow. The only cell that has all three dependencies fulfilled the shaded one. So this iteration we can only calculate the value of that matrix cell. In the center matrix we are one iteration further, and there are now two cells ready (shaded) to be calculated, and three cells in the right-most matrix.

3.1.2 Filling the S-W matrix

If we want to fill the entire matrix, there are several ways to go.

Sequential

As can be seen in Figure 3.1 it is possible to fill the matrix column or row wise. When

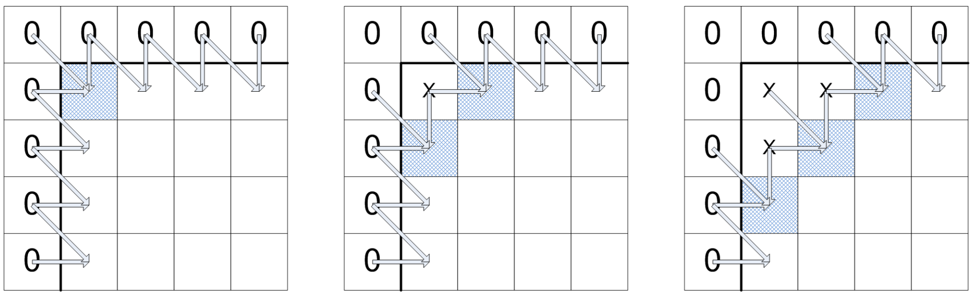


Figure 3.1: Data dependencies in the S-W algorithm

cell $(1, 1)$ is calculated, we can fill $(1, 2)$ and cell $(1, 3)$ after that etc. This way we would, for two sequences of length m and n, need $m \times n$ iterations, which can grow rapidly when the sequences become larger. In Figure 3.2 it is visible in the most left matrix, that we need 16 iterations to fill a simple 4×4 matrix using this method. We can reduce the number of iterations needed if we take advantage of the independence between much of the cells.

Full parallel
In Figure 3.1 it can be seen that the data dependencies allow us to calculate the values of all the cells on every anti-diagonal at the same time. How this works out can be seen in the center matrix in Figure 3.2. Instead of 16, we only need seven iterations, which is $(m + n - 1)$. This is the fastest way to fill a S-W matrix, but requires a maximum of $min(n, m)$ calculations to be done simultaneously, which can sometimes not be possible due to hardware limitations.

Semi parallel
A compromise between sequential and full parallel is the semi parallel method. Here we do not utilize all the available parallelism. An example of this can be seen in the right-most matrix in Figure 3.2. Here we calculated the values of at most two cells in parallel, thereby reducing the amount of hardware we need. This results in a slight increase in required iterations as opposed to the full parallel method, but it is still a lot better than the sequential one.

3.1.3 Terminology

Since the speed by which the S-W matrix is filled plays a major role in this thesis, we will introduce the term GCUPS, or Giga Cell Updates Per Second. This is a widely used performance metric for the Smith-Waterman algorithm and indicates how fast the matrix can be filled. An implementation delivering 10 GCUPS can calculate 10 billion matrix values per second.

Another term that will be used is CUs, or Cell Updates. This can be used to indicate the amount of workload done, or to be done for example.

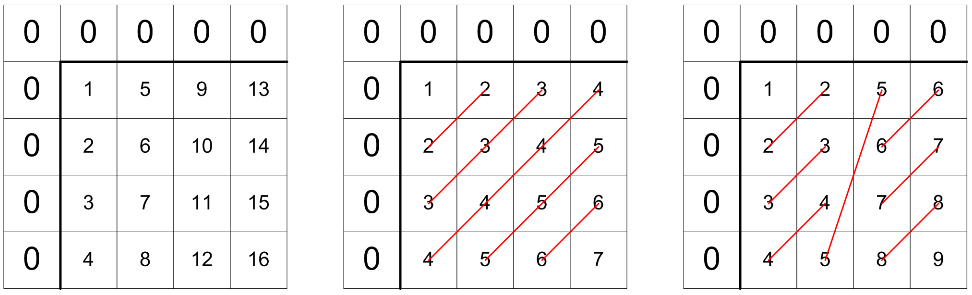


Figure 3.2: Iterations needed to fill the S-W matrix. From left to right: sequential, full parallel, semi parallel. The anti-diagonal lines show which cells can be calculated simultaneously

3.2 CPU implementations

In this section we will look at CPU technology and some known implementations of the S-W algorithm. Furthermore an estimation will be made of the performance of future systems.

3.2.1 SIMD technology

Single Instruction Multiple Data, or SIMD technology is a technique for doing the same task on different pieces of data. SIMD is currently implemented in modern CPUs under the name SSE (Streaming SIMD Extension) and features 128 bit wide registers. These registers can be split up in 16 8-bit words, or eight 16-bit words etc. [38] A single instruction can operate on all these words at the same time, resulting in a maximum of 16 operations per clock cycle. This technique can be used to accelerate S-W.

3.2.2 Modern implementations

The first CPU implementations used a sequential way of calculating all the matrix values. These implementations were slow and therefore hardly used. In 2006, Farrar introduced a SSE implementation for S-W [11]. His work used SSE2 instructions for an Intel processor and was up to six times faster than existing S-W implementations. Two years later, in 2008, Szalkowski et. al. proposed a minor adjustment to Farrars code [38]. Their work is known by the name SWPS3. A SWPS3 version optimized for multithreading was released in 2010 [1]. The SSE implementations can be viewed as being semi parallel. They, discarding startup and finish time, constantly calculate 16, eight or less values at the same time. We will take a look at the performance achieved with these various implementations on various CPU platforms in Table 3.1.

3.2.3 Performance estimations for top-end and future CPUs

With the data from Table 3.1, we can make an estimate of the performance on the current top-end CPUs and take a look into the future. An overview is created in Table 3.2, with estimated peak performances based on the SIMD register width, the number of cores,

	Farrar (2006)	**SWPS3 (2008)**	**SWPS3 (2010)**
Peak performance	2.998 GCUPS	15.7 GCUPS	35 GCUPS
Benchmark hardware	2.0Ghz, Xeon Core 2 Duo, one thread	2.4Ghz, Core 2 Quad Q6600, four threads	2.5Ghz, 2x Xeon Core Quad E5420, eight threads
Estimated peak performance / thread @ 2.5 Ghz	3.75 GCUPS	4.08 GCUPS	4.38 GCUPS

Table 3.1: Performance for different S-W implementations

	Released	**SIMD register width**	**Cores (threads)**	**Clock speed**	**Estimated peak performance**
Xeon Beckton	2010	128	8 (16)	2.26 GHz	32 GCUPS
Opteron Magny-Cours	2010	128	12 (12)	2.3 GHz	48 GCUPS
Opteron Interlagos	2011	128	16 (16)	2.3 GHz (est.)	64 GCUPS

Table 3.2: Estimated peak performance for current top-end and future CPUs

the clock speed and the known speed per core from Table 3.1 [4, 17]. Hereby we assumed linear scaling in the number of cores (suggested by Table 3.1), and the given performances may therefore not be reliable. Non-ideal inter-core communication, memory bandwidth limitations and shared caches could lead to a lower peak performance. Furthermore, no distinction in performance is made between Intel and AMD processors. Hence, Table 3.2 must be used as an indication where the S-W performance could go on modern and future CPUs.

3.2.4 Performance estimations for a multiple CPU system

With the current product line, up to four Opteron CPUs can be linked together via a Hyper Transport 3.0 bus [4]. We take a look at the estimated peak performance of such a system in Table 3.3, assuming no scalability loss due to communication. The fastest system shown can do 256 billion cell updates per second, which is already a lot, especially when you realize multicore processors are around for only roughly six years.

	1 CPU	**4 CPU**
Opteron Magny-Cours	48 GCUPS	192 GCUPS
Opteron Interlagos	64 GCUPS	256 GCUPS

Table 3.3: Estimated peak performance for a 4 CPU system

3.2.5 Summary

Throughout this section we talked about peak performance. Real, average performance could be as much as 25-50% lower [38]. We mainly discussed AMD processors; a study for Intel products will probably give you roughly the same numbers. What we can learn is that the current state of the art CPUs can already bring a lot of performance and that this can grow rapidly through upcoming products. CPUs are very flexible, scalable and a well known technology.

3.3 FPGA implementations

In this section we will take a look at FPGA implementations of the S-W algorithm. FPGAs are pieces of hardware which you can program. An algorithm can be designed using a hardware description language or tool, and mapped onto the flexible FPGA layout. The flexibility, difficulty of design as well as the performance of FPGA implementations fall typically somewhere between pure software running on a CPU and an ASIC (Application Specific Integrated Circuit).

FPGAs can be used to accelerate S-W. Implementations can rely on the ability to create building blocks (processing elements or PEs) that can update one matrix cell every clock cycle. Furthermore many of these building blocks can be linked together in a linear systolic array to create massive fine-grained parallelism. We will discuss these concepts before moving further.

3.3.1 The processing element

The PE is the workhorse of every FPGA S-W implementation. A example of how a S-W PE looks like can be seen in Figure 3.3. It typically consists out of three adders, three comparators and a score lookup, all needed for the calculation of a new matrix cell value. Most of the implementations also have a separate path that keeps track of the matrix maximum. Since these elements are the heart of the system, they need to be as fast and as small as possible. Fast PEs give rise to a full implementation that can run on high frequencies while small PEs let you use more of them.

3.3.2 The linear systolic array

The PEs are put together in a linear systolic array (LSA). Such an array works like the SSE unit in a modern CPU. But instead of having a fixed length of lets say 16, the FPGA based array can have any length. A example of the working of a LSA can be seen in Figure 3.4. Here we can see a 4×8 matrix being filled by a LSA of length 4. In the first clock cycle the first PE calculates the value of the upper-left matrix cell and passes it to the second PE. In the second clock cycle, the first PE calculates the value of the cell at (1,2), while the second PE calculates cell (2,1). In general, every PE handles one column, and its right neighbor PE lags one row behind to fulfill the data dependencies as discussed in Section 3.1.1.

In the example from Figure 3.4 the LSA can process a matrix with any number of rows. But, if the matrix would have had any other amount of columns, our LSA would

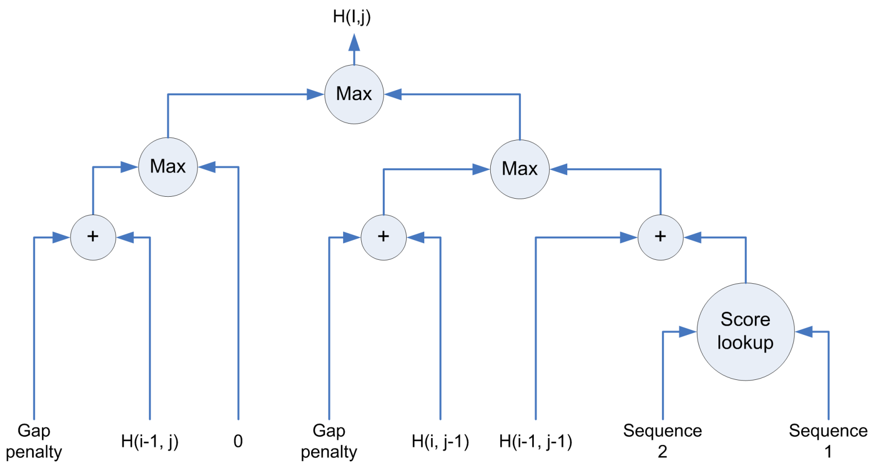


Figure 3.3: A view of a S-W processing element

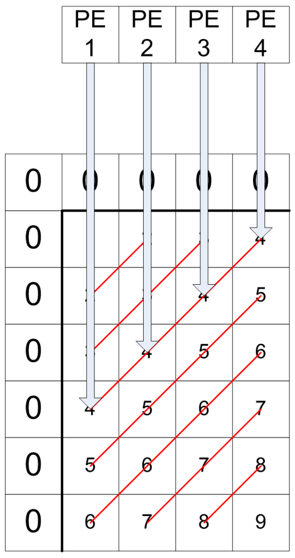


Figure 3.4: A view of a S-W linear systolic array processing an 4×6 matrix. The anti-diagonal lines show which cells can be calculated simultaneously

fail. Since the length of the used sequences can be considered random, the amount of columns in the matrix is random as well. Instead of making an implementation for every possible sequence length, you can clip the sequences to the length of your LSA, or pad the sequence with zeros till it fits the length of the LSA. When your sequence is at least twice the width of the LSA, you can shift your LSA to the right when you have reached the last row to process the next columns. This is also done in every SSE based software implementation, but doing it in hardware is far more complex. An example can be seen in Figure 3.5.

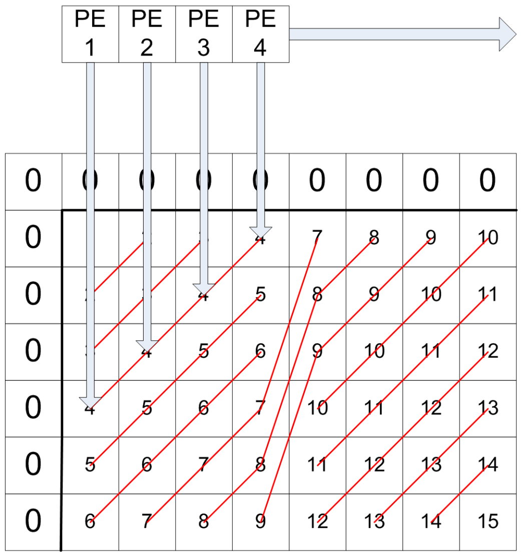


Figure 3.5: A S-W linear systolic array using shifting to process an 8×6 array. The anti-diagonal lines show which cells can be calculated simultaneously

3.3.3 Recursive variable expansion

The Recursive Variable Expansion (RVE) technique, introduced by Nawaz in [25], is a well discussed method to accelerate the S-W algorithm. The technique removes all data dependencies from an algorithm, so that it can be parallelized in ways previously impossible. This can best be explained with a figure. In Figure 3.6 we can again see three S-W matrices. From Section 3.1.1 we know we can, in the first iteration, only calculate the value of the shaded cell in the most left matrix. RVE makes it possible to calculate the value of every possible cell in the first iteration by removing the intermediate steps. This can be seen in the center and right-most matrix of Figure 3.6. Instead of having only three data dependencies with its upper-left neighbors, we now can have any amount of data dependencies, all back to the top row and left-most column of the matrix. It is clear we cannot use the S-W definition from Equation 2.1 anymore. To calculate the value of cell (2,2) (shaded in the right-most matrix in Figure 3.6) directly, we need to do a lot of extra work. This can be seen in Equation 3.1, using the same notation as in Equation 2.1. Instead of finding the maximum out of 4 equations, we now need to compare 8 equations, which are in themselves also more complex than in the reference case. The existing S-W implementations that use RVE are driven by PEs that can calculate the values of a 2×2 or 3×3 cell block every clock cycle. This results in a speedup of roughly a factor two and three respectively. The drawback of this technique is that the PEs become larger and the frequency on which they can run drops [15].

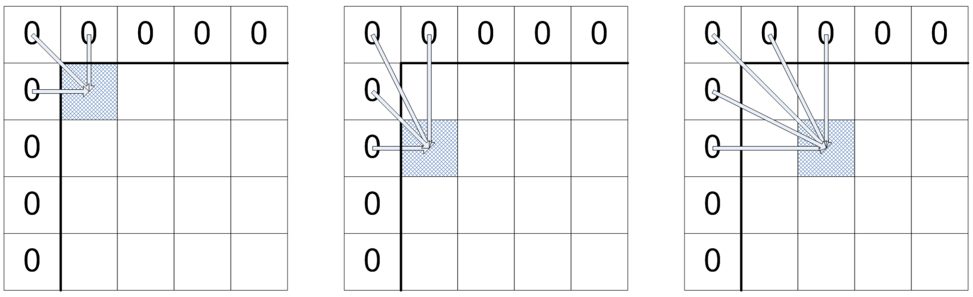


Figure 3.6: The data dependencies in a S-W matrix using RVE

$$H_{i,j} = max \begin{cases} H_{i,j-2} - 2\times\text{d}, \\ H_{i-1,j-2} - d + S_{i,j-1}, \\ H_{i-1,j-2} - d + S_{i,j}, \\ H_{i-2,j-2} + S_{i,j} + S_{i-1,j-1}, \\ H_{i-2,j-1} - d + S_{i,j}, \\ H_{i-2,j-1} - d + S_{i-1,j}, \\ H_{i-2,j} - 2\times\text{d}, \\ 0. \end{cases} \tag{3.1}$$

3.3.4 Modern implementations

In Section 3.2.2 we discussed some existing S-W implementations running on a CPU. A comparable analysis for FPGAs is a bit harder. There are very few real, complete implementations that give results that are usable. Most research implementations only discuss synthetic tests, giving very optimistic numbers for an implementation that could never be used in practice. Furthermore, there is a great variety in the type of FPGA used. Since all FPGA series have a different way of implementing circuitry, it is hard to make a fair comparison. To make things even worse, the performance of the implementations relies heavily on the data widths used. Smaller data widths lead to smaller PEs, which lead to faster implementations. Researchers have the tendency not to mention these numbers.

Nonetheless, a couple of implementations are mentioned in Table 3.4. The first two implementations make the points made above clear, using the same FPGA device, these two implementations differ a factor 23 in performance. The most reliable numbers are from Convey and SciEngines, and since Xilinx already released the Virtex 6 FPGA, these are also the ones that can be considered as being up-to-date. These implementations are also ‘real; they work in practice and are build for maximal performance.

	FPGA	**Freq.**	**#PEs**	**Perf. /FPGA**	**Perf. /system**
[**33**]	Virtex2 XC2V6000	180 MHz	7000	1260 GCUPS	
[**13**]	Virtex2 XC2V6000	112 MHz	482	54 GCUPS	
[**24**]	Virtex2 XC2VP30	79 MHz	20	6.3 GCUPS	
[**24**]	Virtex2 XC2VP30	73 MHz	12	7.8 GCUPS	
[**14**]	Virtex2 XC2VP30	10 MHz	100	4 GCUPS	
[**2**]	Stratix2 EP2S180	66.7 MHz	384	25.6 GCUPS	
Cray XD1 [**9**]	Virtex4	200 MHz	120	24.1 GCUPS	
Convey HC1 [**8**]	Virtex5 LX330	150 MHz	1152	172.8 GCUPS	691.2 GCUPS
SciEngines RIVYERA [**35**]	Spartan6 LX150	?	?	47 GCUPS	6046.0 GCUPS

Table 3.4: Performance of several FPGA implementations

3.3.5 Future FPGA implementations

The performance of S-W implementations on a FPGA can foremost be increased by using larger and faster FPGAs. Larger FPGAs can contain more PEs and therefore deliver more GCUPS. The largest Virtex 6 device has roughly 2.5 times more area than the largest Virtex 5 [41], so the peak performance of the former can be estimated at $2.5 \times 172.8 = 432$ GCUPS (using the Convey implementation). If some new technology (smaller gates for example) makes it possible to run at 200 MHz instead of 150 MHz, that would also give you a 33% increase in performance, bringing the total to 576 GCUPS. This is a major step forward, just by using a new device.

3.3.6 Summary

We can learn some interesting things from the numbers in Table 3.4. First, modern FPGAs can deliver a lot of performance. Second, it is really hard to make an estimation of the performance for a random FPGA given some results on other FPGAs. For a fair comparison every publication should offer the same functionality, and use a platform from the same product line.

3.4 GPU implementations

GPUs have in the last couple of years developed themselves from a fixed function graphics processing unit into a flexible platform that can be used for high performance computing. There are classes of algorithms that can run very efficient on these architectures, and some classes cannot. A very suitable task is the hashing of passwords; a very unsuitable task is unfortunately S-W. Even when looking at recent publications [20], GPUs do not present a major performance gain over CPUs. New GPUs will of course become faster, but so will CPUs. We therefore decided to do no extensive study on S-W performance on GPUs comparable to the ones presented in the previous sections.

3.5 Conclusion

From the summaries in Section 3.2.5 and Section 3.3.6 we can conclude that FPGAs are an interesting choice to accelerate S-W. Currently neither has a huge performance advantage over the other. But, software implementations can be considered to be optimal: years are spend optimizing them. When using FPGAs it might still be possible to make a significant contribution to the subject. Therefore we choose to go with FPGAs.

4 System implementation for accelerated S-W

In this chapter we will talk about our implementation of the S-W algorithm on a hybrid supercomputer. First, we will look at the way the S-W algorithm is used in practice in Section 4.1. Second, we will discuss the platform itself in Section 4.2. Items like the memory system and function dispatch will be explained. This followed by the system design in Section 4.3, the hardware design in Section 4.4 and the software design in Section 4.5.

4.1 Smith-Waterman in practice

Here we will take a look at how the discussed Smith-Waterman algorithm is used in practice. These details are necessary for making a good system implementation.

4.1.1 General usage

The S-W algorithm finds the optimal local alignment between two sequences. This alignment is what researches are interested in, but getting there requires some more steps. When a researcher has a piece of genetic sequence, DNA, protein or something else, it is usually aligned against a database of known sequences. This can for example be the SwissProt protein database, which contains hundreds of thousands of protein sequences from humans and animals [10]. The result is a top list of sequences that have the most resemblance with the query sequence. When necessary, it is possible to show the alignments of your query with those top sequences.

4.1.2 Evaluating results

To create a top list of alignments we have to use some kind of scoring model. The most important parameter is the maximal value found in the S-W similarity matrix. This maximum value gives us some idea about the similarity of the two sequences, but must be corrected for the sequence length. Longer sequences can reach a high matrix value due to a large amount of random hits, where small sequences cannot. The equation used to compute an alignment score can be seen in Equation 4.1 [26]. Here K and S are parameters that can be chosen by the user, and are meant to scale the score to useful proportions. The score can be interpreted as the probability that such a maximal value would be reached at random, so, the smaller the score, the better the alignment. There are all kinds of variations on this equation, all with a specific biological alignment problem in mind. This one is the most trivial, and used throughout our work.

$$Score = K \times \text{ sequence\_A\_length } \times \text{ sequence\_B\_length } \times \mathrm{e}^{-S*max} \tag{4.1}$$

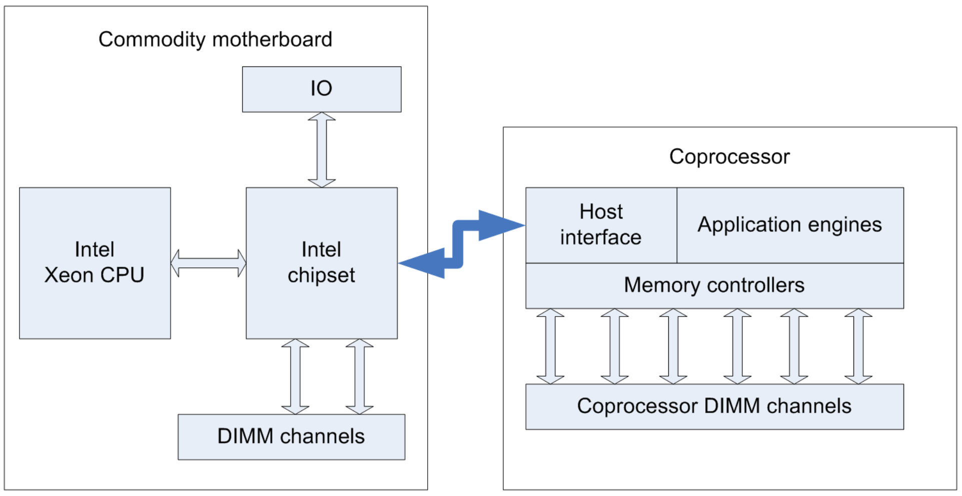


Figure 4.1: Overview of the HC-1 hybrid computer architecture

4.2 The hardware platform

In this section we will give a short introduction into the Convey HC-1 hybrid supercomputer. Several items like the memory organization and the instruction handling will be discussed.

4.2.1 System architecture

The HC-1 is a hybrid computer that consists out of an Intel Xeon processor on a commodity two socket motherboard. The other socket holds a FPGA based coprocessor. The coprocessor has its own high bandwidth memory system that is incorporated into the Intel coherent global memory space. This tight coupling makes it possible to see the coprocessor as extension to the Intel instruction set. An executable can contain both host and coprocessor instructions, and those instructions exist in the same virtual and physical address space.

An overview of the system can be seen in Figure 4.1

4.2.2 The coprocessor

The coprocessor consists out of three major building blocks. We have the Application Engine Hub (AEH), the Application Engines (AEs) and the Memory Controllers (MCs). An overview can be seen in Figure 4.2

Application engine hub

The AEH has a number of tasks. First it handles the communication between the host- and coprocessor. Second, it runs functions received from the host processor. These functions can consist out of for example memory operations, simple arithmetic operations, branches and the so-called custom instructions. The ‘normal’ scalar instructions are fed to the scalar processor to be processed, which is the third task of the AEH. The custom

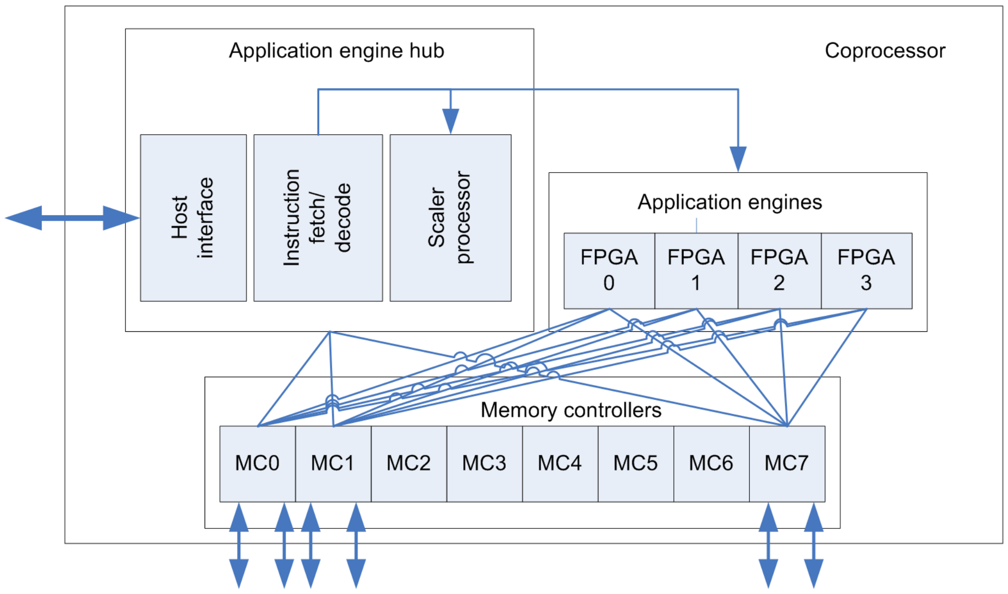


Figure 4.2: Overview of the HC-1 coprocessor

instructions are instructions defined by the user that describe a (possibly complex) operation implemented on the AEs, and are sent to those to be processed. These custom instructions are handled like any other, and it is here where the HC-1 really shows its hybrid nature. The AEH is furthermore connected to all eight MCs for memory operations. We will take a better look at how the dispatch of functions works in Section 4.2.4.

Application engine

The AEs are four Xilinx Virtex 5 LX330 FPGAs. They are used to implement user defined instructions. A small portion of the FPGAs are used for a Convey provided framework, to provide for example communication interfaces. The rest of the area is available for the user to implement their custom instruction. An implementation on the AEs is called a personality. So has Convey a personality to accelerate floating point operations, and a personality for complex financial operations. An implementation made by a user is also called a personality, or a Custom AE (CAE). Every FPGA is connected to eight memory controllers.

Memory controller

The coprocessor has eight memory controllers, each connected to two DIMM banks, resulting in 16 DDR2 memory channels. The memory systems provides a maximal bandwidth of 80GB/second. We will take a better look at the memory system of the HC-1 in Section 4.2.3.

4.2.3 Memory system

In this section we will discuss the memory model of the HC-1 computer in depth.

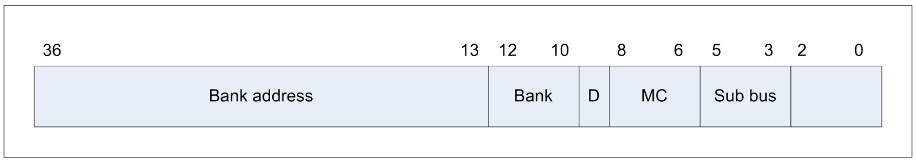


Figure 4.3: Composition of a virtual memory address

The memory model

Both the host- and coprocessor reside in the same virtual memory space. However, the memory is physically split in two, one part for the host-, and a second part for the coprocessor. Accessing data in the other physical part, introduces a large latency. Therefore it is necessary for the user to have its data in the right place, otherwise any computational performance advantage will likely be lost.

The MC - AE interface

Every AE has eight MC interfaces, connecting the user designed hardware with the MCs outside the AE. The links between the MCs and the interfaces on the AEs run at 300 MHz. Because running at this frequency is not possible for a lot of hardware designs, the 300Mhz link is split into two 150Mhz links, which are called the even and odd ports. So in the AE you basically have 16 MC interfaces available. Every interface has a request and a response port and it is possible to do a request and receive data every clock cycle, using memory blocks of one, two, four or eight bytes wide. This results in a maximal bandwidth of $2 \times 150\text{M}\times 8$ bytes= 2.4GB/second for every AE-MC interface. We have four AEs with each eight MC interfaces, so the total system bandwidth is $4 \times 8 \times 2.4 = 76.8$GB/second.

Virtual address composition

Every MC is connected to two out of the 16 DIMM banks, so not every physical memory location is reachable from every MC. The hardware design running on the AEs must therefore be careful which memory request to send to which MC. The system supports two interleave modes, 31/31, and binary interleave, but we will only discuss (and use) the latter. When using binary interleave, the virtual address is composed in a way visible in Figure 4.3. In this Figure D is the targeted DIMM, and MC is the memory controller. In practice, the hardware design should check bit 8:6 of the virtual address and send the request to the appropriate MC. To be able to run at (near) peak bandwidth, all the requests must be equally distributed over the MCs, but also over the banks and sub busses. Luckily, if we want to read a large array from memory in a linear fashion, using blocks of eight bytes, we automatically have to route our requests to the next MC every eight requests. A same analysis holds for the banks and sub busses. Therefore, when using the system in practice, we do not have to be really worried about equal load distribution.

4.2.4 Function dispatch

In this section we will discuss the function dispatch of the HC-1 computer in depth. Instructions that have to end up in the AEs go through the steps described below. A detailed case study can be seen in Section 4.5.3;

In the C/C++ code
The host program is typically written in C/C++ or in Fortran. If we want to make use of a custom instruction, we have to dispatch a function to the coprocessor. How this is done can be seen in Listing 4.1. In this example *fpga_call* is our function that should run on the coprocessor. By using the *copcall_nowait_fmt* function (provided by Convey), we dispatch our function to the coprocessor. The other parameters are *sig* to indicate which personality we want to use, *fpga_instr_handle* to keep track of our call and the last four, which are some user variables associated with the call.

On the coprocessor
Our *fpga_call* function runs on the coprocessor, and consists out of instructions defined by Conveys scalar instruction set. An example can be seen in Listing 4.2. The three move instructions move the *var*1, *var*2 and *var*3 parameters from their *aX* coprocessor registers to the *aeg* registers on the AE. Finally, the *caep*00 call calls the AE to execute custom instruction 00.

On the AE
The AE receives the *caep*00 instruction to be executed. When the AE is finished, it can return data to the coprocessor routine, or write a flag to memory for example.

Listing 4.1: Coprocessor call from C/C++ code
```
extern void fpga_call();

cny_handle_t fpga_instr_handle;
copcall_nowait_fmt  (sig, fpga_call, &fpga_instr_handle, "AAA", var1,
    var2, var3);
```

Listing 4.2: Assembly code running on the coprocessor
```
mov %a8, $1, %aeg
mov %a9, $2, %aeg
mov %a10, $3, %aeg

caep00
rtn
```

4.3 System design

In the current and following sections we will discuss the design of our S-W implementation. For our design we use the previously discussed HC-1 hybrid supercomputer, so we need a hardware design to run on the coprocessor, and a software program controlling the coprocessor and managing IO. First we will focus on some of the global design aspects.

4.3.1 Features

Here we will list some of the features that define our implementation. These are the features that set our design apart from other S-W implementations, and to which we will refer to later in the results section.

- The implementation should be as flexible and efficient as possible.
- The implementation should make use of the platform in a most efficient way by creating a synergetic combination between CPU and FPGA.
- The implementation should make use of the available FPGA area in a most efficient way.

4.3.2 Design decisions

Following the design features, we also made some major decisions before work started, but also during the work. Most of them are listed below. A couple decisions make our problem easier to handle. Given the small timeframe in which the implementation should be made, this was often necessary. Other decisions however make our work flexible and widely usable.

DNA alphabet and linear gaps
The S-W algorithm can be applied to all kinds of sequences, DNA, proteins etc. Since the use of proteins would require a more advanced design (Section 1.2.2) we decided to support only DNA. As discussed in Section 2.2.2 it is also possible to rewrite the S-W algorithm for affine gap penalties. Because this, also, leads to a more complex design, we restricted ourselves to linear gap penalties.

Instructions
Most (academic) S-W implementations [15, 24] name the two sequences involved *query sequence* and *database sequence*, following the general use of the algorithm, where you align a single sequence against a database. The query sequence is traditionally put on the top of the similarity matrix, and the database sequence on the side. One step further in this analysis can be to design your system with the assumption that the query sequence is fixed for a long period of time. This can reduce the complexity of your design significantly, but also reduces the flexibility. To make our system as flexible as possible, we did not use the query/database terms, but based the workload on *instructions*. An instruction is a packet consisting out of two sequences, *sequence A* and *sequence B*, which should be aligned. *Sequence A* is defined to be on the top of the matrix. Every instruction can contain two new sequences, so no simplification of the problem is made. All the information in an instruction can be seen below:

- SequenceA, sequenceB: the memory addresses of the sequences. These base pairs are stored in the host processor part of the memory.
- SequenceA compressed, sequenceB compressed: the memory address of the compressed sequences. These compressed base pairs are held in coprocessor memory.

- DataB: the memory address of an array to store a part of the similarity matrix, held in coprocessor memory.
- Done, max, last_column_max: the memory addresses of the three results, held in host processor memory.

Optimal instruction processing
Instructions containing one of two small sequences are send to the CPU to be processed. The instructions containing large sequences are sent to the coprocessor. Since our LSA has a fixed length, we have to either clip *sequence A* to a multiple of the LSA length, or pad it with *null* characters to obtain the right length (Section 3.3.2). Because sequence padding results in useless computations, we decided to go with clipping. The resulting partly filled matrix can be finished by the CPU. A schematic view of the system can be seen in Figure 4.4. The design constantly keeps track of the best alignments. Since the number of alignments is typically much larger than the number of items in the result queue, the probability of an alignment to end up in that queue is small. This fact is used to minimize the necessarily to finish the partly, by the AE, filled matrix by the CPU. The coprocessor reports not only the maximum value of the alignment, but also the maximum value of the last processed column. We know how much of the matrix is left uncalculated, so we can easily calculate the maximum score that could possibly have been reached by this alignment. If the alignment, even with this maximal possible score, does not end up in top list, we do not have to finish the alignment by the CPU.

Multiple linear systolic arrays
The FPGAs used in the coprocessor are large enough to contain several hundred PEs. Since one very large LSA would only allow aligning long sequences, we decided to split the available area into pieces and create multiple LSAs. This way we can align multiple small or large sequences at once, and this also reduces the amount of idle PEs at startup and finish time. An complete analysis of this is done by Hasan [16] and his work shows that using multiple smaller LSAs pays off in execution time and hardware utilization. We decided to go with LSAs of length 64, what would give us the possibility to use around eight of them.

Main memory as temporally storage
Modern FPGAs have large amounts of BRAM available. The FPGAs used in the HC-1 contain almost 12 megabyte of memory that is accessible in a single clock cycle [41]. It is a very attractive to use this memory for the storage of your sequences and matrix data. But as all good things, it has some drawbacks. So is it nearly impossible to use a large part of this memory as one, at high speeds: the placement and routing needed for a real implementation will become too complex. Another negative point is that, when subtracting the amount of RAM needed for example for the MCs, and dividing the remaining part among several LSAs, we end up with not that much memory at all. Since we want to be able to align very long sequences, we decided to not make use of the BRAM as sequence or data storage, and fetch/store everything from/in the main memory.

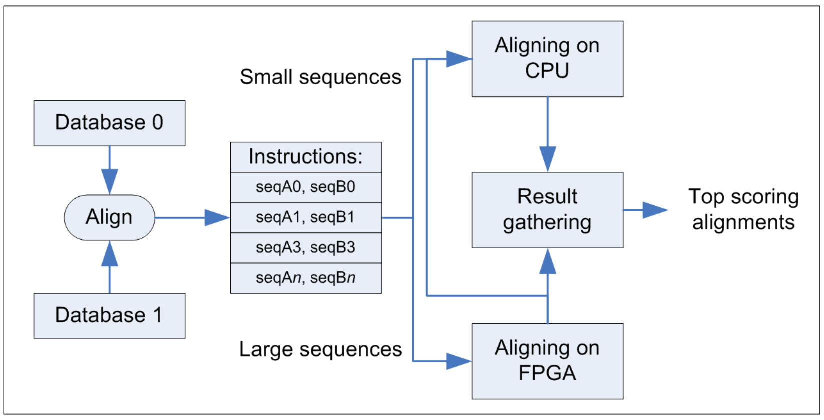


Figure 4.4: System level view of our S-W implementation

Data compression
All the instructions are kept in memory, so if we want to store many of them at the same time, they need to be as small as possible. Furthermore, the coprocessor needs to request all its data from memory, restricted by its bandwidth. To keep the memory footprint as low as possible, we compress the sequence and matrix data to a minimum.

The sequence data consist out of the four different DNA bases. This gives us the ability to express a base in two bits. The largest memory read data width supported by the HC-1 is 64 bits, so we are able to store 32 bases in one variable, and read those from memory in one call. The naive method would be using eight bit chars or even 32 bit integers to store the sequences, so we achieved a compression ratio of four up to 16.

The matrix data used to process the instruction can be compressed as well. Here we use the fact that the value in every matrix cell can only differ a maximum value from its top neighbor, defined by the score model. If we use four bits to represent our scores, we can represent the difference between two adjacent matrix cells in five bits. This can be shown with a small example on sketch paper, and will not be done here. This behavior is used to compress the matrix data differentially. Instead of storing the (for example) full 16 bits of matrix data, we only store the five bit difference between a cell and its top neighbor. In this case this results in a compression ratio of more than three, and the ability to fetch 12 matrix values in one memory call.

4.4 Hardware design

In this section we will cover the hardware design of our accelerated S-W implementation. Several of the major building blocks are discussed. In Figure 4.5 we can see the global hardware layout, with the instruction dispatch, multiple functional units (FUs) which keep and manage the LSAs and a memory crossbar. Instructions enter the design via the instruction dispatch and are put in a queue. If a FU is idle, it receives an instruction from this queue and starts processing. The FU computes, reads and writes data from memory, and eventually writes the results to memory, to go idle again.

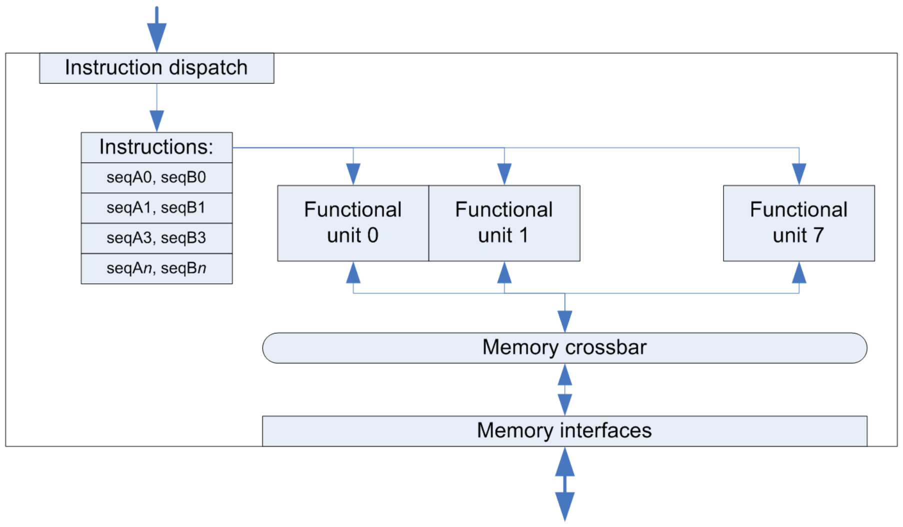


Figure 4.5: Our hardware design. This design is hold by a single AE.

4.4.1 The processing element and the linear systolic array

In order to use the optimal PE in our design, we did an extensive study. This is covered separately, in Chapter 5. To use the PE in practice, we had to choose a data width for the matrix values and the scores. Larger widths would allow the alignment of larger sequences with higher/lower scores, but would also reduce the maximum frequency and increase the area per PE. We found a tradeoff in 12 bits for the matrix values, and four bits for the scores. To have an effective data width of 12 bits, the PE internally must use 13 bits, since we need a sign bit for negative values.

The linear systolic array

The concept of the LSA is discussed in Section 3.3.2. In our implementation the LSA behaves a bit differently, because we use pipelined RVE PEs. It takes two clock cycles before the values of the two matrix cells are computed, which means the second PE can only start after the second clock cycle, instead of after the first as in the reference design. Together with the RVE size ratio, this results in a one to four ratio in 'matrix fill speed' for the x and y direction. When fully operational, the LSA covers a sub matrix of 64×256 cells.

To make use of the LSA in practice, we need some extra control signals. They are listed below:

- A signal to indicate that the input data is valid.
- A signal to reset the stored maximum value when we start a new alignment.
- A signal to reset the stored matrix values every time we start a new 64 column wide sub matrix (macro column).

These signals make the PE a bit larger, and also a bit slower.

4.4.2 The functional unit

This is the most complex part of the hardware design. In its core it has a LSA of optimized PEs, surrounded by queues and logic to manage the instruction processing. First we will discuss the functionality of the FU, followed by its design.

Functionality

The FU is designed to deliver a sustained peak performance, or in other words, process multiple instructions in line having all the PEs busy at all time. The issued instructions can contain sequences of any length, so the FU must implements the ‘array shifting’ technique as shown in Figure 3.5. How the FU processes a single instruction can be seen in Figure 4.6. The Figure shows a matrix with *sequence A*, *sequence B* and *data B*. Here *data B* is the linear block of memory provided by the instruction for temporary storage in the main memory. It can be seen as a column of the similarity matrix. At startup time, it is filled with zeros, by definition. The LSA is shown in two instances; in the first it is processing the first column of 64 bases wide, in the second instance the second macro column. In both cases the first PE of the LSA fetches a *sequence B* and a *data B* item, and all PEs fetch *sequence A* data. The last PE of the array writes its output data back to the *data B* memory. When the LSA starts a new macro column, it automatically reads the right *data B* data. Using this method, the LSA can process multiple macro columns without any drawbacks. The memory requirements are constant and the complexity is totally hidden from the LSA.

The switching between instructions is also hidden from the LSA. A management block detects when it has read the last data from the current instruction, and asks for a new one. The data from the new instruction is pushed directly behind the old data in several queues. Internal counters keep track of when the first instruction is finished and the second started, so that the results get written to memory at the right moment. This way the FU can process multiple instructions seamlessly.

Design

Figure 4.7 shows the design of the FU. The working is explained below.

- The LSA picks its data from three different queues. These queues need to hold as much data as the LSA needs in the worst case memory access time. In the HC-1 this can be up to 1000 cycles, so we need 128×64bit *data B* memory and 64×64bit *sequence B* memory. The *data B* data, before entering the LSA, needs to be differential decompressed. The *data B* data leaving the LSA, is stored in the *data B output* queue (8×64bit) per two items. If we would store them item wise, and pack them together just before sending the data to the store queue, we would create an unstable situation. When needing cycles to send the results, the *data B output* queue would slowly grow and overflow after a couple of instructions.

- The *read request manager* is responsible for enough data in the *sequence A*, *sequence B* and *data B* queues. It creates read requests, puts those in the *read request queue* (8×56bit), and keeps track of the amount of requests in-flight. The request type is stored in another queue (256×2bit), so we can identify the

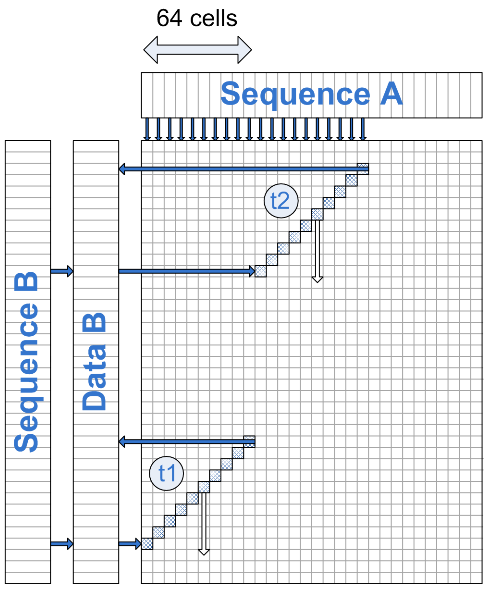


Figure 4.6: The FU processing two large sequences

responses later on. When the last piece of data for an instruction is requested, further requests are stalled and the FU manager is signaled. If the *read request queue* is gets full, due to memory crossbar congestion, requests are stalled.

- In the center of the FU we find the *FU manager*. This block keeps track of the position of the LSA in the matrix and sets some signals appropriately. For example, when the LSA enters a new macro column, we need to reset the latched matrix values in every PE, to make a fresh start. When the last matrix cell has been processed, it signals the store manager that the results are valid. The manager also manages the receiving of instructions.
- The *store manager* picks data from the *data B* output queue, compresses it and puts it in the *store request queue* (8×112bit). As with the read requests, store requests are stalled when the queue tends to overflow. When the results are valid, the *store manager* switches from sending *data B* data, to sending the results to the store request queue.
- Memory responses entering the FU are reordered in a reorder queue (256×64bit).

The queue and reorder queue modules are given by Convey.

4.4.3 The memory crossbar

Since we cannot just send every memory request to every MC (as discussed in Section 4.2.3), we must use a memory crossbar. Using this, the LSA can keep an abstract view of the memory, and does not have to be concerned with picking the right MC. At the time we needed it, Convey could not deliver a working version of a crossbar, therefore we created our own.

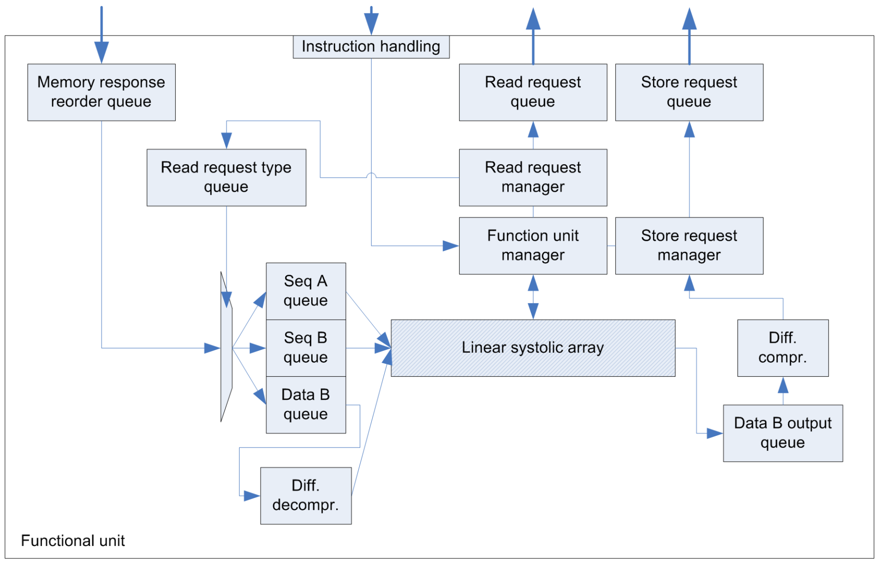


Figure 4.7: Design of a FU

Our hardware design does not require memory bandwidth in the order of the limits of the system, but only around 0.4GB/second per FU or 3GB/second per AE. Nonetheless, we designed a crossbar that could, theoretically, deliver a peak bandwidth of around 20GB/second per AE. This gives us the freedom for potential bandwidth demanding changes later on, and more importantly, it gives us the security of a stable system. Before diving into the design, we take a look at the request density that the crossbar should be able to handle. In Table 4.1 we can see how often requests of a certain type are send to the crossbar per FU and in Table 4.2 we can see the probability that a number of requests go to the same MC. These numbers are obtained from a small simulation, assuming that the address distribution is random, which is a safe one. In Appendix A we discuss this tool in more detail. The moment at which read requests are created by every FU depends partly on the moment they receive other data from the MCs, which is random.

The read requests per FU add up to one request every 5.3 clocks ($16/3 = 5.3$), so in the worst case, where all the FUs read requests are lined up, the crossbar needs to be able to serve eight read requests in 5.3 clocks. From Table 4.2 we can learn that this is possible most of the time, only in approximately one percent of the cases five or more requests point to the same MC. However small, it does mean we need to have a queue in the FU for those cases. All the read requests (and their response) are handled by the even MC ports.

Store requests from all FUs lay a pressure of eight requests per eight clocks on the crossbar, which is just on the boundary of needing a queue. To avoid problems, the FU uses one. All the store requests are handled by the odd MC ports and therefore independent from the read requests.

Memory responses, which enter the crossbar through the even response ports of every MC, behave in a random way. Responses can enter in any order and at any rate; in the worst case every MC can put out a continuous stream of responses all for the same FU. Since the FU can only process items from one queue at a time, all the responses for other FUs are blocked. It is hard to get numbers on this behavior, but by using fairly large queues (64 responses deep) we should be fine.

4.4.3.1 Request composition

In Figure 4.8 we can see the composition of read/store requests leaving the FUs and the responses leaving the MCs.

Read requests
Read requests leaving a FU consist out of a 48 bit virtual address and an eight bit request id. The request id is used by the FU to indentify and reorder the response later on. In the crossbar every read requests gets a three bit FU id which is padded to the request id and together with 53 zeros, send to the 64 bit wide read control input of the right MC. The FU id is used by the crossbar to send the response to the right FU. The virtual address is passed to its 48 bit wide input.

Store requests
Store requests consist only out of a virtual address and data. The 48 bit address and the 64 bit data are sent to the right MC for processing.

Memory response
A memory response enters the crossbar via two inputs. The read control input holds the FU and request id, and the data input holds the data. The three bit FU id is used to identify the right FU, and the data together with the request id is send to it.

4.4.3.2 Crossbar layout

Every FU connected to the crossbar has a read request queue, a store request queue and a memory response finder. Every MC connected to the crossbar has a read request finder, a store request finder and a memory response queue. All the finders check all their queue counterparts for valid data, what means having the right FU or MC mask. When valid data is found, it is passed on to the MC or FU, and a *served* signal is send to the queue. To avoid timing issues, this signal is latched. This however means that we cannot check the same queue twice in a row, since the old data is still there for one clock cycle. This is solved by checking the first four queues at every even clock, and the other four at odd clocks.

This model makes it possible to achieve high bandwidths. From a MC point of view, it is possible to achieve a peak bandwidth in the order of the peak of the system. To make this happen the top item in every request queue should point to a different MC, and the top item in every memory response queue should point to a different FU. In addition, every successive queue item should point to the other four-pack of MCs/FUs as described above. That way, every clock cylce the crossbar can process 16 requests and eight responses, good for almost 20GB/second of bandwidth per AE.

Request type	**Period**
SeqA read	Very large
SeqB read	16 clocks
DataB read	8 clocks
DataB store	8 clocks
Results store	Very large

Table 4.1: Request density to the crossbar per FU

Requests to the same MC	**Probability**
2	0.862
3	0.415
4	0.080
5	0.009
6	0.001

Table 4.2: Probability of request collision, using 8 MCs/FUs

However, from a FU standpoint, which is the one that matters, we can only reach half of that. Since the *served* signal is latched one clock cycle, every request takes at least two cycles to leave the queue. This results in a peak bandwidth of 10GB/second per AE: still enough for our implementation.

4.4.4 Design and testing

Here we will take a brief look at the design and test process, and at the process of generating hardware.

Design and HDL simulation

For the design of the hardware of our S-W implementation, we used a bottom up approach. The first thing we made was the PE, followed by the LSA and the FU. The memory crossbar and the rest of the system followed after. Testing was done in the same order, first only the PE, then the LSA containing the PEs, and later the FU containing the LSA. Testing the crossbar was done only in combination with the FU. To make this possible we had to simulate the MCs behavior in our testbench. This testbench was later extended so support multiple FUs, to make the test of the crossbar more meaningful. The final, total system, was not tested with our own testbench but with the Convey Architecture Simulator (CAS).

Convey architecture simulator

The CAS is a simulation suite provided by Convey, to test a CAE as if it was part of the coprocessor, instead of testing it as a separate part. The CAS simulates everything on the coprocessor, including the MCs. Input to the simulator is given by a user created software program, which can be exactly the same as the final program the user would use to control the real CAE. The simulation is therefore realistic and shows the behavior

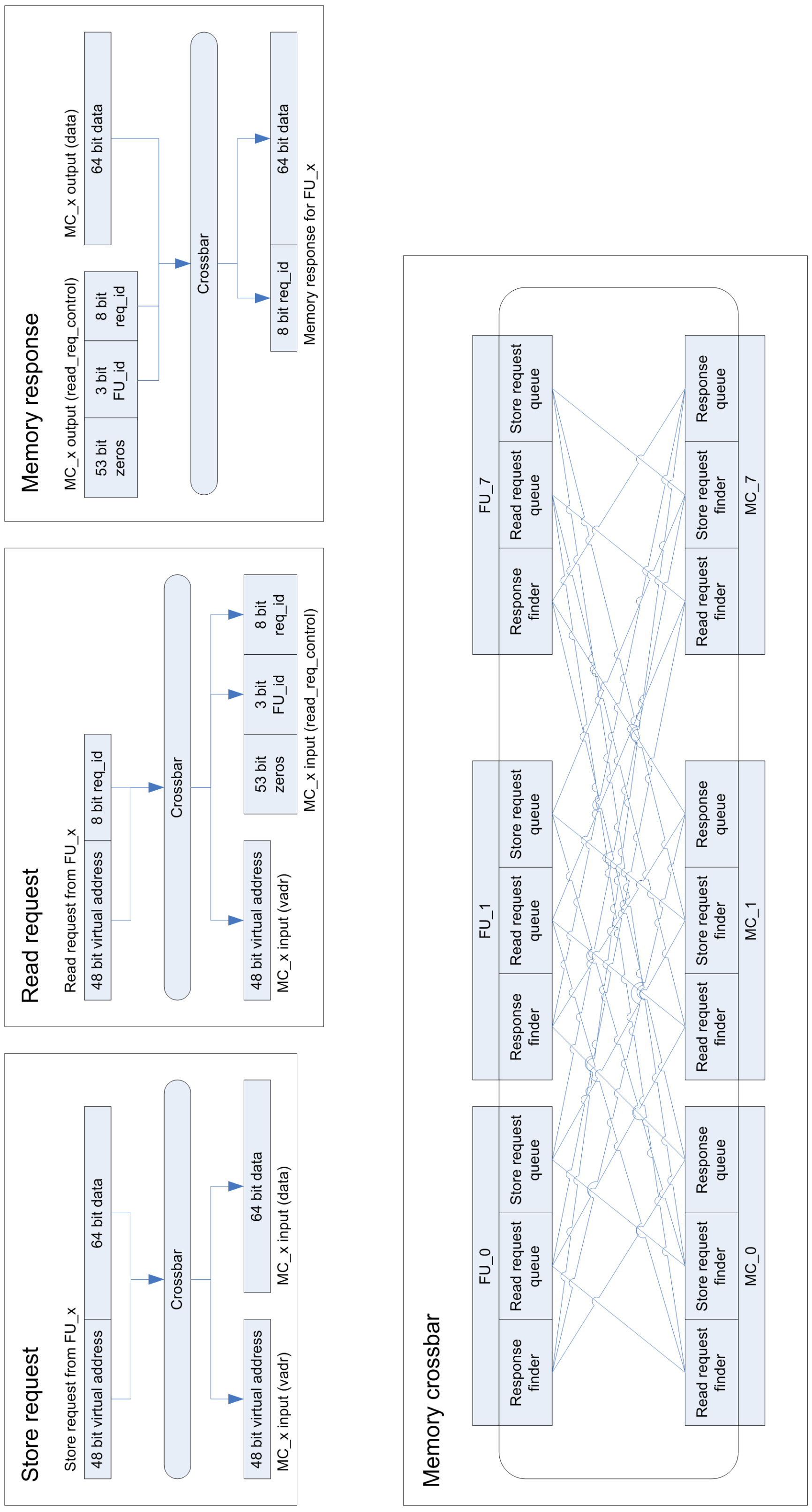


Figure 4.8: Composition of memory requests/response and crossbar layout

of your total design. Running a design in the simulator is obviously slower than it would in reality.

Our S-W implementation was tested in this way and adjusted until its behavior was as intended.

4.4.5 Limitations

However designed with flexibility in mind, our implementation does have some limitations. We will discuss the most important ones.

Bitwidths vs. sequence length
While the internal counters in the FU support sequences up to 16777216 base pairs, the length is in practice limited by the bit width of the similarity matrix and the score scheme. If we would use a (realistic) match score of +2, and have equal sequences, the registers in the PEs will overflow after 2048 ($2^{12}/2 = 2048$) columns. In the worst case, when using a match score of +7, this will happen after 585 ($2^{12}/7 \simeq 585$) columns. It will be up to the user to be aware of this; the FU does not have any detection of avoidance for this limitation.

Memory latency vs. sequence length
Due to the large memory latency of the HC-1, the *data B* and *sequence B* buffers need to be large. This also means that, since the data is fetched 2000 rows ahead, the minimum length of sequence B is 2256 (2000 + 256 for the default LSA behavior) base pairs. Because the data leaving the LSA is stored with latency as well, the final minimal length is even larger. We choose a value of 3072 base pairs, assuming storing a value will not take more than 816 clock cycles. This limitation is handled by the software running on the host CPU.

This limitation could be solved by inserting a small memory cache in the hardware design. FUs aligning small sequences could use that memory instead of the main, to store *dataB* and *sequenceB* data. The FU could for example include a flag with every memory request, whether it request is part of a small alignment or not. The crossbar could have some queues to store the data. Implementing this and achieving the right timings might not be easy.

Clipping in the sequence B direction
Our memory requests for the sequences consist out of 64 bits, or 32 bases. For *sequenceA* this is no problem, since it is already clipped at multiples of 64 bases. For *sequenceB* it is a problem, since by clipping the sequence to fit in whole memory reads, we can miss up to 31 rows in our processing step. Our implementation does currently not take care of this, foremost because it is hard to fix, but also because *sequenceB* is in most cases quite long, so there is only a small chance that the alignment will be in the bottom 32 rows.

	LUTs	LUT %	Registers	Registers %	Frequency (MHz)
Function unit	22662	10,93%	10761	5,19%	186
Memory Xbar (8 FUs)	9352	4,51%	6132	2,96%	224
Convey logic	18448	8,90%	18448	8,90%	unkown
System (7 FUs)	186434	89,91%	61372	29,60%	186
System (8 FUs)	209096	100,84%	67504	32,55%	186

Table 4.3: Resource utilization of our design

4.4.6 Final implementation

Here we will discuss the final specifications or our implementation. Because there is a long way between the first design planning and the real implementation, we will split this section in two. First we will discuss the implementation that is synthesizable and fits on the device. Second, we will discuss the steps we took to make a realizable implementation and the specifications of the result.

4.4.6.1 Synthesizable implementation

Our final system consists out of multiple FUs, the memory crossbar, the instruction queue logic and some logic from Convey. In Table 4.3 we can see the amount resources needed by the main building blocks. The size of the crossbar depends on the amount of FUs attached; here we gave it eight slots.

We can see that it is possible to fit up to seven FUs on the device before the synthesize tool thinks we run out of lookup tables (LUTs). A complete implementation with this amount of FUs will synthesize and still has the possibility to run at 186 MHz.

4.4.6.2 Realizable implementation

When the implementation was found to be synthesizable, we continued to make it implementable as well. After synthesizing the design it must be mapped in the resources available on the Virtex 5 device. For example, the logic is mapped to look up tables (LUTs). Note the difference with the synthesize tool, which only makes an estimation of the amount of LUTs needed. After the complete design is represented in available resources, it can be placed on the FGPA. Everything must be placed in such a way that paths between registers are as short as possible. When everything is placed, the route tool lays down the actual routing. To do the placement and routing for a large design with strict timing requirements is a difficult task and can take up to a day to complete. To make this process possible, we had to make several adjustment and additions to our design:

- Make a floor plan to place our logic on sense full places. This helps the placer by significantly reducing the search space, and gives us some idea of which wires should be fine, and which can become a problem.

- Insert registers/buffers in long wires to make the timing possible. For example, the final interconnect wires between our crossbar and the memory interfaces are latched. Furthermore the crossbar needed some major adjustment as described in the next paragraph.
- Rewrite some logic to avoid large feedback loops across the entire design. For example, the serving of instructions to FUs is changed so that a FU does not have to check every other FU whether it has the right to fetch an instruction. This can be done by latching signals and making the control statements a bit more complex.

Floorplan
The designed floor plan can be seen in Figure 4.9. All the main design components are shown. The *instruction dispatch* and all the *MCs* are placed by Convey, the other placement is based upon an example from Convey, but adjusted to fit our needs. All the logic of a certain component must be fitted inside its region, but other logic is also possible to add. The large blocks, 0 till 5, are the FUs; they are connected to the crossbar via small buffers in *Xbar connection hubA/B*. The $FU0$, $FU1$ and $FU3$ blocks are connected to hub A, the other three to hub B. Those hubs are connected to both *MC connection hub A* and *MC connection hub B*, which are in turn connected to $MC0, 1, 4, 5$ and $MC2, 3, 5, 7$, respectively. This way, in a few hops, every FU is connected to every MC. This hub based design is very necessary for a correct placement. Without it the design would reach about 10% of its intended frequency.

The instruction queue is located in the center of the design, next to the dispatch, so that every FU can pick instructions from it easily.

Final specifications
While it might be possible to put seven FUs on the FPGA, we used only six, as can be seen on the floor plan. This way the FU placement could be done in a 2×3 fashion, and the connections to the crossbar are short and symmetric. Doing so for one FU more might be a lot harder. The design was placed and routed for a 150 MHz clock.

4.5 User application design

The coprocessor is controlled by a program running on the Intel Xeon processor, written in C++. The main tasks of this program are: feeding instructions to the CAE, running S-W on small sequences and completing partly filled matrices returned by the CAE. Most of the code is hidden from the user in a S-W library, which has a easy to use interface. This way users can very easily speed up their alignments without knowing anything about the underlying technology.

4.5.1 Program layout

The three tasks described above need to be performed simultaneously, and at high speed. We therefore choose to use multiple threads (posix threads, [39]), taking advantage of the multicore CPU. The storage of data as well as the communication between threads

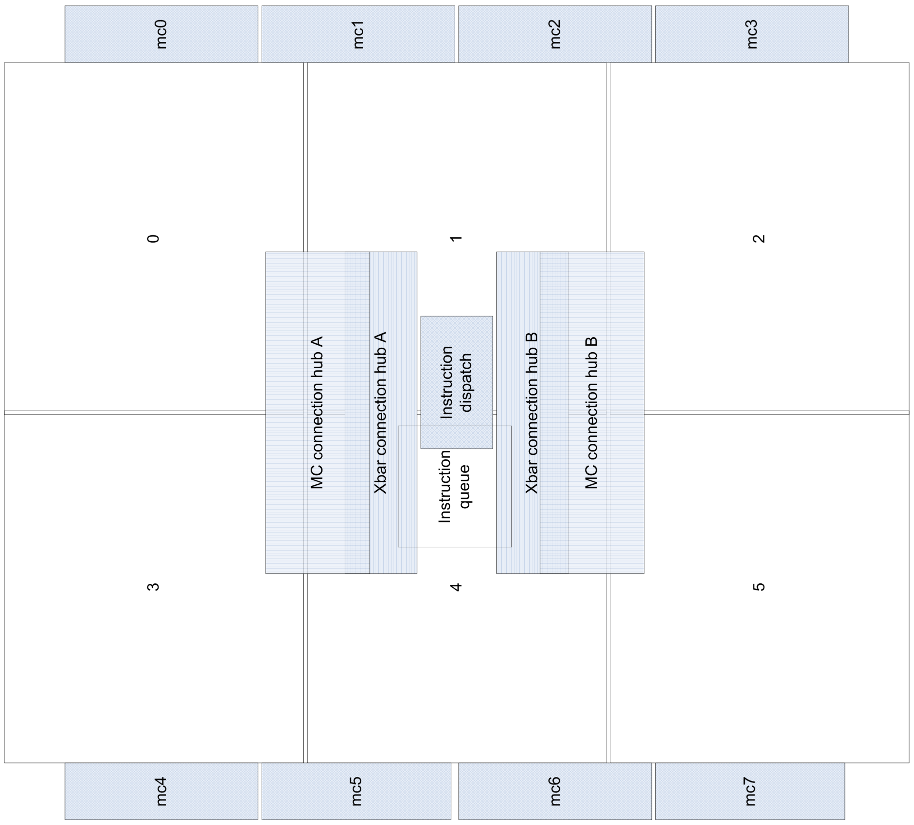


Figure 4.9: Floor plan of the FPGA

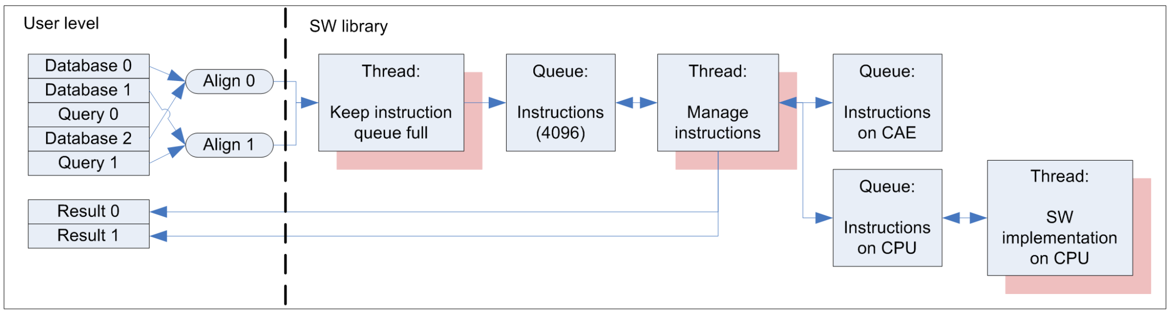

User level
SW library
Database 0
Database 1
Query 0
Database 2
Query 1
Align 0
Align 1
Result 0
Result 1
Thread: Keep instruction queue full
Queue: Instructions (4096)
Thread: Manage instructions
Queue: Instructions on CAE
Queue: Instructions on CPU
Thread: SW implementation on CPU

Figure 4.10: Overview of the software program

is done via queues. An overview of the threads, their tasks and the queues is visible in Figure 4.10. On the left side of this figure we can see the part that is visible to the end user, on the right side the S-W library.

4.5.2 Program workings

The user loads a number of sequence databases into the program. When finished, multiple S-W library calls can be done to align two databases or a database and a query with each other. One of the arguments for this call is a *result_packet*, in which the library stores the best alignments. When the alignment process is finished, the user can read his results.

- First, the user gives the assignment to load a certain database into the memory. The sequences are stored in host memory, the compressed sequences in coprocessor memory. When this is done, the user can give alignment instructions to the library, which passes them to a thread which converts them to instructions. Since a single instruction can, depending on the size of the sequences used, occupy a considerable amount of memory, we limit the amount of instructions in the system to 4096. The mentioned thread checks for finished instructions and overwrites them with new ones, until all the work is done. This process can be seen as creating instructions just in time. Note that all the sequence data resides in memory all the time, so there is no bottleneck there.
- All these instructions are kept in a queue, and are constantly checked by the *instruction management thread*. This thread is the heart of the program and has multiple functions. It tries to find suitable instructions for the CAE, and the CPU S-W implementation; when found, those instructions are put in a CAE/CPU queue. On the other hand, the *instruction management thread* checks those two queues for finished instructions, and does the final processing. This means checking whether the resulting alignment should be put into the results list, or whether a CAE instruction should be finished by the CPU. The score of an alignment is calculated with Equation 4.1.
- The *S-W CPU thread* picks instructions from the CPU thread, and processes them with a scalar S-W implementation. If the selected instruction is a partly finished CAE instruction, the thread needs to decompress the *data B* data before it can start. The matrices are processed column wise, and at every end it is checked whether we need to finish the rest of the matrix. So here we also try to do only the minimal amount of computations.

When all the instructions are processed, the user is signaled and can read the results.

4.5.3 Sending instructions to the coprocessor

Sending instructions to the coprocessor has proven to be a challenging task. A brief description of this process can be seen in Section 4.2.4, here we will describe the final design.

When we do a *copcall_nowait_fmt* call, we dispatch a function to the coprocessor. This function can have multiple *caep* instructions, intended for the AEs. In our design, one alignment instruction equals one *caep* instruction. The AE can halt the dispatch of caep instructions by asserting a *stalled* signal. When an AE is finished with all the instructions (and there are no more pending memory requests), it can assert the *idle* signal. When the coprocessor receives this, the function dispatch is done. Only then, the user can dispatch a new function to the coprocessor.

Hereby it is clear, that if we want to have multiple alignment instructions running on the AEs, we cannot have only one *caep* instruction per function dispatch, since we can only send the next function dispatch after the first is finished (i.e. the AE went *idle*). To overcome this limitation, we designed a coprocessor function running a polling loop. In the loop, a conditional statement checks the status of a *data_ready* flag. If the host software sets this flag to 1 (the host- and coprocessor are in the same memory space), the coprocessor function fetches data from a predefined location and sends a *caep* instruction to the AEs. The *data_ready* flag is then set to 0 by the coprocessor and it returns to its polling state. On the host processor code, the flag is checked for being 0, only then the data repository can be overwritten and the flag changed.

This way, we can send any number of *caep* instructions to the AEs with only one *copcall_nowait_fmt* call or function dispatch.

Optimal processing element design

5

Since the PE is the workhorse of any S-W hardware implementation, we will try to make an optimal one. It must be as small as possible, while having the possibility to run at high clock speeds. To do this, we turned to the RVE technique, introduced in Section 3.3.3. A lot of research has been done on square RVE PEs which can update four or nine matrix cells at once. We are interested in the behavior of larger and non-square RVE PEs. To make this possible we will introduce a new, automated, way of creating the hardware circuitry for a broad selection of PE sizes. Our approach will be checked against published work and performance characteristics will be extracted. Finally we will propose a PE design that performance better than a default one.

Throughout this chapter we will use the $X \times Y$ notation for indicating the dimensions of a RVE PE. So a 2×3 PE means a rectangular PE which updates six matrix cells every clock cycle.

5.1 Recursive Variable Expansion in depth

In this section we will add some depth to the already provided introduction into the subject of RVE.

The main concept of RVE is explained in Section 3.3.3. Here we will explain how we derived Equation 3.1. If we want to calculate the value of a matrix cell at (2,2) in the first iteration, we need to replace the data dependencies with the formulas to calculate that data. This process is made visible in the top row of figures in Figure 5.1. The expansion is split into three parts for better understanding. If we count all the paths from the cell in question to the edges of the matrix we find 13 of them. In other words, the $max\{...$ statement in $H_{i,j} = max\{...$ would contain 13 equations. Luckily, we can do better than that. Taking a better look at the resulting equations, we can see that the result of some of them will always be smaller than the result of others. So is there an equation consisting out of two match scores, and an equation from the same start, consisting out of one match score and two gap penalties. When using practical score models, the latter will always be smaller than the former and can therefore be dropped. Following this analysis we can reduce the 13 equations to seven, shown in Equation 3.1 and the bottom row of Figure 5.1. This optimization process is an important part of the RVE technique. Without it we would create an enormous amount of useless hardware, reducing the performance.

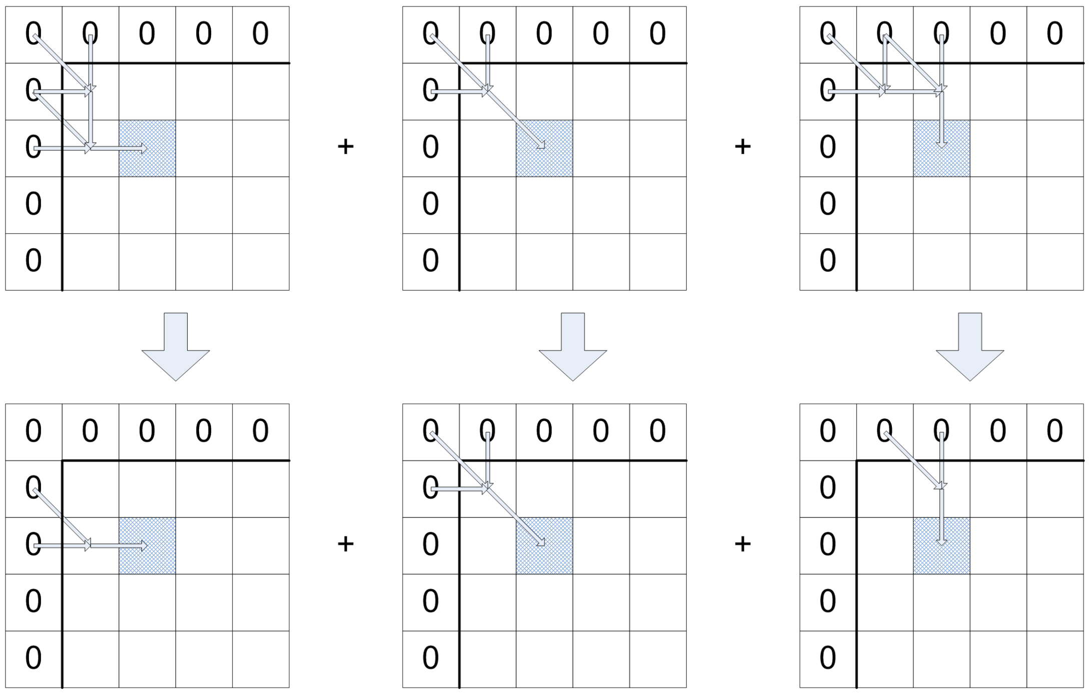


Figure 5.1: Elimination of unnecessary equations

5.2 Automated design

In this section we will discuss the creation of a tool for automatic RVE S-W hardware generation. The existing RVE S-W hardware can update four or nine matrix cells at ones. These designs where made manually and took a lot of effort. Because we want to investigate many more (and larger) designs, we made a tool to do it for us.

The created program works in several steps. First, the program generates all equations of which the maximum should be found for a certain (any) cell. These equations get optimized by removing all those which never lead to the maximum (as explained in Section 5.1). This is mainly done by an exhaustive worst/best case analysis. In Table 5.1 we can see the amount of equations involved for some square RVE PEs, as well as the amount of equations that got cancelled. From the table we can easily see that it indeed becomes impractical to do the cancellation by hand for larger PEs. The resulting equations are evaluated through random testing. Their result is compared with that of a default S-W implementation. This ensures us that we have not missed an equation. On the other hand, the program also checks whether every equation leads to the maximum at least once. This way we can see that we do not have too many equations left. An example of the program output for one cell can be seen in Listing 5.1, where *F[ ]* is the input, *g* is the gap penalty and *x[ ]* is the match score.

In Appendix B we will discuss the workings of our RVE hardware generation tool in depth.

	Total equations	**Cancelled equations**	**Equations left**	**Lines of HDL**
1×1	3	0	3	72
2×2	23	8	15	169
3×3	127	80	47	351
4×4	679	552	127	731
5×5	3651	3320	331	1491

Table 5.1: Equations and lines of HDL for different RVE PE sizes

Listing 5.1: Automatically generated equations for a cell
```
0:  F[0,-3] + 3g
1:  F[-1,-3] + 2g  + x[0,-2]
2:  F[-1,-3] + 2g  + x[0,-1]
3:  F[-2,-3] + 1g  + x[0,-1] + x[-1,-2]
4:  F[-1,-3] + 2g  + x[0,0]
5:  F[-2,-3] + 1g  + x[0,0] + x[-1,-2]
6:  F[-2,-3] + 1g  + x[0,0] + x[-1,-1]
7:  F[-3,-3] + 0g  + x[0,0] + x[-1,-1] + x[-2,-2]
8:  F[-3,-2] + 1g  + x[0,0] + x[-1,-1]
9:  F[-3,-2] + 1g  + x[0,0] + x[-2,-1]
10: F[-3,-1] + 2g  + x[0,0]
11: F[-3,-2] + 1g  + x[-1,0] + x[-2,-1]
12: F[-3,-1] + 2g  + x[-1,0]
13: F[-3,-1] + 2g  + x[-2,0]
14: F[-3,0] + 3g

eq: 63 cancelled: 48 left: 15
no unused equations
```

From the equations we generate a high level hardware description in Verilog. The generated module is a combinatorial circuit without registers. Besides this module, the program also generates a linear array layout which connects all the combinatorial blocks to registers. In the last column of Table 5.1 we can see the amount of code needed to describe the combinatorial behaviour. This column suggests that designing these complex circuits by hand would be hard and error sensitive. The combinatorial RVE PEs were found to be correct for some simple test cases. A code snippet of automatically generated Verilog can be seen in code Listing 5.2.

Listing 5.2: Snippet of automatically generated HDL

```
function signed [0:9] PE_RVE_1_3_function;
input [0:1] seqA;
input [0:5] seqB;
input [0:9] dataA;
input [0:39] dataB;
input [0:3] match_score, mismatch_score, gap_penalty;

reg signed [0:9] match_0_0; reg signed [0:9] match_0_1; reg signed [0:9]
    match_0_2;
reg signed [0:9] gap_1; reg signed [0:9] gap_2; reg signed [0:9] gap_3;
reg signed [0:9] equation_0; reg signed [0:9] equation_1; reg signed [0:9]
    equation_2;
reg signed [0:9] equation_3; reg signed [0:9] equation_4;
reg signed [0:9] max;
reg signed [0:9] max_0; reg signed [0:9] max_1; reg signed [0:9] max_2;
reg signed [0:9] max_3; reg signed [0:9] max_4; reg signed [0:9] temp_max;

begin

    match_0_2 = getMatchValue(seqA[0:1], seqB[0:1], match_score,
        mismatch_score);
    match_0_1 = getMatchValue(seqA[0:1], seqB[2:3], match_score,
        mismatch_score);
    match_0_0 = getMatchValue(seqA[0:1], seqB[4:5], match_score,
        mismatch_score);

    gap_1 = {6'b111111, gap_penalty};
    gap_2 = (gap_1 <<< 1);
    gap_3 = (gap_1 <<< 1) + gap_1;

    equation_0 = dataA[0:9] + gap_3;
    equation_1 = dataB[0:9] + match_0_2 + gap_2;
    equation_2 = dataB[10:19] + match_0_1 + gap_1;
    equation_3 = dataB[20:29] + match_0_0;
    equation_4 = dataB[30:39] + gap_1;

    max_0 = getMax(0, equation_0);
    max_1 = getMax(equation_1, equation_2);
    max_2 = getMax(equation_3, equation_4);
    max_3 = getMax(max_0, max_1);
    max_4 = getMax(max_2, max_3);
    max = max_4;

    PE_RVE_1_3_function = max;

end
endfunction
```

5.3 Results

In this section we will discuss how our new implementation holds against previously published work. After that we will see how it behaves with respect to frequency, latency and throughput. To create all the results, we used a 36×36 S-W matrix.

5.3.1 Comparison with previous work

The two major aspects of every high performance hardware design is its area, and its frequency of operation. For the RVE implementations of S-W this is not different. The, by Nawaz, handmade 2×2 and 3×3 designs were optimized to an incredible extend and it is probably safe to say that those designs are optimal regarding hardware usage and (theoretical) latency. For the automated case we assumed that a lot of optimizations will be done by the compiler (Xilinx XST).

In Table 5.2 we can see a comparison between the hardware usage of some automatically generated RVE PEs versus the handmade ones. The second column shows the amount of adders in the resulting design, as well as the amount of '+' signs in the Verilog code. If the compiler would not do any optimizations, these two numbers would be equal. However, they are not, so the compiler does some common sub expression elimination. If we compare the numbers to the handmade designs, we can see that the compiler does not come close to the optimal case. It can be said that detecting which terms must be added first for optimal hardware savings is too hard to be left to the compiler. After investigation, it was found that every equation is plainly processed from left to right, barely paying attention to common sub expression detection.

What makes the comparison even worse for the automated case, is that we only generated hardware for the outer cells, while Nawaz' solution also computes the value for the inner cells. For the 2×2 case this does not lead to more adders, but for the 3x3 and larger cases, it will.

Regarding the comparators, the automated case comes much closer to the handmade numbers. This is due to the fact that most of the comparators are used for the tree that finds the maximum of every equation result. Creating such a tree is pretty straightforward and generating efficient Verilog code is therefore easy.

As said, besides area, the maximal frequency is an important aspect of a hardware design. However, this very much depends on the target FPGA and some design specifications, such as data width. We did not have the tools to test our design on the same hardware as the existing solutions; therefore it is not possible to make a fair comparison. But, it is safe to say, that our new implementation runs at frequencies at least as good as the existing ones. Giving the compiler a high level behavioral description will probably lead to faster hardware than with a low level schematic, as is done by Nawaz, since the compiler is better aware of the underlying platform behavior than a hardly trained human.

From the paragraphs above we can conclude that area wise our solution is not optimal, but it is at least as good frequency wise. An investigation in how we can modify our

	Automated design		**Handmade design**	
	Adders (in code)	Comparators	Adders	Comparators
2×2	19 (22)	16	14	17
3×3	85 (98)	63	54	54
4×4	300 (333)	188	unkown	unkown
5×5	916 (1069)	481	unkown	unkown

Table 5.2: Comparison between our automated and existing RVE designs

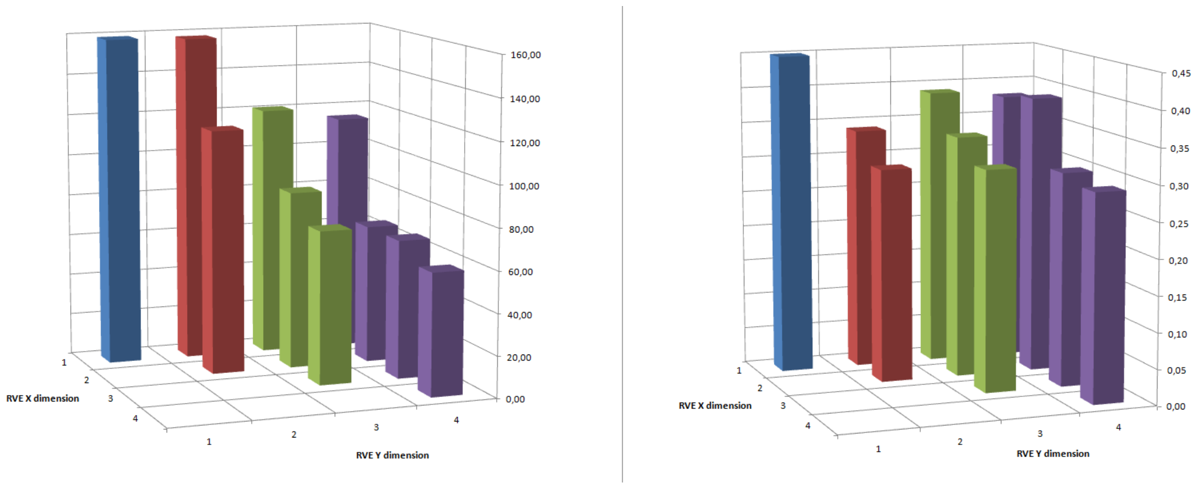


Figure 5.2: Left: Maximal frequency (MHz) of an PE versus its RVE dimensions. Right: Time required (ns) to fill a 36×36 matrix versus the RVE dimension of the used PE

behavioral description so that the compiler can find common sub expressions, would be worth an effort.

5.3.2 Frequency and latency

The research performed on RVE hardware focuses mainly on the latency between calculating the first and the last value of the matrix. In other words, how fast the matrix is filled. Therefore, our research starts here as well. First, we will look at the frequency each design can run on; this can be seen in Figure 5.2. The main diagonal represents the square cases, and it is clear that every increase in size must be paid with a lower frequency. All other forms run on lower frequencies as well, but the way the frequency drops is less clear. For example, if we look at the 1×1 and 1×2 PE, we can see that they can run at almost the same frequency, while the 1×3 PE has to give away 30 MHz. This cannot be explained by the way the formulas expand (number of sequential additions) and the depth of the 'find maximum' tree. For the 1×1, 1×2 and 1×3 case, the addition depth is respectively two, three and three. The maximum tree depth is respectively two, three and three. This shows that the way the circuitry gets implemented at a low level plays a major role in determining the maximum frequency, and not only the theoretical logic depth. The large drop in frequency from 1×n to 2×n designs come partly from

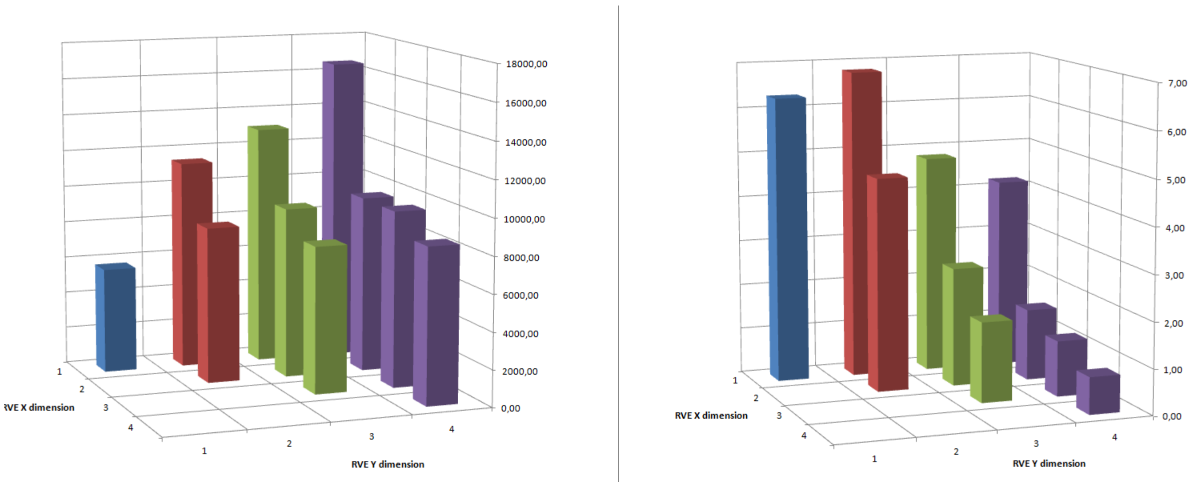


Figure 5.3: Left: Throughput (MCUPS) of a linear array versus its RVE dimensions. Right: Throughput per area (GCUPS/slice) versus its RVE dimensions

the fact that with $1\times n$, we do not have to take care of the clipping error mentioned in Nawaz' work [23], while for $2\times n$ we do.

$$Latency = \frac{(36/PE\_X) + (36/PE\_Y) - 1}{Frequency} \tag{5.1}$$

From the frequency, we can calculate the latency with Formula 5.1. The results are shown in Figure 5.2. For the square cases, we can see that the time required to fill the matrix becomes smaller for larger PE sizes. This is expected and confirms Nawaz' work, and in fact the cornerstone of the research in RVE. The improvement from 2×2 up becomes negligible, and suggests that there is not really a reason to go beyond this size.

The latency tells us how fast the matrix will be filled, but since this is always in the order of milliseconds or less for practical cases, it is not really of interest. Far more important is how much work can be done every time unit. This is the throughput of the system, and will be discussed in the next section.

5.3.3 Throughput and performance per area

In Section 5.3.2, we looked at the latency of different RVE designs, and showed that latency is not really a useful metric. Here, we will look at the throughput of the array, or in other words the amount of matrix cell updates, it can perform per second. In Figure 5.3 we can see the throughput for the same RVE designs. We calculated the values using Formula 5.2. As with the latency, we can see that the throughput becomes better from 1×1 to 2×2 PEs, and that the improvement comes to almost a halt for larger designs. This is due to the fact that the frequency rapidly decreases for larger PEs. Only the $1\times n$ PEs become increasingly better, this is due to the fact that the frequency does not drop that fast here, as mentioned before in Section 5.3.2.

$$Throughput = PEs \times PE\_size \times Frequency \tag{5.2}$$

Since our research is targeting a real implementation, only looking at the throughput is not enough. We are making an implementation that will use every part of the FPGA to squeeze every bit of performance out of it. Therefore we also have to look at the area every implementation takes, giving rise to the throughput/area metric. We can calculate this following Formula 5.3. As area metric we took the estimated amount of Virtex 5 slices the map tool gave us. The results are shown in Figure 5.3. It becomes clear that, when taking the area into account, RVE does not perform better than the default 1×1 case at all for almost every case. Only the 1×2 RVE PE gives a better throughput per area ratio than the default case. We will take a better look at this case in Section 5.4.

$$Throughput\ per\ area = \frac{Throughput}{Number\ of\ slices} \tag{5.3}$$

5.4 The 1×2 RVE processing element

Since the 1×2 RVE case is the only design that gives a better performance per area ratio than the default 1×1 design, we will examine this case further in this section.

5.4.1 Basic hardware design

First, we created a new 1×1 and 1×2 design. By designing them ourselves, we could be sure they were both optimal. To evaluate the designs properly, we would need the maximal frequency it can run on, and the area it takes. The former is easy, when synthesizing the XST tool tells us the worst case register to register delay, and thereby also the maximal frequency. Getting the accurate area is a bit harder. A lot of papers synthesize their design for a Virtex 2 or Virtex 4 device. For these architectures the XST tool makes an estimation of the number of slices the design would occupy. Researches typically use this estimation as area metric. When synthesizing for Virtex 5 devices, you only get the number of LUTs and registers the XST tool thinks the design would use. To be as complete as possible, we did both ways of area measurement. The results can be seen in Table 5.3. In the first two columns we can see the different area and frequency data for the new 1×1 and 1×2 designs. In the bottom rows several throughput per area ratios are shown. It can be seen that the 1×2 design performs worse regarding Virtex 5 LUTs as well as Virtex 4 slices. Since a Virtex 5 device contains the same amount of LUTs as registers, we are mainly interested in the one that performs worse of these two, the LUTs in this case. This result contradicts the results from the previous section, but this can be explained by the fact that the automatically generated designs where only rough approximations of the best results, focusing on the global picture of RVE behavior. With the new designs, the 1×1 is relatively better. To improve the performance of the 1×2 design, we turned to the technique of pipelining.

The schematic of the (relevant aspects of the) 1×2 design can be seen in Figure 5.5. For better understanding, a top level view of the data and sequence in- and outputs can be seen in Figure 5.4.

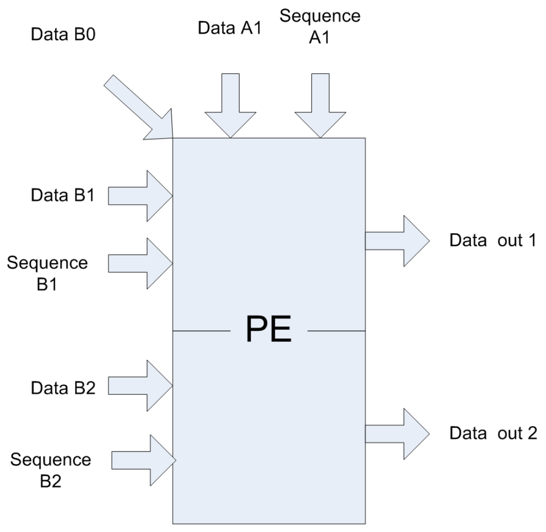


Figure 5.4: Dataflow around a 1×2 RVE PE

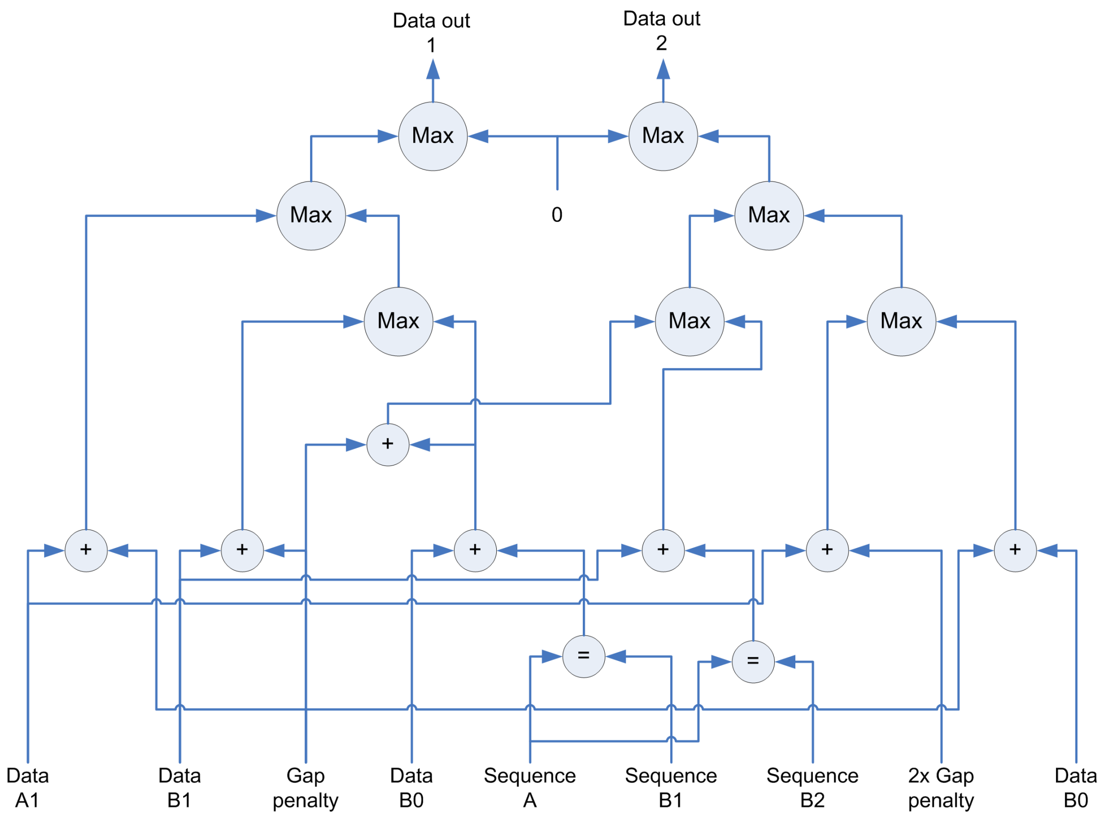


Figure 5.5: A 1×2 RVE PE

	1x1	1x2	1x1 pipelined	1x2 pipelined
Virtex 5 LUTs	180	376	180	343
Virtex 5 registers	48	64	96	112
Virtex 4 slices	97	188	107	166
Virtex 5 frequency (Mhz)	176	148	226	186
Virtex 4 frequency (Mhz)	141	121	169	130
CU/clock	1	2	1	2
Throughput Virtex 5 (MCUPS)	176	296	226	372
Throughput Virtex 4 (MCUPS)	141	242	169	260
Throughput / Virtex 5 LUT	0,98	0,79	1,26	1,08
Throughput / Virtex 5 register	3,67	4,63	2,35	3,32
Throughput / Virtex 4 slice	1,45	1,29	1,58	1,57

Table 5.3: Results for various 1x1 and 1x2 designs

5.4.2 Hardware pipelining and optimization

Pipelining is a well known technique for increasing the maximal frequency and throughput of a hardware design by placing registers between combinatorial logic parts. This is also used by Nawaz to increase the frequency of his designs.

To see whether we could get the 1×2 design to perform better than the 1×1, we made a pipelined version of both. The results can be seen in Table 5.3. The pipelined 1×1 design performance better than the default 1×1 design regarding Virtex 5 LUTs and Virtex 4 slices. It uses more registers, which is not compensated for entirely by the higher frequency, therefore the throughput per register is worse. The results for the pipelined 1×2 PE can be seen in the last column. For this version we did not place the registers on the most trivial place, between the adders and the find maximum tree. Instead we tried to optimize the design in such a way that we ended up in the best possible throughput per area ratio's. These optimizations came down to the following points:

- Removing as many adders and comparators as possible. This is done by rewriting some of the equations. For example, $max(A + C, B + C)$ can be reduced to $max(A, B) + C$. By doing so we remove one adder.
- Minimizing the amount of intermediate registers. This is achieved by doing as much reduction (by the *max* operation) as possible in the first pipeline stage.
- Balancing the pipeline in such a way that the delay of both stages is as equal as possible.

The resulting PE has a better Virtex 5 LUT and Virtex 4 slice performance than the default 1×1 PE, which is a good result. When we compare it however with the pipelined 1×1 PE, it is equally good regarding Virtex 4 slices, but worse when looking at the Virtex 5 LUTs. Finally we choose to use the pipelined 1×2 PE for two reasons. First, given the way other researchers present their work, we tended to go with the Virtex 4

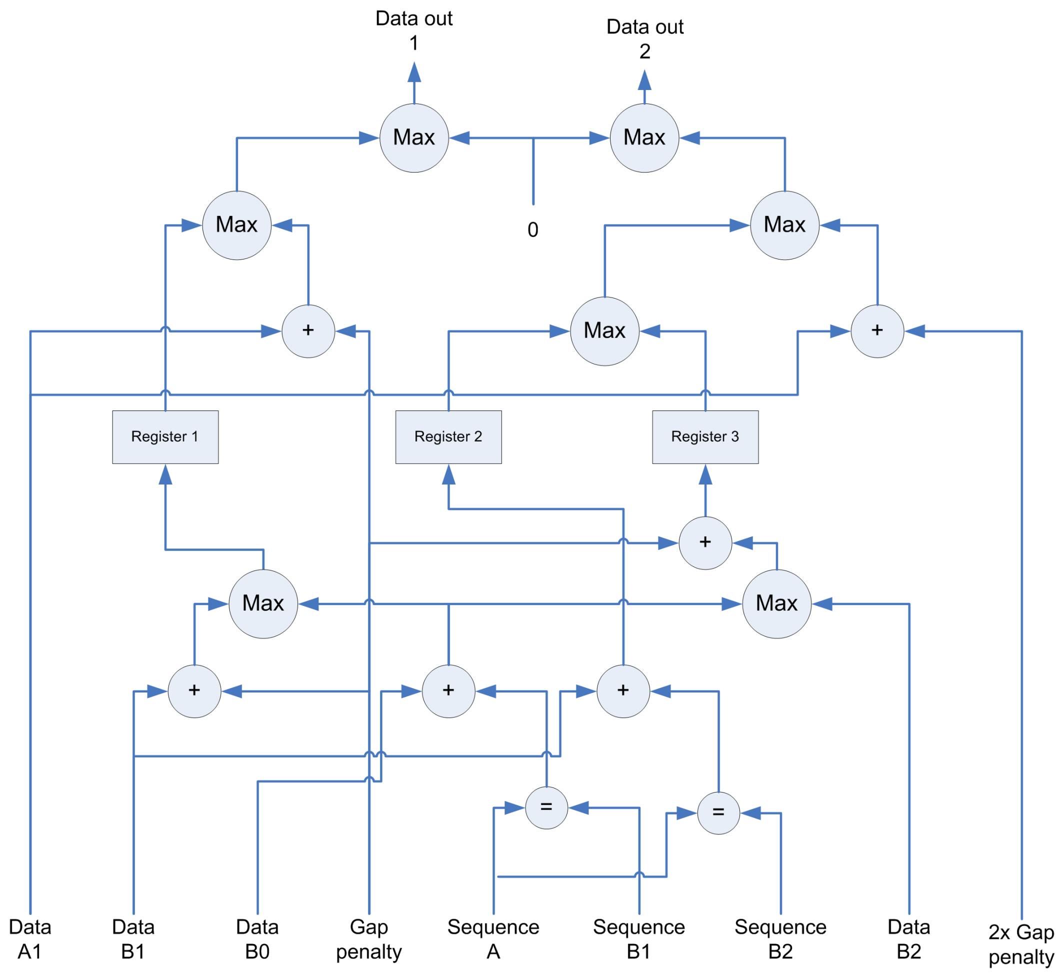


Figure 5.6: View of an optimized and pipelined 1×2 RVE PE

slices as most important metric. Second, the RVE technique was suppose to be one of the pillars on which this entire research should lean. Therefore we really wanted to use it.

The schematic of the pipelined and optimized 1×2 PE can be seen in Figure 5.6. An interesting thing to point out is that *Data A*1 makes its entrance in the circuitry in the second stage. Using this value in the first stage would create a data hazard, since it is still unknown by then. The *Data B*1, *B*2, *B*3 and the sequence data is available from the start, and can therefore be used in the first pipeline stage.

5.5 Further exploration of PE sizes

Seeing the good results from the 1×2 PE, we took a step back and wondered whether the used optimization techniques could be applied to other designs as well.

5.5.1 The 2×1 RVE processing element

Instead of 1×2, we could also position the PE in a 2×1 manner. This however, introduces some difficulties. As pointed out at the end of Section 5.4.2, when using a 1×2 PE, we need to wait for the value of *Data A*1 to become ready, before we can use it in a calculation. If we would rotate the PE 90 degrees, we would not only have *DataA*1 unavailable in the first stage, but also a *Data A*2. These data dependencies lay some extra pressure on the second pipeline stage, bringing the whole design out of balance. The maximal frequency of a 2×1 PE is considerably lower than of its 1×2 counterpart.

5.5.2 The 1×3 RVE processing element

Despite the results from Section 5.3.3, we wanted to try using the optimization techniques discussed on a 1×3 PE, and hopefully end up with even better results. Without going into details, we can say that this is not possible. Going from 1×2 to 1×3 gives you a 50% boost in performance, while it is impossible to keep the growth of the hardware design below 50%.

5.6 Conclusion

In this section we will draw some conclusion from our research in developing a fast and efficient RVE based S-W PE.

Automated design and results
From our research into automatically generating RVE hardware, some things became clear:

- The main optimizations of RVE hardware design cannot be handed to the XST compiler. It does a poor job at detecting common expressions and the possibility to rewrite (restructure) some parts.
- The maximal frequency of a PE is not as trivial as just the logic depth. This became clear in numerous occasions, and showed that it is, for example, sometimes not better to eliminate all common expressions due to a latency penalty. So when designing circuitry, the underlying platform must always be taken into account.
- The RVE solutions published so far are not better if we take the area into account. Our exploration suggests that it is worth it to look at rectangular cases like 1×2 PEs instead of square cases.

The 1×2 RVE PE
We showed that it is possible to create an RVE PE that performs better than a default implementation, even when taking the area into account. The new design is almost 10% better then a non pipelined 1×1 PE. When comparing it with a pipelined 1×1 PE it is not clear which is best. We decided to go with a RVE PE because of mostly soft reasons. We will come back to this subject in Section 7.1.2.

6 Implementation results

In this chapter we will take a look at how our implementation holds against several tests. During the work we clearly did not only focus on GCUPS, but also on flexibility and efficiency. We introduced a new paradigm to look at S-W processing, where workload might be cancelled, and where the FPGA and CPU work together. Some of the results are aimed at validating this concept, while other are still aimed at raw GCUPS. At the end we take another look at S-W performance on different hardware platforms and compare our implementation to a selection of other FPGA based realizations.

6.1 Isolated results

In this section we will look at the performance of several design blocks in simulation. Several performance metrics will be extracted, and those will be mirrored to the theoretical and ideal behavior.

6.1.1 Functional unit performance

Here we will simulate our FU design, and see whether the performance is as expected. We wrote a small program that could generate the necessary Verilog statements for a sense full test, and put those in a Verilog testbench format. In addition, we wrote a testbench module to simulate the memory system and instruction dispatch, so we could focus on the part in question. First we run two simulations of a single instruction, followed by simulations of a stream of instructions. Timing was started at the moment the LSA started processing, so the time for prefetching the data required for the first instruction is discarded. We stopped recording at the moment the maximum value was written to the *store request queue*. The results can be seen in Table 6.1. Here the first column shows how many instructions we are simulating, followed by the sequence lengths, the timings, the cell updates done, the performance and the percentage of the theoretical peak. The theoretical peak of our FU is 19.2 GCUPS, calculated by multiplying the frequency with the amount of PEs and the amount of cell updates per PE per clock cycle (150M$\times$64$\times$2=19.2G).

Result A shows us that our FU does not run at peak performance, but at only 96.86% of that. Result B, also one instruction, reaches 99.20%. This difference can be explained by the startup and finish time required by the LSA. In the first 128 clock cycles, not every PE is used, which is also true for the last 128 clock cycles. This amount of cycles follows from the LSA length and the two pipeline stages. Since the sequences in result B are larger, the time that PEs lay idle is relatively smaller. We can easily calculate whether the not optimal performance really comes from this behavior. If we add 128 clock cycles to the start time, and subtract 128 clock cycles from the finish, we have

	Seq. A length	Seq. B length	Start at (us)	Write max at (us)	Total time (us)	MCU done	GCUPS	% peak
1 instr. (A)	128	4096	2,749	30,940	28,191	0,524	18,598	96,86%
1 instr. (B)	512	4096	2,789	112,892	110,103	2,097	19,047	99,20%
4 instr. (C)	256	3072	2,789	44,632				
	512	4096		153,848				
	768	5120		358,628				
	1024	6144		686,275	683,486	13,107	19,177	99,88%
8 instr. (D)	128	3072	2,749	24,115				
	256	4096		78,722				
	384	5120		181,112				
	512	6144		344,936				
	640	7168		583,845				
	768	8192		911,492				
	896	9216		1341,529				
	1024	10240		1887,608	1884,859	36,176	19,193	99,96%

Table 6.1: Results for simulating the functional unit

the time that the LSA is fully utilized. To correct the workload, we have to subtract two triangles of matrix cells, 64×256 wide. For Result A, the total processing time than becomes 26.741us, and the total workload 0,508 MCUs. A simple division results in 19,187 GCUPS, or 99.93% of the peak performance. The small difference left is caused by a couple of extra register layers before data enters the LSA, and before the maximum value is really written.

As we look at result C and result D, it is clear that the performance of our FU reaches the theoretical limit. The time percentage some of the PEs are idle becomes negligibly small, and the FU delivers a sustained peak performance while processing multiple instruction, containing sequences of different lengths. This was one of the design goals set in Section 4.4.2.

6.1.2 Memory crossbar performance

In this section we will look at the synthetic and practical performance of our crossbar. To recap, we designed the crossbar so that it could deliver a comfortable amount of bandwidth, far more than we actually needed. We wrote a small program to create Verilog testbenches and used those to put our design to the test. First we take a look at some synthetic tests, followed by more realistic scenarios.

Synthetic tests

To get an idea of the crossbar performance, we connected it to six queues, acting as six FUs. Those queues where filled with 64 items each, and when they where full, the crossbar could start serving them. The time was recorded from the moment the crossbar

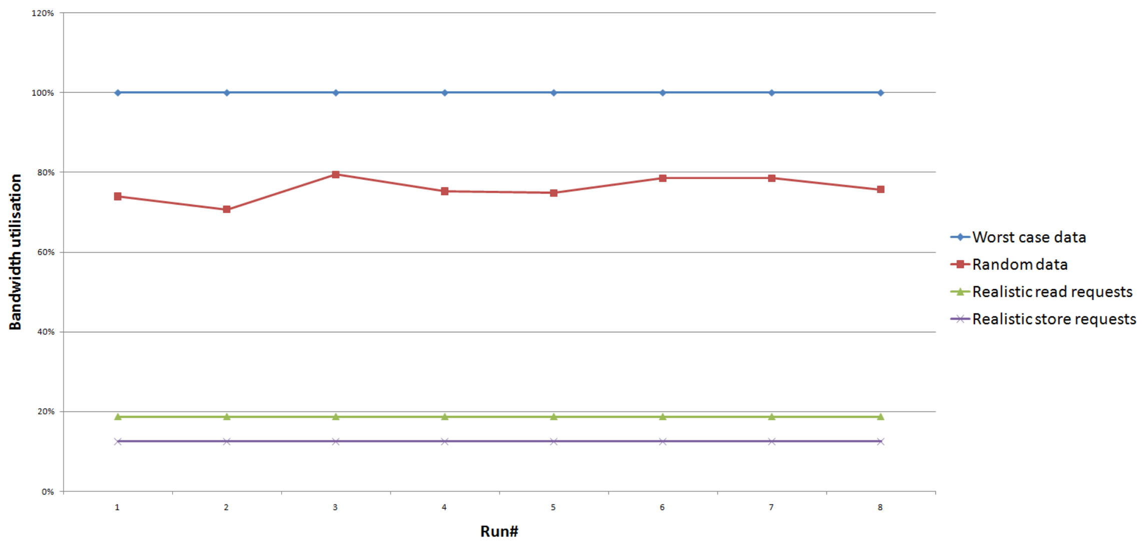


Figure 6.1: Graph showing the peak, random and realistically used bandwidth of the request side of our crossbar

could start, till every queue was empty. First the queues where filled with a random MC distribution, the achieved bandwidth is visible in Figure 6.1. The crossbar reaches about 75% of its possible peak bandwidth with these random datasets. Using stochastic theory to explain this behavior is beyond the scope of this thesis. It does show however that it does pretty well for large streams of random data.

The second part of the test consisted out of filling the queues with worst-case distributed data. What this means can be read in Section 4.4.3.2. Now the crossbar performs at 100% of its peak bandwidth, just as it was designed for. This is visible in Figure 6.1.

Realistic tests

We move on to the more realistic tests. Here we will simulate the behavior of a FU, and see whether the crossbar can handle the amount of data. Instead of filling the queues beforehand, we fill them during the run, at a rate representative for our FU design. First we look at the store requests. They are issued every eight clock cycles by every FU, and in this test we created a total of 64×6 of them. We aligned all the requests in time, so that we ended up with a worst case test. When running, the average number of items in the queues turned out to be zero, and the maximum one. This is easily explained by the fact that we have six requestors, and eight clock cycles to serve. Therefore we will always, even in the worst case (every request pointing to the same MC), be done in time and have no requests pending. This is as expected and as discussed in Section 4.4.3. The bandwidth used by store requests is only 12.5% of the peak, as can be seen in Figure 6.1. Something similar holds for the read requests, only their density is a 1.5 times larger. But, even when using worst case data the maximal queue size is two items (the first serve signal is latched), and the average size zero. The used bandwidth is 18.75% of the possible peak. Remember that the read and store requests are send to a (virtual) different memory port, so their bandwidths are independent. It is clear from Figure 6.1 that our crossbar has more than enough bandwidth to handle the read and store requests from the FUs.

For the memory responses it is a lot harder: harder for us to make performance estimation, and harder for the crossbar to handle, since the responses can come in any order and at any rate. In our design the *memory response queues* are 64 items deep; lets see if that is deep enough. If we consider a worst case scenario, where every FU starts at exactly the same time with its first instruction. Every FU will create almost 200 read request to fill its data buffers in an equal amount of clock cycles. Since we are talking worst case, the responses will enter the crossbar at a continuous rate of eight responses per clock cycle, and the first 8×25 responses will point to the same FU. In this scenario, our crossbar will fail. It will take up to 400 clock cycles before the last item of these first 8×25 responses is served, by then the queue holding it will be long over flown.

For a more realistic picture, we ran several tests to check the memory response behavior. In Figure 6.2 we can see the maximum size of all the memory response queues, for different data arrival rates. To, again, simulate worst case behavior, all the responses are aligned in time. The FU_id in every response is random, and therefore a different test run could give different results. We however believe that, after running several tests, the numbers shown are realistic and representative. Using a stream of 256×8 responses, we can see that the system is stable up to an arrival period of four clock cycles, a smaller period will (eventually) lead to an overflow. We defined three regions of interest: realistic steady state behavior, realistic worst case burst behavior and a queue overflow. On the left we see the steady state behavior; our system will typically require that amount of bandwidth, or less (5.3×(8/6)). In the middle there is a region where the crossbar might end up during a burst of responses, which can only occur at startup time. The upper bound for this region is set by 2×(8/6), or 2.33, the realistic worst case memory response period. On the right we have the region where our crossbar would overflow. Luckily there is a small gap between the worst case burst and overflow region. It shows that the size we choose for the memory response queues is just right: they will probably never overflow. This is consolidated further by the fact that it is, by design, impossible for different FUs to start at the same moment, and that the duration of response bursts will be smaller than in our testbench.

An interesting observation is that queue number three and number seven are the first ones that could eventually overflow, while queues zero and four are fine. This is due to the logic that selects a queue to serve. The logic always starts at the lowest queue, and only if it does not find a response in the first three queues, it checks the fourth. This biased queue serving clearly results in an unbalanced system, improving this might give us better results.

Data compression analyzed

Recall that we compress our sequence and matrix data to a minimum to keep the memory footprint and bandwidth as low as possible. The compression of sequence data can be seen as trivial, there are probably (hopefully) no implementations around using 32 bits to represent a DNA alphabet. To use differential compression for the matrix data however

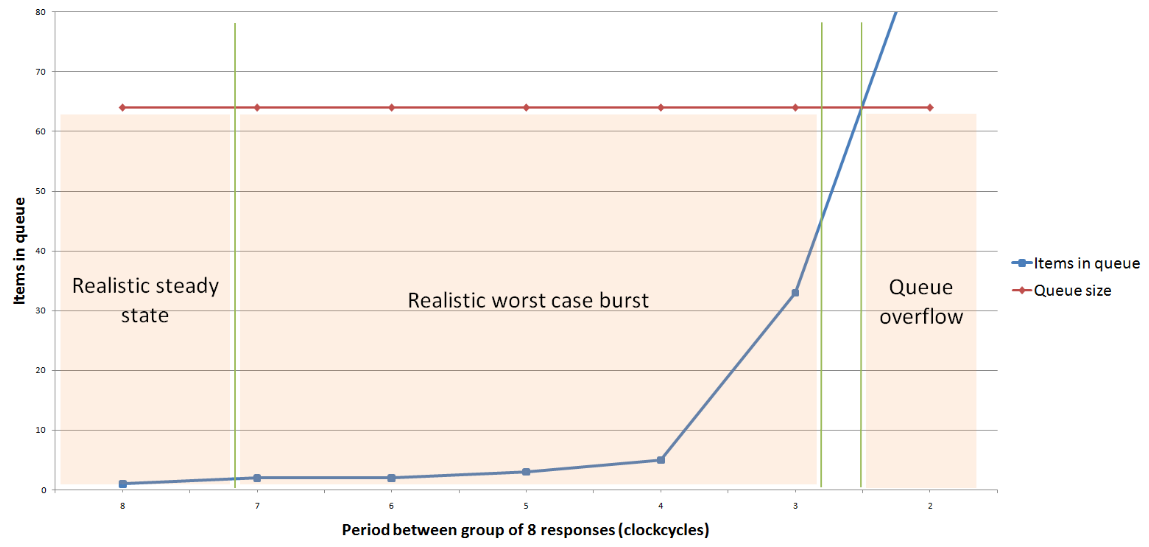


Figure 6.2: Graph showing maximal queue size for different memory response arrival rates

is not so trivial. When we would switch off the compression, the memory response rate will double. The steady state behavior will still be in the safe zone, but during realistic worst case bursts the system could definitely end up in a queue overflow.

On the request side we can see that the amount of store requests would become 2.5 times as large as without using compression. This would however still be okay most of the time, but maybe an address alignment between several FUs could introduce a temporally request stall. The amount of read requests would double in size, just as with the memory responses, and result in a behavior like the store requests.

We can see that compressing the data might not be absolute necessary, but makes life a lot easier. With compression we can say that our system is absolute stable in probably every case, without we could not.

6.1.3 User application & workload distribution

Besides the isolated hardware tests, we also wanted to test our software program (Section 4.5) without the coprocessor. There were several things we would like to verify:

- Cancellation of workload.
- Workload distribution between CPU and CAE.
- Creation of results.

To make it possible to test the program without the coprocessor, we came up with the idea to use a lookup table to deliver the values that would otherwise have been calculated by the CAEs. This lookup table was generated beforehand by a small program and looked like Listing 6.1. Every line represents an instruction with from left to right: the first

32 characters of *sequence A*, the first 32 characters of *sequence B*, the max for clipped sequences, the last column max for clipped sequences and the max of the entire two sequences. This lookup table was used in a separate thread to simulate the coprocessor.

Listing 6.1: Snippet from our result lookup table

```
gcggcaaacccggctcacaccctccacgccgg tagtaaaattaaattaattataaaattatata 246 246 256
gcggcaaacccggctcacaccctccacgccgg atttaaaatataatattaatgtactaaaactt 243 243 264
...
attacaagagcgatgcacactctgaacgacac tagtaaaattaaattaattataaaattatata 333 333 333
attacaagagcgatgcacactctgaacgacac atttaaaatataatattaatgtactaaaactt 328 328 328
...
```

Test data & system
To test the program we used a database and some test queries. As a database we used the latest (13/01/2011) release of the viral1.genomic file from the RefSeq repository [34]. This is a curated database containing a lot of full virus genomes, and several other virus related sequences. There are more files in the viral section, but this one contained plenty of data. It has 3093 sequences, with a total length of 61604948 base pairs. The smallest sequence is 200 base pairs long, the largest 1181548. As queries we randomly selected eight small pieces of sequence from this file, with a total length of 3389 base pairs.

Because our program is multithreaded, and the CPU in the HC-1 only has 2 cores and no hyper threading, we decided to use a different platform to run our isolated test. This way we could see the workings and performance of our program without having the threads being de-scheduled every now and then. The used machine contained an Intel Core i7 920 CPU, which has four cores and the ability to sustain eight threads.

Results
During our tests we aligned all the queries against the database in a single run, all the results were stored in the same packet. This might not represent a sense full test (that would for example be a single query against the database), but it is fine for our case. The metrics we were looking at were the workload distribution between CPU and CAE, the amount of workload that got cancelled, and the final alignments, while varying the amount of alignment results kept in a top score list. In Figure 6.3 we can see the results. We varied the amount of stored results between two and 25, and it is clear that our workload cancellation system works. With only two stored results, the score boundary for entering the top list is such that 11.5% of the matrix cells can be cancelled. In this case, the CPU hardly does any work.

As we move towards a larger results list, we can see that the amount of work done by the CPU converts to 5.7%, and the amount of matrix cells that got cancelled converts to 7.1%. The rest of the workload is done by the CAEs. This can be seen as a steady state result, since we have multiple queries and a large results list. A CPU/CAE workload distribution ratio of roughly 1:20 is a good and in practice useable result, since a modern CPU is just on the edge of delivering this amount of GCUPS compared to FPGA implementations. To make the advantage of our setup even clearer, we also drew a line representing the workload that should have been done if we would pad every *sequence A* to a multiple of 64, instead of clipping it. In that case also small sequences would have

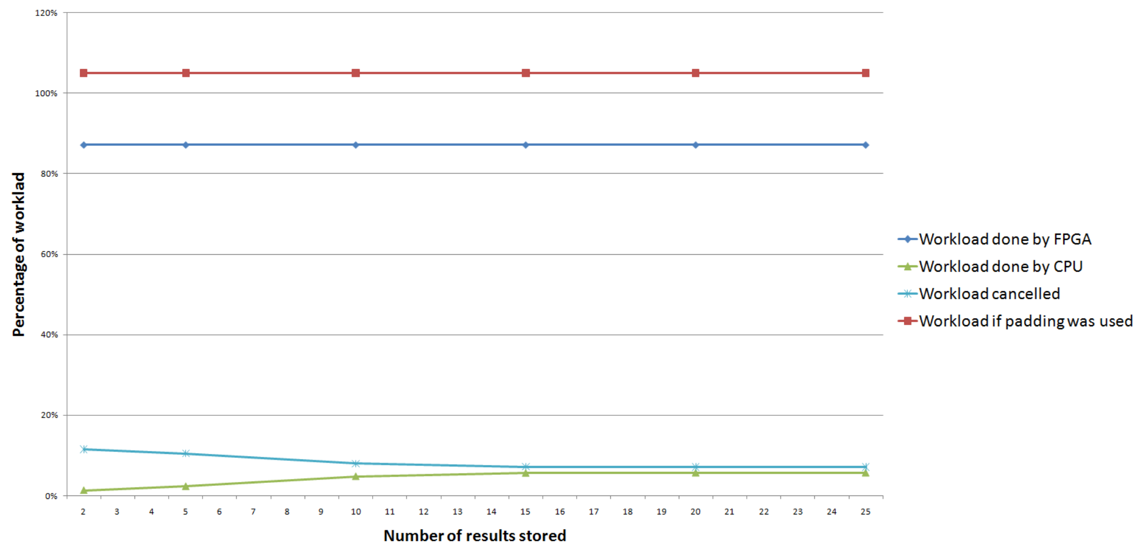


Figure 6.3: Workload distribution and cancellation for our test database and queries

Rank	Query sequence	Database sequence	Score
1	#4	Heliothis virescens ascovirus 3e, complete genome	1,52E-64
2	#1	Heliothis zea virus 1, complete genome	1,33E-46
3	#0	Frog virus 3, complete genome	4,12E-34
4	#0	Soft-shelled turtle iridovirus, complete genome	3,36E-33
5	#3	Zaire ebolavirus, complete genome	6,15E-32

Table 6.2: Alignment results for our test database and queries

been send to the CAE. Our system has to do over 13% less work due to that early stage design decision. The size of the results list can also be seen as a system parameter, you can tune it depending on the CPU power available.

Lastly, we take a look at the actual alignment results. The top 5 alignments are shown in Table 6.2. There are some interesting things to remark. First, the top two alignments clearly stand out over the last three. Looking at our queries we can see that #4 and #1 are indeed the largest queries, so that makes sense, since longer alignments lead to better scores. Furthermore, from the double hit from query #0, we can conclude that the genomes of the 'Frog virus 3' and the 'Soft-shelled turtle iridovirus' share a large almost identical subsequence.

6.2 Implementation results

In this section we will discuss the results and characteristics of our full implementation.

6.2.1 Software S-W performance

As discussed in Section 4.5.2 our software running on the host CPU features a S-W implementation. Currently this is a scalar implementation, since it has been found hard to integrate an SSE2 optimized version in our program. This however results in a S-W version that is an order of magnitude slower than popular implementations. We achieve up to 0.15 GCUPS per CPU core using our approach, against 4 GCUPS for solutions presented in Section 3.2.2. Since our implementation does use the workload cancellation paradigm, the practical speed can be much higher, depending on the dataset used.

6.2.2 System performance

The performance of a piece of hardware is easy to calculate, since it is always the same. For our implementation, the peak performance in GCUPS for the CAEs is the amount of FUs, times the amount of PEs per FU, times the amount of cell updates per PE per clock, times the clock frequency. This comes down to $24\times64\times2\times150\text{M}=460.8$ GCUPS. This is the peak performance of our CAE design, but should, by design, also be the sustainable performance for millions of alignments long. The performance of our total design however can be even higher, since we also use the CPU, and cancel some of the workload. To test this all, we uploaded our personality to the coprocessor and run our application. Sadly enough, we ran into some issues, discussed below.

Inperfect results

After running some tests and analyzing the results, it became clear that only the first alignment of every FU is correct, the following returned values are garbage. We assume this problem is caused by a fault in the *data B* differential en- or decoder. When starting a new alignment, these blocks must reset their internal registers on the right moment, so that we do not subtract the last value of the first matrix from the first value of the second matrix. We assume we are one clock cycle early or late in this process. This could probably have been fixed relatively easy, but since there were more reasons why the full system did not work, we did not try this.

Slow CPU

A second and more serious problem was the speed of the CPU. Our program relies on the CPU to finish instructions and to feed new instructions to the CAEs using multiple threads. Since the CPU used in the HC-1 can only sustain two threads at the same time, at certain moments, the *instruction management thread* gets de-scheduled. Unfortunately, it showed that this off-time was too long to keep the CAEs busy: their queues ran out of instructions before the thread was back online. This problem could be solved by using larger queues, or preferably, by using a faster CPU with more cores.

We tried to improve this by implementing our design with an instruction queue three times as large. This has proven to be impossible, the design would not place and route anymore with the right timings. This is due to the large size of every instruction, 384 bits, and the inability to place that amount of memory in a small place.

6.2.3 System simulation results

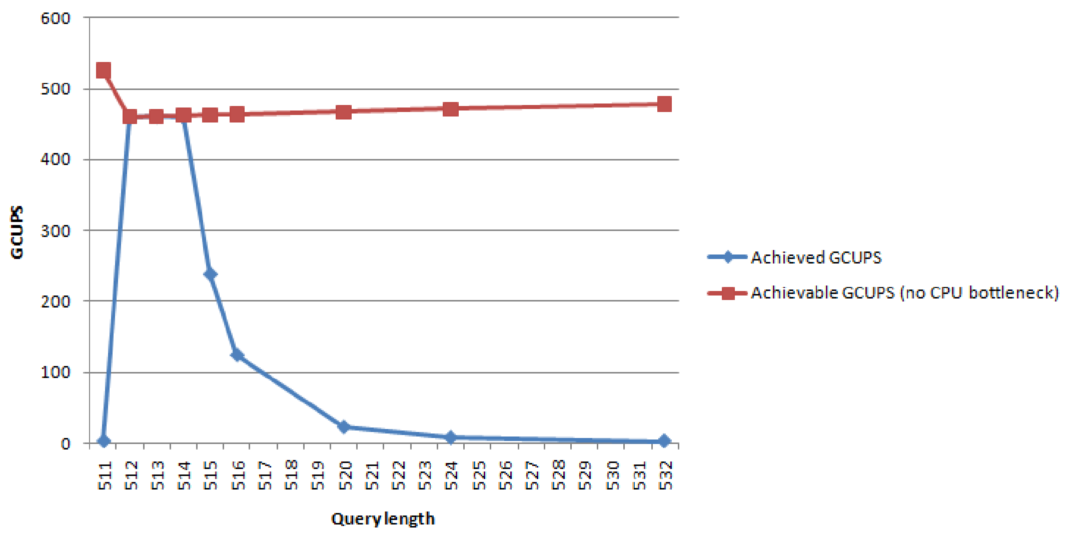
Due to the inability to run benchmarks on a real system, we continued with the software simulator described in Section 6.1.3. To make it realistic, we added a timing feature: the instructions where only set as being done after the amount of time it would take in reality on the CAEs.

Test data & system
For the simulations we did not use a existing database, but created one of our own. This allowed us to control the workload and the sequence length distribution. Since it is impossible in software to simulate the 100% FU utilization behavior, we used very long sequences ranging from 100K to 1M base pairs, thereby eliminating any overhead cost between separate instructions. The total database size was around 300M base pairs. As queries we again randomly selected eight pieces of sequence from the database, varying in length.

As a test system we used the same Core i7 as described in Section 6.1.3. The top scores list has two slots.

Results
In Figure 6.4 we can see the performance results of our system. It shows the system performance in GCUPS versus various query lengths. We can see that for queries with a length which is a multiple of 64, our system performs at its peak performance of 460 GCUPS. If the queries become larger, our software post processing and cancellation system must kick in. For sequences with two or three extra base pairs this works fine, we are basically doing cell updates for free. If the queries become larger, the behavior changes quite a bit. Instead of doing more and more cell updates for free (and thereby having an increasing practical performance), the performance drops dramatically. It is clear that our software post processing step cannot keep up with the performance of the CAEs. This is as expected: in Section 6.2.1 we explained that our scalar S-W implementation is a factor 20 times slower than it could (and as it shows should) be. Note that, the slower the CPU, the longer it will take before certain high scores enter the top list. This results on the same hand in less instructions being cancelled. So the CPU kind of ruins the party for itself: by being so slow, it also has to do extra work. The difference between the two showed graphs can be explained by the fact that they use different datasets. The behavior we are checking heavily depends on the data used.

The same figure also shows the practical GCUPS when we would not have a CPU bottleneck. Here we can see that the performance increases from queries with a length modulo 64 = 0 to queries with a length modulo 64 = 63, as intended. A notable aspect is the difference in peak performance in the two graphs shown. This is due to the relative behavior in workload cancellation. For small sequences a relative larger portion may be cancelled, thereby increasing the practical throughput.



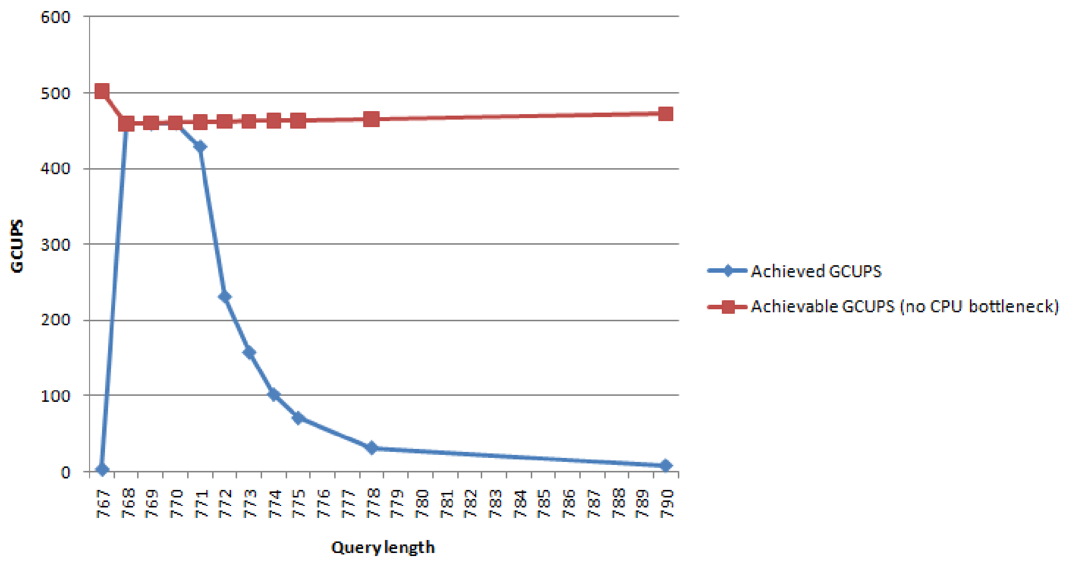


Figure 6.4: System performance for various query lengths

6.3 Comparison with other work

In this section we will compare our work with existing S-W implementations. Since it is impossible to compare our implementation directly with ones on different platforms, or even with ones on another FPGA platform, we divided the comparison in several layers. First, we will look at the behavior of S-W on three hardware platforms regarding five metrics. Hereafter we will discuss the functionality of our implementation versus some other FPGA based implementations, followed by a comparison between the Convey S-W personality and our own.

The reason why we cannot compare different implementations is the difference in platform and implementation details. Aspects like the alphabet used, the gap model, the type of database and of course the hardware itself, mean such a great deal that almost any comparison would be unfair.

6.3.1 Comparison between several hardware platforms

In Chapter 3 we discussed three different platforms and their potential to accelerate S-W. With the knowledge developed during this work and our own implementation weighing in, we would like to take another look at the status quo. The platforms are of course the CPU, the GPU and FPGAs. For all the platforms we regarded a single, but maximal equipped system. For the CPU this means a four way, 48 core Opteron machine. For GPUs a fast PC with 4 high end graphics cards, and for FPGAs the fastest S-W system known, the one from SciEngines [35].

The results can be seen in Figure 6.5. We will discuss them per metric:

- Performance / Euro. FPGAs can deliver the best amount of GCUPS per Euro, followed closely by GPUs. The gap between GPUs and CPUs can be explained by the extra money you have to pay for a 4 way CPU system, while plugging 4 GPUs on a commodity motherboard is free. This result is as expected, and the reason why FPGAs are used for HPC. S-W might not be the algorithm of choice to show a major performance per Euro gain from using FPGAs. Other algorithms, like symmetric ciphers, are a better candidate for that [35].

- Performance / Watt. This is another important reason for using FPGAs. It is clear that here, in contrast to the previous metric, FPGAs are the absolute winner. Full systems can deliver thousands of GCUPS for around a 1000 Watts. Note that, while not visible in the graphs, CPUs score around twice as good as GPUs.

- Flexibility. It is here where FPGAs bite the dust. To get numbers for this metric, we did a little thought experiment. We tried to imagine how many hours it would take to change a fast linear gap implementation to use affine gaps. A skilled engineer will probably manage to do this in a day for a CPU implementation, in a couple of days for GPU implementations, and many weeks for their FPGA counterpart. By talking to Convey, it indeed showed to be a lot of work to change their DNA/linear gap S-W personality to a more complex one.

- Scalability. Here we took CPUs as baseline. Given a suitable problem they are very scalable, there is an entire industry build around their interconnects. GPUs can take advantage of those, but will introduce some extra latency. Therefore they score a bit lower than CPUs. FPGAs have no default implementation platform, if you however take the effort to build a specific hardware framework, they scale very well.

- Future prospect. There are interesting thinks going on in the HPC world. Customers can choose between traditional many core CPUs, GPU HPC solutions [28] or upcoming FPGA products. We believe that GPUs as we know them now (or as we knew them a couple of years ago) will disappear in the future, to be replaced by architectures like Knights Ferry [17, 40] and Fusion [4]. CPUs are therefore

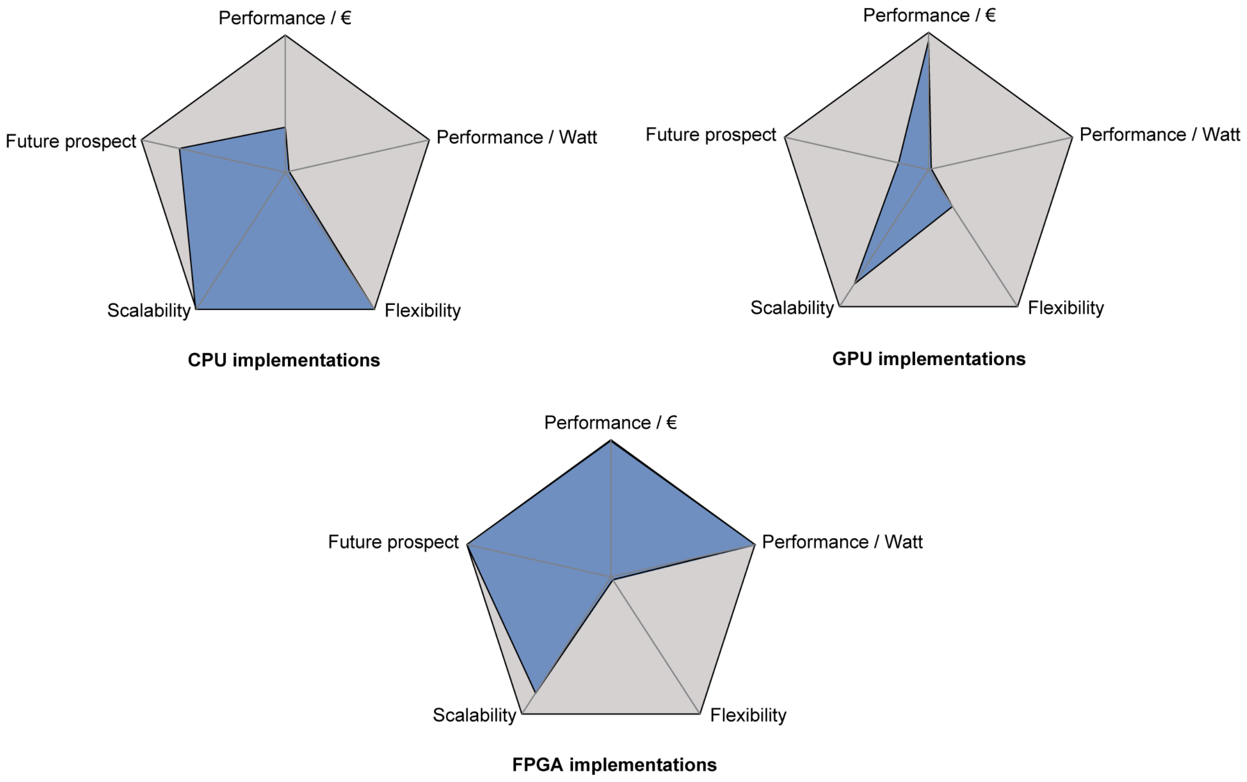


Figure 6.5: Five S-W metrics analyzed for different platforms

scoring pretty good on this metric. In the very specific HPC areas however, where memory bandwidth requirements are low and the problem is very composable, FPGAs will likely continue to be the best choice. S-W is on the edge of being such an algorithm, as we will discuss later in the conclusions and recommendations.

6.3.2 Comparison of functionality between FPGA implementations

Some of the features provided by our implementation are unique. Most of our features are targeting at the goal of using the platform in a most efficient way. We designed every component so that it can deliver a sustainable peak performance for a long period of time, unseen in current implementations. In Table 6.3 we can see five S-W FPGA based implementations and their behavior regarding three metrics, two targeting flexibility and the sustainability of performance and one regarding peak performance. It is clear that our proposed solution scores best the first two metrics: it is the most flexible and efficient implementation known. It is however not optimized for a peak performance, while others are.

6.3.3 Comparison with the Convey S-W personality

The difference in features between Conveys S-W approach and ours is discussed in the previous section. Here we will take a look at the raw performance of both. Since they

	Align sequences of any length	**Align any number of sequences seamlessly**	**Optimized for peak performance**
Propossed	Yes	Yes	No
[8] (Convey)	No, up to $128 \times n$ base pairs	No	Yes
[21] (Lloyd)	Yes, 1 cycle overhead per LSA stride	No	No
[2] (Altera)	Yes, large overhead cost (10-15%)	No	Yes
[29] (Oliver)	Yes, some overhead	No	Yes

Table 6.3: Functionality of our and several other S-W implementations

run at the same platform, this is the only fair comparison possible. In Table 6.4 the results can be seen. It is clear that the lack of features in the Convey solution, is paid back in performance (fourth row): it is clearly faster than our attempt. Which one is best obviously depends on your wish list.

As discussed in Section 4.4.6 we do not fully utilize the FPGA. First, there is room for another FU and second, the design can run at higher frequencies. Both opportunities to increase the performance where not taken, in both cases because they were out of our league regarding time and knowledge. Fitting another FU would require extensive knowledge of floor planning and letting the FU run at a higher frequency would require creating a separate clock domain and would bring a lot of synchronization problems along. If we would have had the opportunity to implement both optimizations, the performance would be around $4 \times 7 \times 128 \times 180$M=645.1 GCUPS, still slower than the version from Convey. This performance is showed at the bottom of the table. It is clear that the features offered by our implementation lay some pressure on the maximum performance attainable. In the middle of the table we can see the amount of resources that are relatively spent on control logic per FU. If we for example would have chosen not to support sequences of any length, we would for instance not need the differential en- and decoding, saving major area (*data B*, the temporary matrix data, would not exist). It might then be possible to support eight FUs instead of seven.

It is important to realize, that our attempt to create a optimal PE was partly aimed at creating one that could run at high frequencies. By not utilizing that attempt, we end up with a performance per area ratio that might be worse than a default 1×1 PE. There is another note that can be placed on the 180 MHz mentioned. Our PE can run at frequencies up to 30 MHz higher, so we lose a lot by interfacing it in our FU. A more extensive knowledge of writing high performance Verilog, and a better understanding of synthesizing software, might contribute to an even higher maximum frequency, and thereby an even higher performance.

	Proposed	**Convey**
FUs	6	18
CUs/FU/clock	128	64
Frequency (MHz)	150	150
Performance (GCUPS)	460,8	691,2
% LUTs used / FPGA	71,0%	unkown
% LUTs control logic / FU	20,1%	unkown
Synthesis frequency (MHz)	186	unkown
Achievable FUs	7	
Achievable frequency (MHz)	180	
Achievable performance (GCUPS)	645.1	691.2

Table 6.4: Comparison between Conveys and our S-W personality

7 Conclusions and recommendations

In this chapter we will say some final words on our work, and give recommendations for further research.

7.1 Conclusions

Here we will name several conclusions based on this thesis.

7.1.1 Evaluation of our implementation

We presented a novel approach to look at S-W acceleration methods. Instead of designing an isolated high speed module, which gets its work from separate dispatches, we presented a system design based on a sustainable peak performance. Furthermore we introduced a work distribution system to utilize the FPGAs and CPU in a most efficient way. Lastly, unnecessary cell updates are being cancelled to reduce the workload.

We did not succeed in running a full system test due to several issues. Therefore we cannot say that every aspect we introduced will work out as planned. However, from the isolated and simulated tests we can conclude several things:

- The functional unit works as planned. We designed a S-W systolic array which can run at 100% utilization for a long period of time.
- The memory crossbar works as planned, it can easily deliver the required bandwidth.
- We showed it is possible to make a PE that performance better per area than a naive implementation. We furthermore showed that it is possible to use this RVE PE is a full system design.
- The workload cancellation paradigm seems to work, we can indeed cancel a noticeable amount of cell updates.
- The workload distribution system seems to work. Aligning small sequences on the CPU makes sense.
- Making the complete system works would require more than just the HC-1 and the code we wrote. Since the HC-1 is very limited in its CPU computing power, it would have to be hooked up to some fast CPU cluster to run the software S-W implementation and post processing steps. The host program as well as the hardware design must be tuned to create a stable and fast environment.

The last bullet point makes an important point very clear. Our work must be seen as a proof of concept for certain ideas, it will never work on this hardware platform in the way presented.

Another important realization is that our efforts to create a system that could run at 100% load might have been in vain. To make this happen, we had to sacrifice logic resources and development time that possibly could have been spend better on a higher peak performance. The amount of time we win by utilizing every clock cycle can be estimated to be around a single percent or less. A simpler but faster design might in the end be a better choice. This points out that it is always important to keep the impact of your design decisions in mind. The workload cancellation paradigm however, especially with the points from the next section in mind, really pays off in execution time. Doing these checks cost little effort and can reduce the workload significantly.

7.1.2 Using RVE for Smith-Waterman

Using RVE PEs in a full system design was one of the reasons this entire research was started. During the research described in Chapter 5 we discovered that the work done on RVE so far, has not resulted in a usable PE: the throughput per area was not better than with a default 1×1 PE. We tried to make a 1×2 PE that could perform better than the default implementation. We succeeded in this, thanks to pipelining, with a small margin. Far later however, we realized that the optimizations done for the pipelined 1×2 PE could also have been applied to the pipelined 1×1 PE. After doing this (in the final stages of the project), we ended up with a pipelined 1×1 PE that performs far better than all the other PEs discussed. This result eliminates every possible use RVE could have had for S-W. Not using RVE furthermore makes the control logic and wiring far less complex, and gives the place and route tool more freedom due to less spread logic.

So, when looking back, we made the wrong choice. We were so focused on getting RVE working, that we turned a little blind for the alternatives and negative side effects of using RVE. Furthermore, we believe that the estimated amount of Virtex 4 slices is not the most important metric. With the insights we have now, we would only have looked at the amount of Virtex 5 LUTs the design would take, since LUTs represented the bottleneck for the final implementation.

We would like to say some final words on RVE. The S-W algorithm, or the class of problems it can represent, is implemented on FPGAs by using a lot of PEs, as discussed thoroughly. The final system performance comes from the frequency, the number of PEs and the number of cell updates per PE per clock. Using a technique that makes one aspect better (number of cell updates per PE per clock), but makes another one (number of PEs) so much worse that the total performance drops, is not an appropriate approach. More general, the whole idea of developing a technique that 'is better when taking area not in account' while targeting HPC on FPGAs for dynamic programming algorithms, means the same as designing a PE 'without taking the frequency in account', or designing an airplane without taking the mass of materials in account.

7.1.3 Making an implementation on the HC-1

The gross of our time was spent on making our implementation on the HC-1 hybrid supercomputer. Here we will discuss some notable aspects of the architecture.

- Memory system. In general, the memory system of the HC-1 is user friendly. The scatter/gather architecture gave the pleasant ability to fetch and store small amounts of data, and the interfaces where easy to use. The latency however was a major drawback for our design. The HC-1 is built around the concept of global memory which is cache coherent throughout the platform. For S-W both of these features are not necessary: it would also be possible to do S-W alignments in distributed memory. In that case, the memory latency could be much better.

- Instruction dispatch. Working with the instruction dispatch was not user friendly at all. The whole concept of having a coprocessor and function dispatches is probably aimed at the general purpose computing Convey likes to see on their system, but is completely unnecessary for simple HPC tasks. It severely interferes with the way the user can communicate with the FPGAs and thereby might limit the usability. We spent to many days on getting the communication between CPU and FPGA right, while the concept of our design does not seem that farfetched at all.

Overall the HC-1 and its framework are nice to work with. We think it has a little overkill on the technical aspects, which might not be necessary for most HPC applications.

7.2 Recommendations for further research

Of course we have several recommendations for the continuation of this research. First we will describe two possible high level optimizations, followed by a re-evaluation of FPGAs as platform for S-W acceleration.

7.2.1 Possible improvements

Before and during the implementation phase we came up with some optimizations that could improve the performance significantly. We did not have the time to implement them, so we would like to mention them here.

- Fast cancellation on the FPGAs. Instead of only looking at the last one up to 63 columns for possible cancellation, it is also possible to implement a cancellation strategy on the FPGA. In parallel with the normal S-W work, a little control block could check whether it pays off to continue the current similarity matrix. If not, the FU could immediately jump to the next alignment. This does not have to cost a large amount of resources so almost any amount of extra cancelled columns would be a worthy result.

- LSAs of different lengths. When using LSAs all 64 matrix columns wide, the average number of columns unfinished is 31. If we would use LSAs of different lengths, we might end up with a better coverage. After doing an automated exhaustive search (Appendix A), it showed that six LSAs of for example 48, 66, 68, 69, 70 and 79 PEs wide would result in an average of seven uncalculated columns. This significantly reduces the workload for the CPU, probably resulting in a more balanced system. On the other hand, it also introduces less opportunity to cancel workload (if only cancellation in software is used). For FPGA implementations that use sequence padding instead of clipping, this optimization can be of even more importance.

Both optimizations mentioned have not been seen in an implementation to date.

7.2.2 Re-evaluation of FPGA usage

In Section 6.3.1 we stated that FPGAs have a good future prospect regarding S-W acceleration. Here we would like to add a little depth and discussion to that statement. While it is proven possible to make a rather simple implementation that has a better performance / Euro ratio than a software implementation, doing so for a more complex case might be difficult. By supporting proteins, affine gaps and larger sequences the performance might drop to a level from where the FPGAs no longer have an easy lead over CPUs. When looking a bit deeper, at the core of the S-W algorithm, we will mainly see additions, an operation where FPGAs do not have a huge advantage over CPUs. If we compare an addition with a logic AND function, they both take one CPU cycle, but the latter introduces a far smaller logic delay on a FPGA, since only a couple of independent gates are required. Algorithms containing a lot of bit operations (hash functions for example) are therefore a better candidate for a huge performance gain by using FPGAs.

Furthermore S-W requires data input for every alignment. For simple implementations only the sequences are required, but when you want to support larger sequences, you have to use some sort of matrix data storage. While the bandwidths are not a problem (especially on the HC-1), it does require buffers, logic, IO etc. Algorithms that generate their own data are therefore much easier to accelerate.

The conclusion is that S-W can be faster per Euro on a FPGA than on a CPU, but only if you limit your problem space. Since the end of FPGA development is not in sight, this behavior will continue into the future.

Bibliography

[1] Marco Aldinucci, Massimiliano Meneghin, and Massimo Torquati, *Efficient s-w on multi-core with fast-flow*, Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing (2010).

[2] Altera, *Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform*, Altera White Paper (2007).

[3] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman, *Basic local alignment search tool*, Journal of Molecular Biology (1990).

[4] AMD, *Amd website*, `http://www.amd.com/`, 2010.

[5] Geoffrey J. Barton, *Protein sequence alignment and database scanning*, Oxford University Press, 1996.

[6] Azzedine Boukerche, Rdolfo Bezerra Batista, and Alba Cristina Magalhaes Alves de Melo, *Exact pairwise alignment of megabase genome biological sequences using a novel z-align parallel strategy*, Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (2009).

[7] Jacques Cohen, *Bioinformatics, an introduction for computer scientists*, ACM Computing Surveys (CSUR) (2004).

[8] Convey, *Convey website*, `http://www.convey.com`, 2010.

[9] Cray, *Cray website*, `http://www.cray.com/`, 2010.

[10] EBI, *Swissprot website*, `http://www.ebi.ac.uk/uniprot/`, 2010.

[11] Michael Farrar, *Striped s-w speeds database searches six times over other simd implementations*, Oxford Journal (2006).

[12] Center for bioinformatics Peking, *Cbi website*, `http://www.cbi.pku.edu.cn/`, 2010.

[13] Mustafa Gok and Caglar Yilmaz, *Efficient cell design for systolic sw implementations*, International Conference on Field Programmable Logic and Applications (2006).

[14] Laiq Hasan and Zaid Al-Ars, *An efficient and high performance linear recursive variable expansion implementation of the smith-waterman algorithm*, Annual International Conference of the IEEE Engineering in Medicine and Biology Society (2009).

[15] Laiq Hasan, Zaid Al-Ars, Zubair Nawaz, and Koen Bertels, *Hardware implementation of the smith-waterman algorithm using recursive variable expansion*, Proceedings of 3rd International Design and Test Workshop IDT08 (2008).

[16] Laiq Hasan, Zaid Al-Ars, and Mottaqiallah Taouil, *High performance and resource efficient biological sequence alignment*, 32nd Annual International Conference of the IEEE EMBS (2010).

[17] Intel, *Intel website*, `http://www.intel.com/`, 2010.

[18] Alexander Isaev, *Introduction to mathematical methods in bioinformatics*, Springerl, 2004.

[19] Server Kasap, Khaled Benkrid, and Ying Liu, *High performance fpga-based core for blast sequence alignment with the two-hit method*, Unkown (2008).

[20] Marijn Kentie, *Biological sequence alignment using graphics processing units*, Master's thesis, TU Delft, 2010.

[21] Scott Lloyd and Quinn O. Snell, *Sequence alignment with traceback on reconfigurable hardware*, International Conference on Reconfigurable Computing and FPGAs (2008).

[22] Hadon Nash, Douglas Blair, and John Grefenstette, *Comparing algorithms for large-scale sequence analysis*, Proceedings of the 2nd IEEE International Symposium on Bioinformatics and Bioengineering (2001).

[23] Zubair Nawaz, *Recursive variable expansion - a transformation for reconfigurable computing*, Ph.D. thesis, TU Delft.

[24] Zubair Nawaz, Koen Bertels, and H. Ekin Sumbul, *Fast smith-waterman hardware implementation*, IEEE International Symposium on Parallel & Distributed Processing (2010).

[25] Zubair Nawaz, Mudassir Shabbir, Zaid Al-Ars, and Koen Bertels, *Acceleration of biological sequence alignment using recursive variable expansion*, ProRISC 2007 (2007).

[26] NCBI, *Ncbi website*, `http://www.ncbi.nlm.nih.gov/BLAST/tutorial/Altschul-1.html`, 2010.

[27] Saul B Needleman and Christian D. Wunsch, *A general method applicable to the search for similarities in the amino acid sequence of two proteins*, Journal of Molecular Biology (1970).

[28] NVIDIA, *Nvidia website*, `http://www.nvidia.com`, 2010.

[29] Tim Oliver, Bertil Schmidt, and Douglas Maskell, *Hyper customized processors for bio-sequence database scanning on fpgas*, ACM/SIGDA 13th international symposium on Field-programmable gate arrays (2005).

[30] ORNL, *Human genome project website*, `http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml`, 2010.

[31] William R. Pearson, *Searching protein sequence libraries: comparison of the sensitivity and selectivity of the smith-waterman and fasta algorithms*, Genomics (1988).

[32] William R. Pearson and David J. Lipman, *Improved tools for biological sequence comparison*, Proc. Natl. Acad. Sci. USA (1988).

[33] Kiran Puttegowda, William Worek, Nicholas Pappas, Anusha Dandapani, and Peter Athanas, *A run-time reconfigurable system for gene-sequence searching*, 16th International Conference on VLSI Design (2003).

[34] NCBI RefSeq, *Refseq site*, `http://www.ncbi.nlm.nih.gov/RefSeq/`, 2010.

[35] SciEngines, *Sciengines website*, `http://www.sciengines.com/`, 2010.

[36] Kal Renganathan Sharma, *Bioinformatics: Sequence alignment and markov models*, McGraw-Hill Professional, 2008.

[37] Temple F. Smith and Michael S. Waterman, *Identification of common molecular subsequences*, Journal of Molecular Biology (1981).

[38] Adam Szalkowski, Christian Ledergerber, Philipp Krhenbhl1, and Christophe Dessimoz, *Swps3 - a fast multi-threaded vectorized smith-waterman for ibm cell/b.e. and 86/sse2*, BioMed Central (2008).

[39] POSIX threads, *Posix threads website*, `https://computing.llnl.gov/tutorials/pthreads/`, 2010.

[40] HPC Wire, *Hpc wire, knights ferry versus fermi*, `http://www.hpcwire.com/features/Compilers-and-More-Knights-Ferry-v-Fermi-100051864.html`, 2010.

[41] Xilinx, *Xilinx website*, `http://www.xilinx.com`, 2010.

Specification of code and support tools

A

Here we will take a look at our Verilog code for the hardware design, the code of the user application and several tools we wrote to support the design and test process.

A.1 Verilog code

In Listing A.1 we can see the tree structure of the Verilog code of our hardware design. The parts written by Convey are not shown. The nested structure of the functional unit, LSA and the PE is clearly visible. We see only one PE, since the complete PE array is created using a generate statement. The multiple FUs are not generated, but instantiated separately and therefore all visible. Our memory crossbar is on the same level as the FUs, and holds all the request and response finders.

The approximate number of lines is shown in the second column. An interesting observation is that the LSA module is pretty small, clearly our generate statement and smart wiring keeps the number of lines low.

Listing A.1: Structure of our Verilog code

```
| structure                                       | lines of code (appr.) |
| ----------------------------------------------- | --------------------- 
| / cae_pers                                      | 1000                  |
|   +-- functional unit 0                         | 950                   |
|   |   +-- linear systolic array                 | 125                   |
|   |       +-- processing element                | 200                   |
|   +-- functional unit ...                       |                       |
|   +-- functional unit 5                         |                       |
|   +-- memory crossbar                           | 1200                  |
|   |   +-- store request finder 0                | 150                   |
|   |   +-- store request finder ...              |                       |
|   |   +-- store request finder 7                |                       |
|   |   +-- read request finder 0                 | 150                   |
|   |   +-- read request finder ...               |                       |
|   |   +-- read request finder 7                 |                       |
|   |   +-- memory response finder 0              | 130                   |
|   |   +-- memory response finder ...            |                       |
|   |   +-- memory response finder 5              |                       |
|   +-- instruction queue                         |                       |
| / constrains.ucf                                | 130                   |
```

A.2 User application code

The layout of our user application code can be seen in Listing A.2. Only the main methods are shown. We can see the user application itself, the S-W library, a S-W CPU library and a list of some global methods.

The first mentioned is the part that the end user will see. This piece of code is pretty small, indicating that the program is easy to use. The user application talks to the S-W library as indicated in Section 4.5. In the listing we can see the three functions necessary to load a database and to create alignment instructions. The library has furthermore a start and stop function. The S-W CPU library contains our CPU S-W implementation. It offers a default (full) implementation, and one that tries to cancel as many columns as possible. The SW_cpu_finish_fpga function decompresses the *data B* received from the CAEs and calls one of the two software implementations to finish the last columns. This library is called from the SW_CPU thread.

Besides the two libraries, we also have a couple of functions that live in the global space. The three most important ones are the ones that run in a separate thread. These are also discussed in Section 4.5. Another method worth mentioning is the one that actually sends an instruction to the coprocessor. It mainly consists out of copying variables to the right position and setting the right flags. The encode_DNA method converts a char array to a unsigned long long array, where every 64 bits contain 32 bases.

Listing A.2: Structure of our C++ code

```
| structure                                   | lines of code (appr.) |
| ------------------------------------------- | --------------------- 
| / User application                          | 50                    |
| / S-W library                               |                       |
|   +-- align_db_against_db                   | 5                     |
|   +-- align_seq_against_db                  | 85                    |
|   +-- load_db                               | 110                   |
|   +-- start_processing                      | 50                    |
|   +-- stop_processing                       | 5                     |
| / S-W CPU library                           |                       |
|   +-- SW_cpu_finish_fpga                    | 40                    |
|   +-- SW_cpu_full                           | 45                    |
|   +-- SW_cpu_with_cancel                    | 60                    |
| / Global methods                            |                       |
|   +-- manage_instructions (thread)          | 75                    |
|   +-- manage_workloads (thread)             | 140                   |
|   +-- SW_cpu (thread)                       | 55                    |
|   +-- send_instr_to_fpga                    | 70                    |
|   +-- encode_DNA                            | 45                    |
|                                             |                       |
| / Coprocessor code                          |                       |
|   +-- coprocessor function                  | 110                   |
|                                             |                       |
| / All code                                  | 1410                  |
```

A.3 Support tools

During the design and test process we created several small tools to help us. Here we will name and discuss them briefly.

Creating test databases

To not be dependent on existing DNA databases, we wrote a tool that could create them. Some parameters are the total size, the minimal length and maximal length of the sequences. Furthermore the user can specify how many of the sequences must be smaller than a certain threshold. This way the resulting database could contain for example more small sequences than large ones.

It is also possible to automatically generate some queries from the generated database.

Creating lookup tables

To create the lookup tables used to simulate the CAEs, we wrote a little tool. This is actually not much more than a software S-W implementation, a bit of control code and some in- and output code.

Crossbar collisions

To get the numbers in Table 4.2 we wrote a Monte Carlo simulation. The tool generates arrays of random numbers and interprets them as memory requests. By doing many runs, we end up with the probabilities shown in the table. It would also have been possible to get these numbers from theory, but would probably have taken us far more time.

Creating crossbar and functional unit testbenches

To test our crossbar and functional unit we used large Verilog testbenches.

For the crossbar this meant a number of assignments to queues or registers. Since the tests required code in the order of hundreds of assignments, these statements where automatically generated with a tool. Its output is a piece of code, with a correct syntax, that could be copy pasted in the testbench framework.

For the FU testbenches the tool generated Verilog statements that filled the simulator memory with sequence data. When testing multiple large sequences, this could also be in the hundreds of lines of Verilog code.

Linear systolic arrays of different lengths

To get the different LSA lengths mentioned in Section 7.2.1 we wrote a small program. It does an exhaustive search for the optimal LSA lengths with the following limitations:

- The total amount of PEs should be between 6×64 (384) and $6\times64 + 16$ (400).
- The minimal length of a LSA is 64-16 (48), the maximum length is 64+16 (80).
- The sequence contains between 64 and 4096 base pairs.

We searched for the LSA lengths where, first, the modulo is minimal, and second, the number of PEs is a large as possible. There are many solutions that give the same best outcome, one example is: 48, 66, 68, 69, 70 and 79. In this case the average modulo is seven and the total number of PEs is 400.

B Automated RVE hardware design tool

To create RVE PEs of various sizes we wrote a tool. It creates and optimizes all the equations for a certain RVE dimension, and outputs a Verilog description of the PE.

B.1 Creating all the equations

RVE is invented as a loop transform. By recursively expanding the loop variables, one can eliminate all data dependencies. In our case we want to rewrite the S-W definition so that we can calculate the value of any matrix cell at ones. To do so we did not look at RVE as a loop transform, but as a tree problem. The equations involved in S-W then become paths from a root (the cell we want to calculate) to a known cell. An example of this is shown in Figure B.1 for a 3x3 matrix. For the sake of clarity, only a small portion of the tree is drawn. We want to calculate the cell with the X, so we spawn three children (the data dependencies). Since the value of these children is unknown, every child spawns again three children. This process is continued until every leaf node is a known value, or as in the example, a letter. Every node to node connection represents a gap or a match score. As said, an equation is now represented by a path from the root to a known cell, and the number of equations is therefore equal to the amount of leaf nodes.

This is implemented with a class structure, where every class creates three children, until it has reached the necessary depth. Every node has a list of matches and gaps which represent the equation until then. When creating new children, the parent passes this list including a new match to one of them, and including an extra gap to the other two children. Since a lot of equations use the same matches, these matches are stored in a separate structure. This can be seen as a matrix where every cell holds the score for the transition from that cell to its lower-left neighbor. In the next section such a structure is called a score matrix.

When the entire tree is created, we traverse it depth first. At arrival at a leaf node, we write the corresponding equation to a useable *Equation* format, and store it in a list.

B.2 Optimizing the equations

Since a lot of equations are not needed for the correct calculation of our unknown cell, we can cancel them. In our program this is mainly done in a brute force way. First, we create every possible match matrix. That way we have every possible combination between matches and mismatches that can happen for our RVE dimension. For every matrix we calculate the score of every equation and keep track of the best score. Before we switch to the next matrix, we check which equations are the ones that have lead

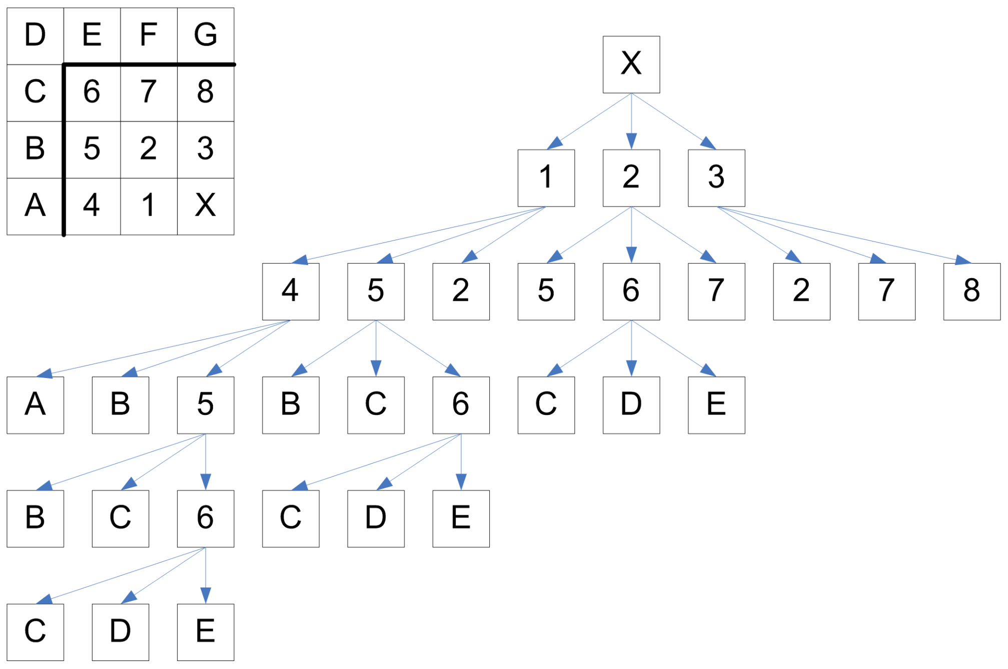


Figure B.1: RVE as a tree problem

to the maximal score found, and give them a flag. After we have processed all the match matrices, we check for equations without a flag. Clearly they have never been the equation that leads to the maximal score, and can therefore be discarded.

This method works for elimination almost every unnecessary equation. There are however some cases in which this system fails. If we look at the two cases shown in Figure B.2, our described method will not work. If we use a score system with +2 for a match, -1 for a mismatch and -2 for a gap, the left path (or equation) will give us a best case score of -2. The right path will give us a worst case score of -2. Our brute force system will keep both, since they both reach the maximum in their own best case match matrix. Analyzing these two equations together will however lead to the conclusion that you need only the right one, since it is always at least as good as the other one.

To deal with these difficult cases we wrote a piece of code that detects ‘mismatch’ versus ‘gap - gap’ patterns, and cancels the latter equation. This is also done for ‘match - match’ versus ‘gap - match - gap’ patterns, since the same analysis holds here. For larger matrices and more complex paths such an analysis becomes very non-trivial, and the scoring scheme used becomes a contributing factor. However, during extensive random testing of our final equations, it seems like our cancellation algorithm is doing a good job.

B.3 Dealing with the clipping error

The clipping error as described in [23] makes life a bit more complicated. Figure B.3 illustrates the problem. In the path shown, it might be possible that the score of the

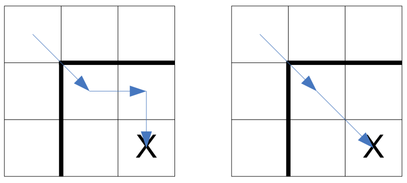


Figure B.2: Difficult case for our automated equation cancellation

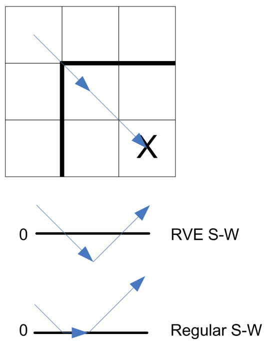


Figure B.3: Clipping error when using RVE

first intermediate step drops below zero. Following the S-W definition, it is clipped at zero, as visible in the bottom subfigure. Using RVE however, this intermediate clipping step no longer exists. The result is that, in this case, the final result is too low, as can be seen in the middle subfigure. When this clipping error can occur, and how it can be treated can be read in detail in Nawaz' work [23]. Therefore we will not repeat it here.

To cope with the error we check every equation for the possibility of clipping. If the equation falls in the definition, we give it a flag. This flag is used later on when we calculated scores (as discussed in the previous section), and when we generate hardware in the final step of our program. These equations are hardware is not written as in the example in Listing 5.2. Instead, all the terms are added in the right order, and clipped at zero before going to the next addition.

When testing RVE sizes of 5×5 and larger, we encountered an error in the clipping handling. Whether this comes from a bad implementation, or a faulty clipping error handling definition cannot be said. But as explained before, analyzing long paths through large matrices becomes very non-trivial; therefore we think that the definition is not universal enough and only works for small matrices. Since we would not use such large PEs anyway, we did not bother.

B.4 Generating HDL

When all the equations are ready, it is time to generate Verilog code. An example of this can be seen in Listing 5.2. The lion's share of the code is pretty straightforward. To create the final *find maximum* tree code however we did some extra work. Since the XST compiler does a poor job at high level code optimization, it cannot create a minimum depth tree from a given number of *max* operations. Therefore, we wrote a small algorithm that does that automatically. Another small optimization that we do is group the match and gap scores together, so they do not have to be recalculated for every equation.

We looked at the possibility of eliminating common subexpression ourselves. Which terms could be grouped together for a maximal performance and area gain showed to be a difficult task. Even for simple cases, the XST compiler had a different opinion about what is fast and what is not than we had. This led us to do no common subexpression elimination at all.