A Highly Parameterised and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment

Khaled Benkrid, *Senior IEEE Member*, Ying Liu, *IEEE Student Member*, AbdSamad Benkrid, *IEEE Member*
The University of Edinburgh, School of Engineering and Electronics, the King's Buildings, Mayfield Road, Edinburgh, EH9 3JL, Scotland, UK
k.benkrid@ieee.org

Abstract

This paper presents the design and implementation of the most parameterisable FPGA-based skeleton for pairwise biological sequence alignment reported in the literature. The skeleton is parameterised in terms of the *sequence symbol type* i.e. DNA, RNA, or Protein sequences, the *sequence lengths*, the *match score* i.e. the score attributed to a symbol match, mismatch or gap, and the *matching task* i.e. the algorithm used to match sequences, which includes global alignment, local alignment and overlapped matching. Instances of the skeleton implement the Smith-Waterman and the Needleman-Wunsch algorithms. The skeleton has the advantage of being captured in the Handel-C language, which makes it FPGA platform-independent. Hence, the same code could be ported across a variety of FPGA families. It implements the sequence alignment algorithm in hand using a pipeline of basic processing elements, which are tailored to the algorithm parameters. The paper presents a number of optimisations built into the skeleton and applied at compile-time depending on the user-supplied parameters. These result in high performance FPGA implementations tailored to the algorithm in hand. For instance, actual hardware implementations of the Smith-Waterman algorithm for Protein sequence alignment achieve speed-ups of two orders of magnitude compared to equivalent standard desktop software implementations.

1. Introduction

Biological sequence alignment is a widely used operation in the field Bioinformatics and Computational Biology (BCB). It aims to find out whether two or more biological sequences (e.g. DNA, RNA or Protein sequences) are related or not [1]. This has many important real world applications. For instance, if some information about one of the sequences is already known (e.g. the sequence represents a cancerous gene) then this information could be transferred to the other unknown sequences, which could be vital in early disease diagnosis and drug engineering. Other applications include the study of evolutionary development and the history of species and their groupings [1][2]. However, biological sequence alignment is also a computationally intensive operation and with biosequence databases growing exponentially every year, there is indeed a need for an equally faster computing technology to keep up with this growth. Desktop computer systems are not capable to achieve this task within realistic execution times. As a result, heuristics-based sequence alignment algorithms which are more suitable to software implementation have been designed to reduce their execution time on such systems [3][4][5]. In this paper, we concentrate on the exhaustive search algorithms which provide gold standards against which less accurate algorithms are benchmarked against. For exhaustive search algorithms, many Single Instruction Multiple Data (SIMD) and systolic architectures using special purpose hardware have been built to speed up their execution, often with one order of magnitude or more speed-up compared to a workstation based implementation [6][7][8]. More recently, reconfigurable hardware in the form of Field Programmable Gate Arrays (FPGAs) has been used as a high performance programmable platform for sequence alignment algorithms [9][10][11][12][13][14][15]. Nonetheless, the number of researchers working on FPGA-based accelerators for sequence alignment and BCB applications in general, remains low. This is due mainly to the relative newness of the two areas, but more importantly, perhaps, to the knowledge gap between bioinformaticians and molecular biologists on the one side and hardware design engineers on the other side [16][10].

In an attempt to make a contribution towards bridging the aforementioned gap, this paper presents the detailed design and implementation of a generic and highly parameterised FPGA skeleton for pairwise biological sequence alignment. Compared to previously published FPGA implementations, our solution provides the most parameterised one, hence allowing users to tune in FPGA hardware to suit their particular need. It has also the added advantage of being designed in an FPGA-platform-independent language, namely Handel-C, which makes it possible to target a variety of FPGA families and architectures e.g. Xilinx, Altera, and Actel, or even standard ASICs.

The remainder of this paper is organised as follows. First, important background information on biological sequence alignment algorithms is presented. Then, previous work in the area of high performance biological sequence alignment is presented. After that, the design and FPGA implementation of our highly parameterised skeleton is detailed. A comparative evaluation of our implementation then follows, before conclusions and plans for future work are laid out.

2. Background

Biological sequences (e.g. DNA) evolve through a process of mutation, selection, and random genetic drift [17]. Mutation, in particular, manifests itself through three main processes, namely: *substitution* of residues (i.e. a residue A in the sequence is substituted by another residue B), *insertion* of new residues, and *deletion* of existing residues. Insertion and deletion are referred to as *gaps*. This should be taken into account when aligning biological sequences. In practice, the degree of alignment of sequences is measured by a score, which is obtained by a summation of terms of each aligned pair of residue in addition to possible gap terms. Score terms for each aligned residue terms are obtained from probabilistic models and stored in *score* or *substitution matrices* [1]. Figure 1 presents an example of such substitution matrices, namely the BLOSUM50 which is a 20x20 matrix for Protein residues (also known as amino-acids).

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B
A	5	-2	-1	-2	-1	-1	-1	0	-2	-1	-2	-1	-1	-3	-1	1	0	-3	-2	0	-2
R	-2	7	-1	-2	-4	1	0	-3	0	-4	-3	3	-2	-3	-3	-1	-1	-3	-1	-3	-1
N	-1	-1	7	2	-2	0	0	0	1	-3	-4	0	-2	-4	-2	1	0	-4	-2	-3	4
D	-2	-2	2	8	-4	0	2	-1	-1	-4	-4	-1	-4	-5	-1	0	-1	-5	-3	-4	5
C	-1	-4	-2	-4	13	-3	-3	-3	-3	-2	-2	-3	-2	-2	-4	-1	-1	-5	-3	-1	-3
Q	-1	1	0	0	-3	7	2	-2	1	-3	-2	2	0	-4	-1	0	-1	-1	-1	-3	0
E	-1	0	0	2	-3	2	6	-3	0	-4	-3	1	-2	-3	-1	-1	-1	-3	-2	-3	1
G	0	-3	0	-1	-3	-2	-3	8	-2	-4	-4	-2	-3	-4	-2	0	-2	-3	-3	-4	-1
H	-2	0	1	-1	-3	1	0	-2	10	-4	-3	0	-1	-1	-2	-1	-2	-3	2	-4	0
I	-1	-4	-3	-4	-2	-3	-4	-4	-4	5	2	-3	2	0	-3	-3	-1	-3	-1	4	-4
L	-2	-3	-4	-4	-2	-2	-3	-4	-3	2	5	-3	3	1	-4	-3	-1	-2	-1	1	-4
K	-1	3	0	-1	-3	2	1	-2	0	-3	-3	6	-2	-4	-1	0	-1	-3	-2	-3	0
M	-1	-2	-2	-4	-2	0	-2	-3	-1	2	3	-2	7	0	-3	-2	-1	-1	0	1	-3
F	-3	-3	-4	-5	-2	-4	-3	-4	-1	0	1	-4	0	8	-4	-3	-2	1	4	-1	-4
P	-1	-3	-2	-1	-4	-1	-1	-2	-2	-3	-4	-1	-3	-4	10	-1	-1	-4	-3	-3	-2
S	1	-1	1	0	-1	0	-1	0	-1	-3	-3	0	-2	-3	-1	5	2	-4	-2	-2	0
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	2	5	-3	-2	0	0
W	-3	-3	-4	-5	-5	-1	-3	-3	-3	-3	-2	-3	-1	1	-4	-4	-3	15	2	-3	-5
Y	-2	-1	-2	-3	-3	-1	-2	-3	2	-1	-1	-2	0	4	-3	-2	-2	2	8	-1	-3
V	0	-3	-3	-4	-1	-3	-3	-4	-4	4	1	-3	1	-1	-3	-2	0	-3	-1	5	-4

Figure 1. The Blosum50 substitution matrix

Gap penalties depend on the length of the gap and are generally assumed to be independent of the gap residues. We distinguish between two main types of gap penalties: *linear* and *affine*. In the former, the cost of a gap of length g is given by a linear function:

$$Penalty(g) = -g*d$$

In the case of affine gap penalties, however, a constant penalty is given to opening a new gap, while a linear, and often smaller, penalty is given to subsequent gap extensions:

$$Penalty(g) = -d-(g-1)*e$$

This is a more realistic model as it is often the case that gaps of few residues are as frequent as gaps of a single residue. The following discusses two alignment algorithms, namely the Needleman-Wunsch global alignment algorithm and the Smith-Waterman local alignment algorithm. For the sake of simplicity, we will first assume a linear gap model. The case of affine gap models will be discussed subsequently in Section 2.4.

2.1 Global alignment: the Needleman-Wunsch Algorithm

The Needleman-Wunsch algorithm is a dynamic programming algorithm which finds the optimal global alignment between two sequences $X = x_1x_{2..}x_{i..}x_M$ (of length M), and $Y = y_1y_{2..}y_{i..}y_N$ (of length N) [18]. The algorithm builds a matrix F of scores, where each element $F(i,j)$ is the best alignment between segment $x_1x_{2..}x_i$ of X and $y_1y_{2..}y_j$ of Y. Matrix F is built recursively using the following equation:

$$F(i,j) = \max\begin{cases} F(i-1, j-1) + s(x_{i,}y_j), \\ F(i-1, j) - d, \\ F(i, j-1) - d. \end{cases} \qquad (1)$$

This means that given the best alignment between: $x_1x_{2..}x_{i-1}$ and $y_1y_{2..}y_{j-1}$ (i.e. $F(i-1, j-1)$), $x_1x_{2..}x_{i-1}$ and $y_1y_{2..}y_j$ (i.e. $F(i-1, j)$), and $x_1x_{2..}x_i$ and $y_1y_{2..}y_{j-1}$ (i.e. $F(i, j-1)$), the best alignment between $x_1x_{2..}x_i$ and $y_1y_{2..}y_j$ is the largest score of three alternatives:

- An alignment between x_i and y_j, in which case the new score is $F(i-1,j-1)+s(x_i,y_j)$ where $s(x_i,y_j)$ is the substitution matrix score or entry for residues x_i and y_j.
- An alignment between x_i and a gap in Y, in which case the new score is $F(i-1,j)-d$, where d is the gap penalty.
- An alignment between y_j and a gap in X, in which case the new score is $F(i,j-1)-d$, where d is the gap penalty.

This is illustrated graphically in the following figure.

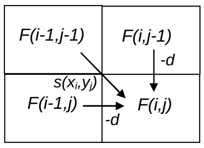
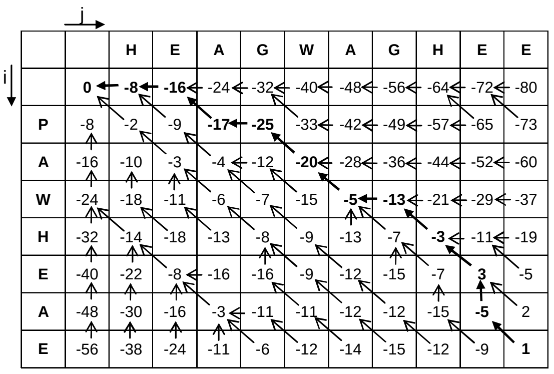


Figure 2. Dynamic Programming illustration

It is important to consider the case of boundary conditions. Indeed, in order to compute the matrix F values from (1), we need to define values for $F(0,0)$, $F(i,0)$ for $i=1,2,…M$, and $F(0,j)$ for $j=1,2,…,N$. $F(0,0)$ does not represent any alignment per se, and is hence the initial value from which $F(i,j)$ should be accumulated. It is thus always set to 0. $F(i,0)$ and $F(0,j)$ however represent alignment of a prefix of X (Y respectively) to all gaps in Y (X respectively). As a result: $F(i,0)=-i*d$ and $F(0,j)=-j*d$.

Following the above procedure, the best global alignment between X and Y is hence obtained by tracing back from the final cell in the matrix i.e. $F(M,N)$. For this, we keep track of the matrix cell from which each cell's $F(i,j)$ was derived. This is done by storing a pointer, in each cell, to the cell from which *F(i,j)* was derived i.e. above, left, or above-left. This is illustrated in the complete example of a Protein sequence alignment using the above algorithm given in Figure 3 below. Here, the arrows point to the parent cell from which a particular cell has been computed and are used in the traceback step. The latter starts from cell *F(M,N)=F(7,10)* and moves backward to the cell from which the current cell has been derived. In general, if cell *(i,j)* was derived from cell *(i-1,j-1)*, we add the pair of symbols x_i, y_j to the front of the current alignment. If, however, it was derived from cell *(i-1,j)*, we add x_i to the front of alignment *X* and a gap to the front of alignment *Y*. Finally, if cell *(i,j)* was derived from cell *(i,j-1)*, we add y_i to the front of alignment *Y* and a gap to the front of alignment *X*.



		H	E	A	G	W	A	G	H	E	E
	0	**-8**	**-16**	-24	-32	-40	-48	-56	-64	-72	-80
P	-8	-2	-9	**-17**	**-25**	-33	-42	-49	-57	-65	-73
A	-16	-10	-3	-4	-12	**-20**	-28	-36	-44	-52	-60
W	-24	-18	-11	-6	-7	-15	**-5**	**-13**	-21	-29	-37
H	-32	-14	-18	-13	-8	-9	-13	-7	**-3**	-11	-19
E	-40	-22	-8	-16	-16	-9	-12	-15	-7	**3**	-5
A	-48	-30	-16	-3	-11	-11	-12	-12	-15	**-5**	2
E	-56	-38	-24	-11	-6	-12	-14	-15	-12	-9	**1**

Best Global Alignment:
```
H E A G A W G H E - E
- - P - A W - H E A E
```

Figure 3. Illustration of the Needleman-Wunsch Algorithm

2.2 Local Alignment: The Smith-Waterman Algorithm

Instead of looking for the best global alignment of two biological sequences *X* and *Y*, a much more useful algorithm would look for the best alignment between subsequences of *X* and *Y*. This is particularly useful when comparing longer biological sequences, where global alignment can lead to weak correlation and hence misleading results, or when comparing two highly diverged sequences where only part of the original sequence has been under strong enough selection to preserve noticeable similarity [1].

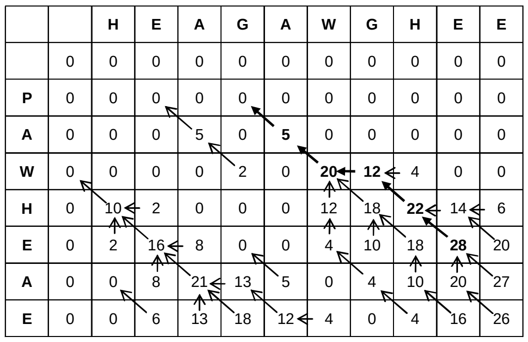
The Smith-Waterman algorithm is a dynamic programming algorithm, which finds the best scoring alignment of subsequences of *X*, *Y* i.e. the best local alignment of the two sequences [19]. It is closely related to the aforementioned Needleman-Wunsch algorithm, and differs from it in two points:

- *The recursion equation*: In the Smith-Waterman algorithm, this is given by the following equation:

$$F(i,j) = \max\begin{cases} 0 \\ F(i-1,j-1)+s(x_{i,}y_j), \\ F(i-1,j)-d, \\ F(i,j-1)-d \end{cases} \quad (2)$$

  Compared to equation (1), the term 0 is added to the maximum expression since it is always better to start a new local alignment if the best alignment so far has a negative score. As a consequence of this, the top row and left column values of the alignment matrix i.e. F(i,0) and F(0,j) should be set to 0's instead of *–i*d* and *–j*d* respectively.
- *Traceback*: Since we are looking for the best local alignment, the maximum score could lie anywhere on the matrix. It is hence necessary to find the maximum score in the matrix and start the traceback procedure, as explained above in the Needleman-Wunsch algorithm, from that position. The traceback procedure ends when 0 is met. The latter could also be anywhere in the alignment matrix.

The Smith-Waterman algorithm is illustrated Figure 4 below where the best local alignment between the same two sequences given in Figure 3 above is found.



		H	E	A	G	A	W	G	H	E	E
	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	5	0	**5**	0	0	0	0	0
W	0	0	0	0	2	0	**20**	**12**	4	0	0
H	0	10	2	0	0	0	12	18	**22**	14	6
E	0	2	16	8	0	0	4	10	18	**28**	20
A	0	0	8	21	13	5	0	4	10	20	27
E	0	0	6	13	18	12	4	0	4	16	26

Best Local Alignment:
```
AW GHE
AW - HE
```

Figure 4. Illustration of the Smith-Waterman Algorithm

2.3 The Case of Overlapped Matches

In some instances, it is useful to allow for overlapping between sequences when aligning them (see Figure 5).

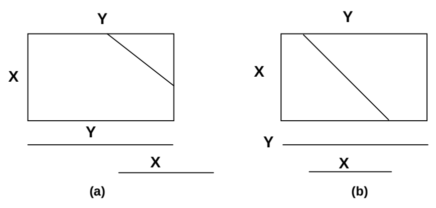
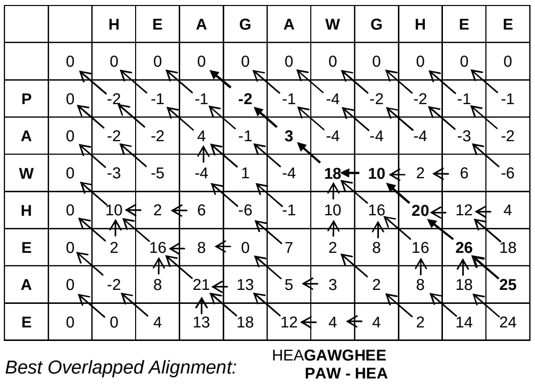


Figure 5. Overlapped sequence matching

This is a type of global alignment in which sequences are simply allowed to overlap without incurring any penalty. Hence, the initial value of *F(i,0)* and *F(0,j)* should be set to 0 instead of *–i*d* and *–j*d* respectively (see Figure 6). Moreover, here, the traceback can start anywhere on the right border or bottom border of the alignment matrix (instead of starting at the last element of the matrix *F(M,N)* as in the case of the Needleman-Wunsch algorithm).



		H	E	A	G	A	W	G	H	E	E
	0	0	0	0	0	0	0	0	0	0	0
P	0	-2	-1	-1	**-2**	-1	-4	-2	-2	-1	-1
A	0	-2	-2	4	-1	**3**	-4	-4	-4	-3	-2
W	0	-3	-5	-4	1	-4	**18**	**10**	2	6	-6
H	0	10	2	6	-6	-1	10	16	**20**	12	4
E	0	2	16	8	0	7	2	8	16	**26**	18
A	0	-2	8	21	13	5	3	2	8	18	**25**
E	0	0	4	13	18	12	4	4	2	14	24

Best Overlapped Alignment:

```
HEAGAWGHEE
   PAW - HEA
```

Figure 6. Illustration of the case of overlapped matches

2.4 Alignment with Affine Gap Scores

As mentioned earlier, affine gap penalties provide a more realistic model of the biological phenomenon of residue insertions and deletions. The affine gap penalty is defined using two constants *d* and *e* as follows (where *g* is the gap length):

$$Penalty(g)=-d-(g-1)*e$$

A similar algorithm to the one presented for linear gap penalties can be used here (both for local and global alignment). However, multiple values of each pair of residue *(i,j)* need to be computed instead of just one (i.e. *F(i,j)*). Figure 7 illustrates the case where three values need to be computed for each residue pair [1].

```
I  G  A  xi
L  G  V  yj
```

$$\longrightarrow F(i,j)=\max\begin{cases}F(i-1,j-1)+s(x_i,y_j),\\ I_x(i-1,j-1)+s(x_i,y_j),\\ I_y(i-1,j-1)+s(x_i,y_j)\end{cases}$$

```
A  I  G  A  xi
G  V  yj  -
```

$$\longrightarrow I_X(i,j)=\max\begin{cases}F(i-1,j)-d,\\ I_X(i-1,j)-e,\end{cases}$$

```
G  A  xi  -  -
S  L  G   V  yj
```

$$\longrightarrow I_y(i,j)=\max\begin{cases}F(i,j-1)-d,\\ I_y(i,j-1)-e,\end{cases}$$

Figure 7. The case of affine gap penalties

The top case corresponds to the best score *F(i,j)* up to *(i,j)* where x_i is aligned to y_j. The following case gives the best score $I_x(i,j)$ where x_i is aligned to a gap, whereas the last case gives the best score $I_y(i,j)$ where y_j is aligned to a gap.

Another algorithm which can be used in the case of affine gap penalties uses only two values for each residue pair instead of three and is given by the following recursive equations [1]:

$$F(i,j)=\max\begin{cases}F(i-1,j-1)+s(x_i,y_j),\\ I(i-1,j-1)+s(x_i,y_j)\end{cases} \tag{3}$$

$$I(i,j)=\max\begin{cases}F(i,j-1)-d,\\ I(i,j-1)-e,\\ F(i-1,j)-d,\\ I(i-1,j)-e\end{cases}$$

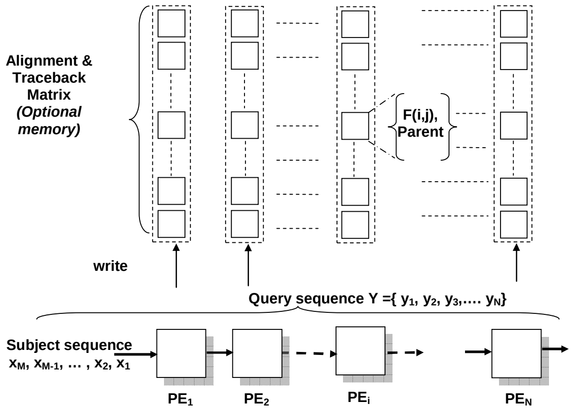
3. High Performance Biological Sequence Analysis: Previous Work

The computational complexity of the above dynamic programming algorithms for pairwise sequence alignment is proportional to the product of the lengths of the two sequences to be aligned i.e. *O(MxN)*. Given the sheer immensity of biological sequence databases and their exponential growth rate, a PC-based implementation of the above algorithms quickly runs into problems. For instance, assuming a Protein database of 100 million residues, a sequence of length 1000 would need something of the order of 10^{11} ($=10^8$x1000) basic operations to find the optimal alignment. Assuming a PC can perform 30 million of these basic operations per second, one single sequence alignment (with all database entries) would take around 1 hour to complete. It is hence clear that the computation time would quickly become an issue if we want to scan the database with tens or hundreds of sequences [1][9].

In order to speed up sequence analysis applications, a number of parallel architectures have been developed. Single Instruction Multiple Data (SIMD) architectures based on a network of programmable processors are among these solutions and include the MGAP [6], Kestrel [7] and Fuzion [8]. Although such architectures are capable of considerable speed-ups compared to a standard PC solution, they are often costly both in terms of design and programming [11]. Other solutions have used special purpose hardware for the implementation of parallel processing elements with the aim of increasing processing density and achieve even higher speed-ups. Such architectures also allow for systolic arrays to be implemented, a computing paradigm that is extremely suitable for the dynamic programming algorithms presented above. Instances of this family of architectures include BISP [20], SAMBA [21] and BIOSCAN [22]. The advent of reconfigurable hardware in the form of FPGAs makes such architectures even more appealing. FPGAs, like ASICs, are capable of providing considerable speed-ups compared to general purpose processors with the added convenience of reprogrammability. An algorithm implementation could hence be tuned to different needs both at compile time and at run-time. Moreover, FPGAs are now riding the process technology curve [23] which makes them even more attractive a solution as a reliable high performance platform for biocomputing [24][25][26]. For instance, a number of FPGA implementations of the Smith-Waterman algorithm have been reported in the literature recently [9][10][11]. However, none of these implementations offers the same degree of parameterisability as our implementation, as will be apparent in subsequent sections. Moreover, our implementation was designed using a high level hardware language in the form of the ANSI-C based Handel-C language [27], and achieved performance figures comparable to the best reported results in the literature, if not better, as will be shown in Section 5. This in itself is important as it means that higher level hardware languages can be used to achieve high performance implementations of computational biology applications, hence bridging the aforementioned gap between the bioinformatics applications and high performance hardware.

4. Our Hardware Implementation

Figure 8 presents a linear systolic array implementation of a general purpose pairwise sequence alignment problem based on the dynamic programming algorithms presented above. The linear systolic array consists of a pipeline of basic processing elements (PE_i) each holding one query residue x_i, whereas the subject sequence is shifted systolically through the array. This architecture could be easily deducted from a data dependency graph of the dynamic programming algorithms presented in Section 2 [20].



Alignment & Traceback Matrix (Optional memory)
F(i,j), Parent
write
Query sequence Y ={ y1, y2, y3,.... yN}
Subject sequence xM, xM-1, ... , x2, x1
PE1
PE2
PEi
PEN

Figure 8. Linear processor array architecture for pairwise sequence alignment

Each PE performs one elementary calculation (see Equation 1). Indeed, each cycle, the PE generates one alignment matrix element *F(i,j)* and saves the direction of the cell from which the result has been derived (called *Parent* in Figure 8). The latter can be any of three nominal values: *top*, *left*, or *diagonal,* and could hence be represented in 2 bits. Each PE end up generating one column of the alignment matrix after *M* cycles (*M* being the length of the subject sequence). However, the calculation at PE_{i+1} depends on the result from PE_i, which means that each PE is one cycle behind its predecessor. The full alignment of two sequences of lengths *N* and *M* is hence achieved in *M+N-1* cycles. Figure 9 illustrates the execution of the recursive equation of the Smith Waterman algorithm in such architecture.

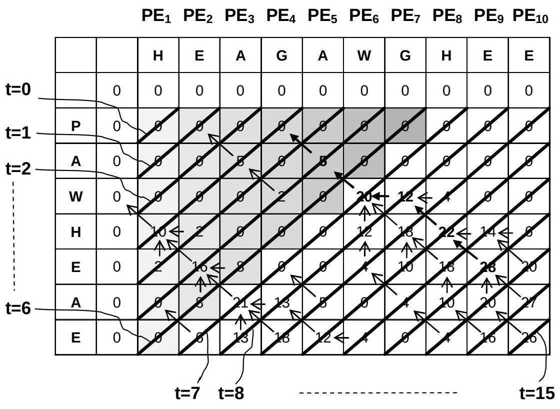
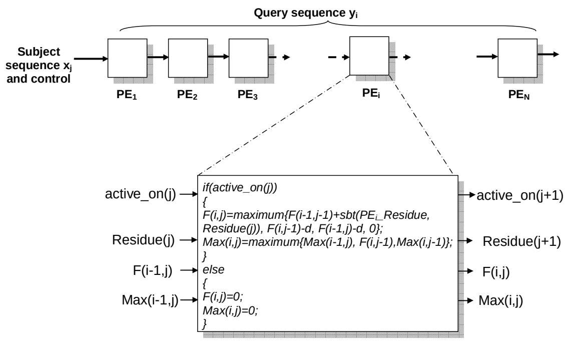


Figure 9. Illustration of the execution of the Smith-Waterman example of Figure 3 on the linear array processor of Figure 8

The architecture of figure 8 can cater for different sequence symbol types, sequence lengths, match scores and matching task. Indeed, the sequence symbol type e.g. DNA or Proteins, will only influence the word length of the input sequence e.g. 2 bits for DNA and 5 bits for Proteins. Moreover, the query sequence length dictates the number of PEs. The match score attributed to a symbol match depends on the substitution matrix used. Given a particular substitution matrix e.g. BLOSUM50, all possible match scores for a particular symbol represent one column in the substitution matrix. Hence, for each PE, we store the corresponding symbol's column in the substitution matrix, which we use as a look-up-table. A different substitution matrix will hence simply mean a different look-up-table content. The penalty attributed to a gap can also be stored in the PE. In the case of linear gap penalties, only one value is needed (*d*), whereas affine gap penalties need two values (*d* and *e*) as explained in section 2.4 above. Note that in the case affine gap penalties, multiple values of each pair of residue need to be computed in the recursion equation instead of just F(i,j). This could either be two (F(i,j) and I(i,j)) or three (F(i,j), $I_x(i,j)$ and $I_y(i,j)$) as explained in Section 2.4 above.

Finally, the linear array of figure 8 can also cater for different matching tasks with few changes. For instance, the difference between global alignment, local alignment and overlapped matching resides in the initial values of the alignment matrix (border values), the recursive equation implemented by the PE as well as the starting cell of the traceback procedure. The border values of the alignment matrix simply represent initial values attributed to each PE (0's for local alignment and overlapped matching and *–j*d* in the case of global alignment) as well as the initial values attributed to PE_0 (0's for local alignment and overlapped matching, and *-i*d* for global alignment). Moreover, the recursive expression in the case of local alignment is not allowed to take on negative values as it saturates to zero, unlike the case of global alignment and overlapped matching. This simply implies the use of *saturated* arithmetic in the case of local alignment. Finally, the traceback in the case of local alignment needs to start from the maximum element in the alignment matrix, which requires the array structure to calculate the maximum of all results *F(i,j)*. This can be performed in a systolic manner using Figure 8 architecture with each PE calculating the *maximum-so-far* and broadcasting it to the next PE in the chain. Figure 10 gives the pseudo-code implemented by the PE in the case of the Smith-Waterman algorithm with linear gap penalty. The corresponding pseudo-code for global alignment and overlapped matching would only be altered by the initial conditions and a slight modification to the recursion equation as explained in section 2 above (see equation 1 in particular).



```
if(active_on(j))
{
F(i,j)=maximum{F(i-1,j-1)+sbt(PEi_Residue,
Residue(j)), F(i,j-1)-d, F(i-1,j)-d, 0};
Max(i,j)=maximum{Max(i-1,j), F(i,j-1),Max(i,j-1)};
}
else
{
F(i,j)=0;
Max(i,j)=0;
}
```

Figure 10. Linear array architecture for the Smith-Waterman Algorithm (using linear gap)

In the case of overlapped matching, the maximum calculation needs to be performed only in the last PE (PE_N) and at the last calculation cycle of each PE since the traceback starts from the cell with the maximum value in the bottom row and right-most column. Global alignment, however, need not calculate the maximum element in the alignment matrix as it always starts the trace-back from the last element.

The above clearly shows that it is possible to capture all of the variations enunciated in Section 2 in a single core description and achieve custom implementations from user parameters supplied at compile time. This will explained in Section 4.2 below, but before that, the case of long sequences is considered.

4.1 The case of long sequences

In reality, biological sequences are often hundreds, if not thousands, long. This means that the memory requirement of the above dynamic programming algorithms is often measured in Megabytes. For instance, assuming 16 bits per cell, the alignment matrix of two sequences of length 2000 requires 64Mbit of memory. Storing such amount of data on FPGAs is clearly not possible in today's technology. Hence, only the linear systolic array of Figure 8 can be implemented on FPGA (i.e. without the alignment matrix storage). This is not problematic in reality since a query sequence is often compared to a database of thousands of sequences before few candidates with the highest match scores are actually aligned. Hence, the linear systolic array can be used to measure the maximum match score between the query sequence and each subject sequence in the database before few subject sequences (i.e. those with the highest maximum match score) are chosen. The complete alignment (i.e. with traceback) can thus be performed with these few sequences in software [11]. The time needed for this is relatively small compared to scanning a whole database of sequences searching for high scoring matches.

In our core, the alignment matrix of Figure 8 is optional. The user can thus decide not to implement it at compile time, in which case the hardware inferred is shown in Figure 11. Here, only the final matching score between two sequences (i.e. the maximum element in the alignment matrix in the case of local alignment, the maximum element in the bottom row and right-most column of the alignment matrix in the case of overlapped matching, and the last element in the alignment matrix in the case of global alignment) is stored.

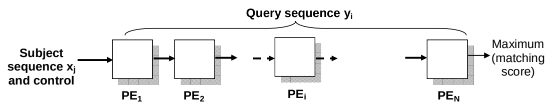


Figure 11. Linear Array Architecture with no Alignment Matrix Storage

The on-chip memory limitation is not the only problem, however, in the case of long sequences. Indeed, logic limitation also means that the number of PEs that could be implemented on an FPGA is limited. For instance, the maximum number of PEs that could be implemented on a Xilinx XC2V6000 Virtex-II FPGA in the case of the Smith-Waterman algorithm with affine gap penalties is ~250. Clearly, this is not sufficient for many real world sequences. The solution in such cases is to partition the algorithm in hand into small alignment steps and map the partitioned algorithm on a fix size linear systolic array (whose size is dictated by the FPGA chip in hand). This problem is well studied in the VLSI design arena [28][29]. The following illustrates the solution.

Let us assume the general case of a query sequence of length M and a linear systolic array of size υ where $M > \upsilon$ and $k = \lceil M/\upsilon \rceil$. First, the necessary linear systolic array of length M is conceptually extended to an array of length $k*\upsilon$ with the last $k*\upsilon\text{-}M$ PEs filled zero-value substitution table columns. That way, these extra PEs do not influence the overall alignment result. After this conceptual step, the resulting linear systolic array of length $k*\upsilon$ is folded into the physical array of length υ. In reality, υ represents the maximum number of PEs that we can fit on the FPGA chip in hand. As a result of this mapping, the alignment process is performed in *k* passes over the linear array. For this, we need a FIFO to store intermediate results from each pass before they are fed back to the array input for the next pass (see Figure 12). The depth of the FIFO is dictated by the length of the subject sequence. In general, the FIFO depth should be sufficient to hold the biggest sequence in the database of sequences against which the query sequence is compared.

Another consequence of the above mapping is that each PE should now hold *k* substitution matrix columns (or look-up-tables) instead of just one. Indeed, PE_i in the folded architecture should now be able to hold the look-up-tables of PE_i, $PE_{\upsilon+i}$, $PE_{2*\upsilon+i}$,…., $PE_{i+(k-1)*\upsilon}$. A *pass counter* is used to switch between the *k* look-up-tables at different passes. In terms of performance, a folded architecture by a factor *k* results in a slow-down in performance by a factor of *k* compared to a non-folded fully pipelined architecture as data has to be cycled around the same array *k* times instead of being processed in parallel across *k* cascaded arrays. The latency of the folded architecture is however the same as a non-folded architecture. The size of the FIFO is adjusted to guarantee such lossless latency.

In order to load the initial values of the look-up tables used by the PEs, a serial configuration chain is used, as illustrated in Figure 13. When the control bit *Cfg* is set to 1, the circuit is in configuration mode. Distributed memory in each PE then behaves as a register chain. Each PE configuration memory is loaded with the corresponding look-up tables sequentially. At the end of the configuration, *Cfg* is reset to 0 indicating the start of the operation mode. The distributed memory now acts as Read Only Memory. It is used by the dynamic programming recursion equation part of the PE's circuitry as illustrated in Figure 13. The extra configuration circuitry needed is for this is implemented on FPGA fabric. While dynamic reconfiguration could have been harnessed to reduce this area overhead for Xilinx FPGAs, for instance, this would have made our core FPGA-platform-dependent. As it stands, our core can be implemented on different FPGA families and architectures e.g. from Xilinx, Altera, and Actel, or even as a standard ASIC solution.

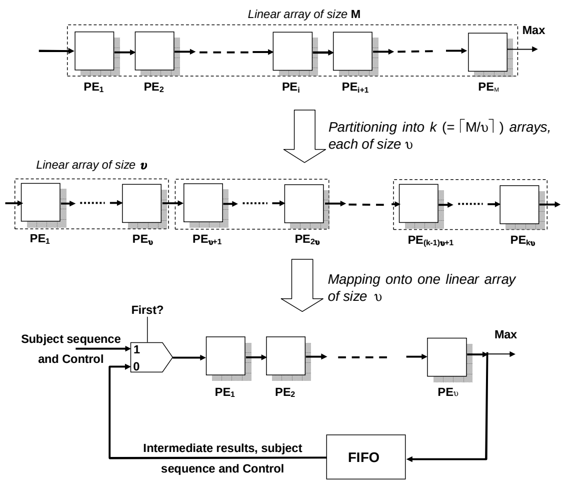

Linear array of size M
Max
PE1
PE2
PEi
PEi+1
PEM
Partitioning into k (= ⌈M/υ⌉) arrays, each of size υ
Linear array of size υ
PE1
PEυ
PEυ+1
PE2υ
PE(k-1)υ+1
PEkυ
Mapping onto one linear array of size υ
First?
Subject sequence and Control
1
0
Max
PE1
PE2
PEυ
Intermediate results, subject sequence and Control
FIFO

Figure 12. Partitioning/Mapping of a sequence alignment algorithm on a fixed size systolic array

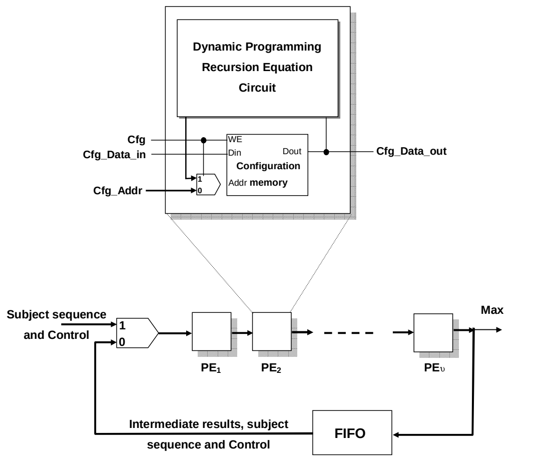

Dynamic Programming Recursion Equation Circuit
Cfg
Cfg_Data_in
Cfg_Addr
WE
Din
Dout
Configuration
Addr memory
1
0
Cfg_Data_out
Subject sequence and Control
1
0
Max
PE1
PE2
PEυ
Intermediate results, subject sequence and Control
FIFO

Figure 13. Partitioning/Mapping of a sequence alignment algorithm on a fixed size systolic array

4.2 Core implementation

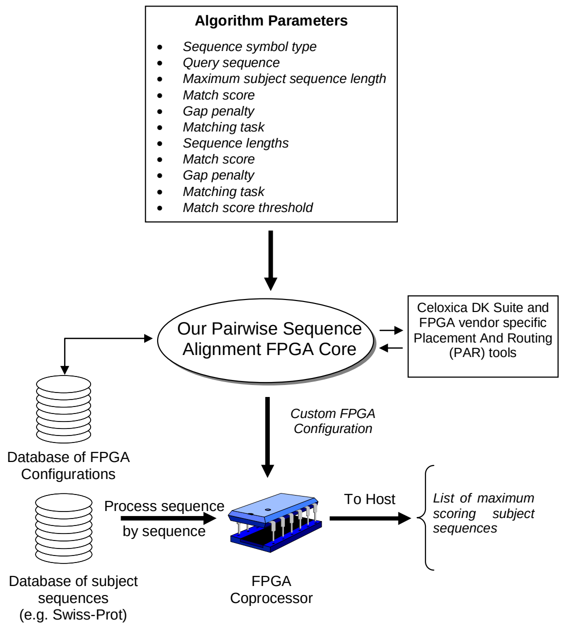
As mentioned earlier, we have captured all of the above variations of a generic pairwise sequence alignment algorithm into a single FPGA core, written in the Handel-C language [27]. The latter is a hardware language that allows hardware designers and application developers to program FPGAs in a C-like syntax, hence reducing the gap between algorithms and hardware. Extensive use has been made of Handel-C pre-compiler directives, macro procedures and macro expressions to parameterise the code. The final core that we have developed is prameterisable in terms of the following:

- The *sequence symbol type* i.e. DNA, RNA, or Protein sequences
- *Query sequence*: Here the query sequence length dictates the number of PEs used. If this could fit into the FPGA in hand, a pairwise sequence alignment can be achieved in one single pass. Otherwise, the above partitioning/mapping (see Figure 12) is performed. In the latter case, the

necessary FIFO instantiation as well as control circuitry for multi-pass processing is automatically generated by the core.

- *Maximum subject sequence length*: this will dictate the minimum necessary processing wordlength and FIFO depth, if necessary.
- The *match score* i.e. the score attributed to a symbol match. This is supplied in the form of a substitution matrix e.g. BLOSUM or PAM matrices.
- The *gap penalty*: This could be either *linear* or *affine*. In the case of affine gap penalty, the core will automatically infer the necessary architecture based on the values of *d* and *e* as well as the substitution matrix in hand (see section 2.4 above).
- The *matching task* i.e. the algorithm used to match sequences. This could be *global alignment, local alignment* or *overlapped matching*.
- *A match score threshold*: This is a match score threshold below which any subject sequence is rejected i.e. only those subject sequences with a higher match score warrant further analysis.

Given the above parameters, our core generates custom FPGA configurations (see Figure 14). It is worth mentioning that the optimal processing wordlength is automatically inferred from the user-supplied parameters based on a worst-case range analysis.



Algorithm Parameters
Sequence symbol type
Query sequence
Maximum subject sequence length
Match score
Gap penalty
Matching task
Sequence lengths
Match score
Gap penalty
Matching task
Match score threshold
Our Pairwise Sequence Alignment FPGA Core
Celoxica DK Suite and FPGA vendor specific Placement And Routing (PAR) tools
Database of FPGA Configurations
Custom FPGA Configuration
Process sequence by sequence
To Host
List of maximum scoring subject sequences
Database of subject sequences (e.g. Swiss-Prot)
FPGA Coprocessor

Figure 14. FPGA co-processor architecture based on our core

The user can easily modify the parameters by simply typing new values to the screen and pressing the compilation button. All the parameters needed are assembled in a header file which the user can edit using a simple text editor.

5. Results and Discussion

As mentioned above, our core has been captured in the Handel-C language. The code did not use any FPGA-specific directives e.g. specific resource inference or placement constraints. This makes it directly retargetable across a variety of platforms including Xilinx and Altera FPGAs. Celoxica's DK4 suite was used to compile our core into EDIF, whereas Xilinx ISE8.1 tool was used to generate the FPGA bitstreams.

A single PE in the case of the Protein sequence processing using the Smith-Waterman algorithm, with linear gap penalty and 16 bit processing wordlength, consumes ~30 slices on average on Xilinx Virtex-II FPGAs, whereas an equivalent affine gap penalty using the equations given in Figure 7, consumes ~85 slices. An equivalent single PE with affine gap penalty using Equations (3), however, consumes ~70 slices. Consequently, a Xilinx XC2V6000-4 FPGA, which contains 33792 slices, can easily fit 250 PEs on chip.

Table 1 presents sample performance figures for instances of the core in the case of Protein sequence alignment with a single pass i.e. the query sequence is fully fitted on chip. *Affine2* and *Affine3* refer to the affine gap models given in Equations (3) and Figure 7 respectively. The former uses only two values for each residue pair in the recursive equations, whereas the latter uses three values (see section 2.4 above). The CUPS (or Cell Updates Per Second) performance used in Table 1 is a common performance measure used in computational biology. Its inverse represents the equivalent time needed for a complete computation of one entry of the alignment matrix, including all the comparisons, additions and maximum computations. The peak CUPS of our implementation is measured by multiplying the number of PEs and the maximum clock frequency.

The clock frequency for all instances shown, and others not shown here, is between 40 and 60MHz. The variations in clock frequency, for instances with the same wordlength, are largely attributed to the high level synthesis tool and, to a lesser extent, the placement and routing tool. The choice of not performing any target FPGA-dependent optimisations is deliberate in order to show the merits of a high level synthesis approach, for it is our aim to reduce the gap between bioinformatics applications and low level FPGA hardware. Nonetheless, these speed figures could be increased further, if need be, at the expense of a higher design effort.

Table 1. Core performance for different instances of our core on a Xilinx XC2V6000-4 FPGA

Number of PEs	Gap Penalty	Processing Word length	Max Speed (MHz)	Peak Performance (CUPS x10^9)
Needleman-Wunsch				
252	Linear	16	50.6	12.75
Overlapped Matches				
252	Linear	16	50.0	12.60
Smith-Waterman				
100	Linear	16	43.5	4.35
252	Linear	10	47.7	12.02
100	Affine 2	16	66.7	6.67
168	Affine 2	16	47.6	8.00
100	Affine 3	10	58.8	5.88
168	Affine 3	16	40.0	6.72

Table 2 below presents sample results for instances of our core in the case of local alignment with a multi-pass implementation where *k* (the number of passes, or folding factor) is equal to 3 and 12 respectively.

Table 2. Core performance for different instances of a multi-pass implementation (with local alignment)

Number of PEs	Gap Penalty	Processing Word length	Clock frequency (MHz)	Peak Performance (CUPS x10^9)
k=3				
252	Linear	10	40.0	10.09
168	Affine 2	10	62.5	10.50
168	Affine 3	10	45.6	7.66
k=12				
168	Linear	10	40.3	6.77
119	Affine 3	10	50.4	5.99

Software implementations of sequence alignment algorithms, equivalent to our hardware implementation, were written in C and run at a speed equivalent to 50 MegaCUPS on a 1.6 GHz Pentium-4 PC. This means that our hardware core outperforms equivalent software implementations by two orders of magnitudes. Given the relative cost of FPGAs compared to general purpose processors (often in the order of 10:1) this performance largely offsets their cost, which shows that FPGAs could well be an viable economical implementation platform for biological sequence analysis applications. Nonetheless, the CUPS metric reflects the peak performance and does not account for communication overheads e.g. pipeline filling/flushing and host to FPGA communication overheads.

In attempt to account for these overheads, a real hardware implementation of our core has been achieved on an Alpha Data XP FPGA Mezzanine PCI-board, which has a Virtex-II Pro FPGA on it, and used the Swiss-Prot Protein database [30]. Figure 15 illustrates our implementation. In it, the database of subject queries is stored on the FPGA board's off-chip memory. The FPGA reads each sequence in turn from memory and compares it to a query sequence initially supplied by the high level application running on the host. At the end of processing, the FPGA supplies a list of high scoring subject sequences back to the host application. The database used in our implementation was a subset of the Swiss-Prot database consisting of 288 subject sequences with an average sequence length equal to that of the whole database, and a query sequence of 362 residues. The number of PEs implemented on the FPGA was 135. Running at a clock frequency of 40 MHz, the overall alignment took 88 milliseconds to perform on the board with an initial configuration time of 244us. An equivalent software implementation written in C took 5516 milliseconds to run on a 1.6 GHz Pentium-4 PC. This represents a 62x speed-up.

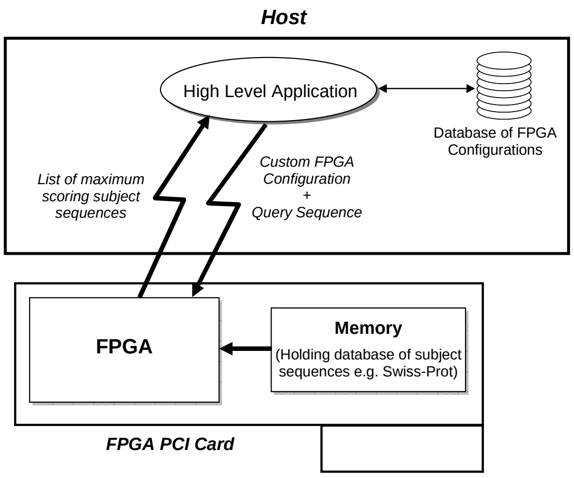

Host
High Level Application
Database of FPGA Configurations
List of maximum scoring subject sequences
Custom FPGA Configuration + Query Sequence
FPGA
Memory
(Holding database of subject sequences e.g. Swiss-Prot)
FPGA PCI Card

Figure 15. Organisation of the real hardware implementation

Performing fair and meaningful comparisons with other implementations is difficult given the difference in technologies and performance measures used, as well as the relatively narrow scope of some of the implementations. Nonetheless, the following attempts to make a useful comparison of our implementation with others reported in the literature.

First, the SIMD architectures in the form of MGAP, Fuzion and Kestrel are all based on an array of PEs with nearest neighbour connections. The MGAP architecture reported in [6] performs global alignment for DNA sequences. Running at a clock frequency of 25MHz, it can achieve 0.1MCUPS, which is much slower than our implementation. This is however not surprising given the gap in the technologies used. The Kestrel architecture is a single board programmable parallel processor with 512 processing elements (PEs). Running at 20MHz, the Kestrel searches a 10 Mbase database with a query size less than 512 in 12 second, giving a performance of 0.4 GCUPS which is still an order of magnitude lower than our implementation. Finally, the Fuzion 150 system [7] is a linear SIMD array of 1536 PEs with a reported peak performance of ~2.5 GCUPS, which is 2 to 3 times slower than our implementation. Besides, one could question the economic viability of such purpose-built SIMD architectures compared to FPGA-based off-the-shelf solutions. Indeed, FPGA technology has clear economies of scale and scope advantages compared to purpose-built SIMD platforms.

Performing fair and meaningful comparisons with equivalent FPGA implementations is also a difficult task given the difference in characteristics (architecture, part and speed grade) of the FPGAs used. Moreover, some publications do not present all of the experimentation parameters. Nonetheless, the following attempts to make meaningful comparisons with some FPGA implementations reported in the literature.

The FPGA implementations presented in [13][14][15] are restricted to DNA sequences, which are a special case of our implementation. The implementation reported in [14] achieves 1260 GCUPS peak performance on a Xilinx XC2V6000-4 FPGA part. The implementations reported in [13] and [15] achieve over 3200 GCUPS on the same part. In comparison, our multi-purpose core achieves ~800 GCUPS on the same part. The difference in performance is justified by the fact that the above implementations have been optimised for DNA processing. The gain in performance is achieved at the expense of less flexibility as these solutions cannot be used for Protein sequence processing. The closest implementations to ours in terms of flexibility have been reported in [9], [10] and [11]. Compared to the implementation reported in [10] on a Xilinx XC2VP30 FPGA, our core achieves twice the speed. It also outperforms the implementation reported in [9] by 3:1. The Verilog-based implementation reported in [11], however, is the closest to our core implementation of all three, as it is targeted to the same FPGA part and employs a similar architecture. Compared to it, our core performs almost as well despite the fact that we have not introduced any placement constraints (unlike in [11]). This is, in part, a testament to the Handel-C language as well as the corresponding synthesis tool. Moreover, none of these three implementations offer the same degree of parameterisation as our core. Indeed, the implementation reported in [9] only supports the Smith-Waterman algorithm with linear gap penalty, albeit for both DNA and Protein sequences, whereas the implementation reported in [10] does not address the problem of partitioning/mapping. The implementation reported in [11] supports both partitioning/mapping and affine gap penalties, but has been designed specifically for Xilinx Virtex FPGAs. Our core on the other hand is FPGA-platform-independent and can be used to target any other FPGA architecture (e.g. from Xilinx, Altera, Actel). Moreover the affine gap model used in [11] is based on the equations given in Figure 7 only, and hence does not take advantage of the hardware optimisations introduced by Equations (3) (see Section 2.4 above).

6. Conclusion

In this paper, we have presented the detailed design and implementation of the most parameterisable FPGA core, reported in the literature, for pairwise biological sequence alignment. The skeleton is parameterised in terms of the sequence symbol type, the sequence lengths, the match score, the gap penalty and the matching task. It implements the algorithm in hand using a pipeline of basic processing elements, which are tailored to the algorithm parameters, with a number of built-in hardware optimisations. These include automatic hardware folding, automatic minimum wordlength inference and compile-time constant propagation. The skeleton results in high performance FPGA implementations which outperform equivalent desktop-based software implementations by two order-of magnitudes. While this in itself has been achieved previously through optimised hardware implementation for specific FPGA architectures, this paper shows that it is possible to achieve such performance using an FPGA-platform-independent hardware language. Indeed, our skeleton has been captured in the Handel-C language which means that the same code can be ported to different FPGA families and architectures. In our experience, Handel-C proved very convenient in describing scaleable and parameterised hardware architectures, with a relatively lower learning curve compared to other hardware description languages. However, the resulting optimised Handel-C description of our skeleton is in essence a hardware architecture description, rather than a software algorithm description, albeit using high level software constructs such as macro procedures, if/else control structures, as well as software-like data structures.

The work presented in this paper is part of a bigger effort by the authors which aims to harness the computational performance and reprogrammability features of FPGAs in the field of

Bioinformatics and Computational Biology. Future work includes the development of FPGA cores for sub-optimal sequence alignment algorithms including the BLAST algorithm, as well as the use of Hidden Markov Models for biological sequence analysis. On the hardware implementation front, we plan to make use of state-of-the-art FPGA-based computing platforms, namely the low latency, high bandwidth, Hypertransport-based FPGA boards, which will allow direct FPGA access to gigabytes of memory, with a data rates of several gigabytes per second.

Finally, it is worth mentioning that the core presented in this paper can be used for any string analysis application e.g. text processor or web server, with very little modification. We intend to explore the data mining application side of our work further in the future.

7. References

[1] Durbin, R., Eddy, S., Krogh, A., and Mitchison, G., '*Biological Sequence Analysis: Probabilistic Models for Proteins and Nucleic Acids'*, Cambridge University Press, Cambridge UK, 1998

[2] Hein, J. '*A New Methodology that simultaneously aligns and reconstructs ancestral sequences for any number of homologous sequences, when a phylogeny is given'*. Journal of Molecular Biology, 6, pp.649-668, 1989

[3] Altschul, S. F., Gish, W., Miller, W., Myers, E.W. and Lipman, D.J. '*Basic Local Alignment Search Tool*', Journal of Molecular Biology, 215, pp. 403-410, 1990.

[4] Pearson, W.R. and Lipman, D.J. *'FASTA: Improved tools for biological sequence comparison'*, Proceedings of the National Academy of Sciences, USA 85, pp. 2444-2448, 1988.

[5] Altschul, S. F., Madden, T. L., Schaffer, A. A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D. J. '*Gapped BLAST and PSI-BLAST: a new generation of protein database search programs*', Nucleic Acid Research, Oxford Journals, 25(17), pp. 3389-3402, 1997.

[6] Borah, M., Bajwa, R.S., Hannenhalli, S., and Irwin, M.J. '*A SIMD solution to the sequence comparison problem on the MGAP'*, ASAP'94 Proceedings, IEEE Computer Science, pp. 144-160, 1994

[7] Dahle, D., Grate L., Rice, E., and Hughey, R. '*The UCSC Kestrel general purpose parallel processor'*, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 1243-1249, 1999

[8] Schmidt, B., Schröder, H., and Schimmler, M '*Massively Parallel Solutions for Molecular Sequence Analysis*', Proceedings of the 1st IEEE International Workshop on High Performance Computational Biology, pp. 186-193, 2002.

[9] Yamaguchi, Y., Maruyama, T., and Konagaya, A. 'High Speed Homology Search with FPGAs', Proceedings of the Pacific Symposium on Biocomputing, pp.271-282, 2002.

[10] VanCourt, T. and Herbordt, M. C. 'Families of FPGA-Based Algorithms for Approximate String Matching', Proceedings of Application-Specific Systems, Architectures, and Processors, ASAP'04, pp. 354-364, 2004

[11] Oliver, T., Schmidt, B. and Maskell, D. '*Hyper customized processors for bio-sequence database scanning on FPGAs*', Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays

[12] Oliver, T., Schmidt, B., Maskell, D., Nathan, D., and Clemens, R., '*High-speed multiple sequence alignment on a reconfigurable platform*', International Journal of Bioinformatics Research and Applications, 2(4), 2006, 394-406.

[13]Bojanic, S., Caffarena, G., Pedreira., C., and Nieto-Taladriz, O., '*High Speed Circuits for Genetics Applications*', Proceedings of the 24th International Conference on Microelectronics (MIEL 2004), Vol. 12, pp. 517-524, 2004

[14] Puttegowda, K., Worek, W., Pappas, N., Dandapani, A., and Athanas, P., '*A Run-Time Reconfigurable System for Gene-Sequence Searching*', Proceedings of the 16th International Conference on VLSI Design VLSI'03, pp. 561 – 566, 2006.

[15] Caffarena et al, '*FPGA acceleration for DNA sequence alignment'*, Journal of Circuits, Systems and Computers 0218-1266, 16 (2), pp.245-266, 2007

[16] Fagin, B. S., Watt, J. G., Gross, R. '*A Special-Purpose Processor for Gene Sequence Analysis*', Computer Applications in the Biosciences', 9(2), pp. 221-226, April 1993

[17] Harrison G. A., Tanner, J. M., Pilbeam D. R., and Baker, P. T. '*Human Biology: An introduction to human evolution, variation, growth, and adaptability*', Oxford Science Publications, 1988

[18] Needleman, S. and Wunsch, C. '*A general method applicable to the search for similarities in the amino acid sequence of two sequences'* Journal of Molecular Biology, 48(3), pp.443-453, 1970

[19] Smith, T.F. and Waterman, M.S. Identification of common molecular subsequences. J. Mol. Biol., 147, pp.195-197, 1981

[20] Chow, E., Hunkapiller, T., Peterson, J., Waterman, M.S. '*Biological Information Signal Processor*', Proceedings of Application-Specific Systems, Architectures, and

Processors, ASAP ASAP’91, pp. 144-160, 1991.

[21] Guerdoux-Jamet, P., Lavenier, D. ‘*SAMBA: hardware accelerator for biological sequence comparison*’, Computer Applications in Biosciences, CABIOS, 12 (6), pp. 609-615, 1997.

[22] Singh, R.K. et al. ‘*BIOSCAN: a network sharable computational resource for searching biosequence databases*’, Computer Applications in Biosciences, CABIOS, 12(3), pp. 191-196, 1996.

[23] Butts, M. All chips will be reconfigurable, Tutorial, 13th International Conference on Field Programmable Logic and Applications, September 2003

[24] Hoang, D.T. ‘*Searching genetic databases on Splash 2*’, in Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, pp. 185-191, 1993.

[25] Gokhale, M. et al. '*Processing in memory: The Terasys massively parallel PIM array*', Computer, 28 (4), pp. 23-31, April 1995.

[26] TimeLogic Corporation, ‘*Decypher Scalable, High Performance Biocomputing Solutions*’, http://www.timelogic.com/

[27] The Handel-C Language Reference Manual, Celoxica Plc, http://www.celoxica.com

[28] Kung, S. Y. ‘*VLSI Array Processors*’, Prentice-Hall, 1988

[29] Moldovan, D. I. and Fortes, J. A. B. ‘*Partitioning and mapping of algorithms into fixed size systolic arrays*’, IEEE Transactions on Computers, 35(1), pp. 1-12, January, 1986

[30] Boeckmann, B., et al., ‘*The SWISS-PROT protein knowledgebase and its supplement TrEMBL’* in 2003 Nucleic Acids Research, Vol. 31, pp. 365-370, 2003.