Design and Implementation of an FPGA-based Core for Gapped BLAST Sequence Alignment with the Two-Hit Method

Server Kasap, Khaled Benkrid and Ying Liu

The University of Edinburgh, School of Electronics and Engineering,
Mayfield Road, Edinburgh EH9 3JL, Scotland, UK
(s.kasap,k.benkrid,y.liu)@ed.ac.uk

Abstract--**This paper presents the design and implementation of the first FPGA-based core for Gapped BLAST sequence alignment with the two-hit method, ever reported in the literature. Gapped BLAST with two hit is a heuristic biological sequence alignment algorithm which is very widely used in the Bioinformatics and Computational Biology world. The architecture of the core is parameterized in terms of sequence lengths, match scores, gap penalties and cut-off, and threshold values. It is composed of various blocks each of which performs one step of the algorithm in parallel. This results in high performance and efficient FPGA implementations, which easily outperform equivalent software implementations by one order of magnitude or more. Furthermore, the core was captured in an FPGA-platform-independent language, namely the Handel-C language, to which no specific resource inference or placement constraints were applied. Hence, the core can be ported to different FPGA families and architectures.**

Index Terms—**FPGA, Gapped BLAST, Sequence Alignment, Two-Hit method**

I. Introduction

Biological sequence alignment is a widespread operation in the world of Bioinformatics and Computational biology (BCB) where sequence databases are searched to find similar sequences to a query sequence, by aligning each subject sequence in the database to the query sequence [1]. The main aim of this operation is to obtain information about a newly discovered biological sequence (i.e. Protein, DNA or RNA) from other known sequences (stored in the database). For instance, if a new sequence is similar to a known sequence representing a cancerous gene, then information pertained to the functionality of the new sequence can be inferred, which is useful in early disease diagnosis and drug engineering. Besides this, study of evolutionary development and history of species can be done through biological sequence alignment [1] [2].

Biological sequence alignment is a computationally intensive operation and with exponentially growing sequence databases (see Figure 1) this task cannot be achieved by desktop computer systems within realistic execution times. Hence, there is a need for faster computing platforms to cope with this growth. Recently, Field Programmable Gate Arrays (FPGAs) have been proposed as a high performance reconfigurable hardware platform for sequence alignment algorithms [3] [4] [5]. Indeed, FPGAs are capable of providing high speed-ups compared to general purpose processors with the convenience of reprogrammability, which makes them an attractive platform to accelerate biocomputing applications.

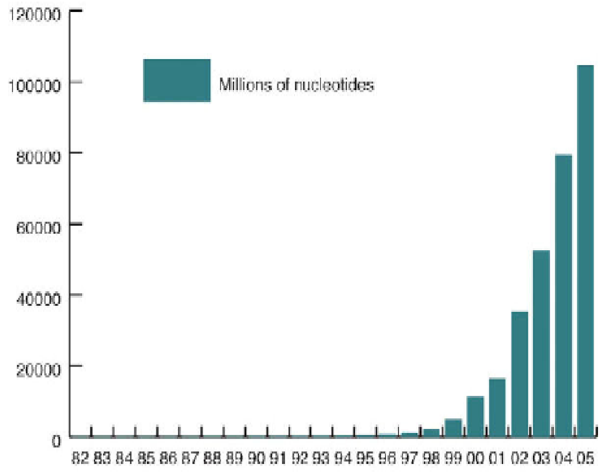


Figure 1. Exponential growth of biological sequence databases over years [1]

There are various biological sequence alignment algorithms some of which are exhaustive and give optimal alignments (e.g. Needleman-Wunsch [6], Smith-Waterman [7]) and some of which are heuristic and give sub-optimal alignments (e.g. FASTA[8], BLAST[9]). In this paper, we concentrate on Basic Local Alignment Search Tool (BLAST) which is a local alignment algorithm. Although it is heuristic, in the sense that it produces local alignments which are not always optimal, it is much faster than ordinary exhaustive dynamic programming algorithms. The design and implementation of a variant of BLAST, namely Gapped BLAST with two-hit method [10], is presented in this paper. The design is captured in the Handel-C language [13] which is a FPGA-platform-independent language, making our design portable across a number of FPGA architectures (e.g. Xilinx, Altera). The remainder of this paper will first present essential background information on the general BLAST algorithm. Then, the design and implementation of our FPGA core for Gapped BLAST with the two-hit method will be detailed. After that, comparative timing performance evaluation of our core against equivalent desktop software is presented. Finally, conclusions are laid out with plans for future work.

II. Background

Biological sequences evolve through mutation, selection and random genetic drift [11]. Mutation, in particular manifests itself through 3 main processes which are as follows:

- Substitution of residues: Residue A in the sequence is substituted by another residue B.
- Insertion of residues: New residues are inserted into the sequence.
- Deletion of residues: Existing residues in the sequence are deleted.

Insertions and deletions result in *gaps* which are taken into consideration when aligning biological sequences. The degree of alignment of biological sequences is measured by a score which is obtained by the summation of score terms of each aligned pair of residues with possible gap penalty terms. Score terms for each aligned residue pair are obtained from probabilistic models which are stored in score or substitution matrices such as BLOSUM50 [1]. The latter is a 20x20 matrix for protein sequence residues. On the other hand, gap penalties depend on the length of the gap and are independent of gap residues. There are two main types of gap penalties:

- Linear gap penalty: The cost of a gap of length g is given by following linear function:

$$\text{Penalty}(g) = -g*d$$

- Affine gap penalty: A constant penalty is given for opening a new gap while a linear and smaller penalty is given for subsequent gap extensions. The cost function of the affine gap penalty is hence given by the following affine equation:

$$\text{Penalty}(g) = -d-(g-1)*e$$

BLAST stands for Basic Local Alignment Tool. It is developed on the ideas of FASTA.BLAST is used for searching both protein and DNA sequence databases for sequence similarities. It is a heuristic local alignment algorithm which approximates the dynamic programming Smith-Waterman algorithm. Since it is a heuristic algorithm, the local alignment it produces is not always optimal. However, it is much faster than the Smith-Waterman algorithm. As a result, BLAST and its variants are some of the most widely used sequence search tools.

The central idea of the BLAST algorithm is that a statistically significant alignment is likely to contain a high-scoring pair of aligned words. BLAST first finds these high scoring pairs of aligned words and then extends them to the real alignment. These words are k-residues long where k is different for DNA and protein sequences. The default k values for DNA and protein sequences are 11 and 3 respectively. There are 3 basic steps of BLAST:

- Pre-processing the query sequence: All k-long words in the query sequence are extracted. Then, words that are similar to these are found. We call the overall results the k-words.
- Scanning the subject sequences: All the subject sequences in the database are scanned one by one for matches with the obtained k-words.
- Extension of the matches: All matches in the subject sequences are extended to form local alignments between the query sequence and related subject sequences in the database.

In subsections II.A-II.C, all basic steps of the BLAST algorithm mentioned above will be explained in more detail.

It is worth mentioning at this stage that the aforementioned basic steps belong to the original BLAST algorithm. However, several variants of the original algorithm have been devised over the years with the aim of increasing its sensitivity while keeping run-times at minimum. All of these variants include the 3 basic steps of the original algorithm, with the addition of new steps. In this paper, we discuss two of these variants, namely BLAST with two-hit method, and Gapped BLAST, which are in subsections II.D and II.E respectively.

A. Step 1: Pre-processing the Query Sequence

An example protein sequence which has 9 residues (or amino acids) is shown below:

LVNRKPVVP

In this first step, we take the query sequence and chop it into overlapping k-words as illustrated below for the query sequence shown above, with k = 3:

Word 0: LVN

Word 1: VNR

Word 2: NRK

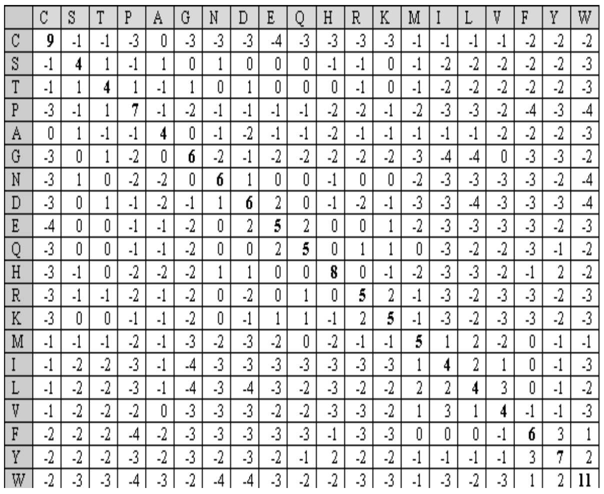
Word 3: RKP

Word 4: KPV

Word 5: PVV

Word 6: VVP

As it can be seen, there are 7 words extracted from the query sequence which are 3 residues long. In general, the number of words extracted equals (m-k) + 1 where m is the number of residues in the query sequence. After this, words similar to each of these extracted words are found through the usage of specific scoring matrix. An example scoring matrix for protein residues (Blosum50) is shown below in figure 2.



	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
C	**9**	-1	-1	-3	0	-3	-3	-3	-4	-3	-3	-3	-3	-1	-1	-1	-1	-2	-2	-2
S	-1	**4**	1	-1	1	0	1	0	0	0	-1	-1	0	-1	-2	-2	-2	-2	-2	-3
T	-1	1	**4**	1	-1	1	0	1	0	0	0	-1	0	-1	-2	-2	-2	-2	-2	-3
P	-3	-1	1	**7**	-1	-2	-1	-1	-1	-1	-2	-2	-1	-2	-3	-3	-2	-4	-3	-4
A	0	1	-1	-1	**4**	0	-1	-2	-1	-1	-2	-1	-1	-1	-1	-1	-2	-2	-2	-3
G	-3	0	1	-2	0	**6**	-2	-1	-2	-2	-2	-2	-2	-3	-4	-4	0	-3	-3	-2
N	-3	1	0	-2	-2	0	**6**	1	0	0	-1	0	0	-2	-3	-3	-3	-3	-2	-4
D	-3	0	1	-1	-2	-1	1	**6**	2	0	-1	-2	-1	-3	-3	-4	-3	-3	-3	-4
E	-4	0	0	-1	-1	-2	0	2	**5**	2	0	0	1	-2	-3	-3	-3	-3	-2	-3
Q	-3	0	0	-1	-1	-2	0	0	2	**5**	0	1	1	0	-3	-2	-2	-3	-1	-2
H	-3	-1	0	-2	-2	-2	1	1	0	0	**8**	0	-1	-2	-3	-3	-2	-1	2	-2
R	-3	-1	-1	-2	-1	-2	0	-2	0	1	0	**5**	2	-1	-3	-2	-3	-3	-2	-3
K	-3	0	0	-1	-1	-2	0	-1	1	1	-1	2	**5**	-1	-3	-2	-3	-3	-2	-3
M	-1	-1	-1	-2	-1	-3	-2	-3	-2	0	-2	-1	-1	**5**	1	2	-2	0	-1	-1
I	-1	-2	-2	-3	-1	-4	-3	-3	-3	-3	-3	-3	-3	1	**4**	2	1	0	-1	-3
L	-1	-2	-2	-3	-1	-4	-3	-4	-3	-2	-3	-2	-2	2	2	**4**	3	0	-1	-2
V	-1	-2	-2	-2	0	-3	-3	-3	-2	-2	-3	-3	-2	1	3	1	**4**	-1	-1	-3
F	-2	-2	-2	-4	-2	-3	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	**6**	3	1
Y	-2	-2	-2	-3	-2	-3	-2	-3	-2	-1	2	-2	-2	-1	-1	-1	-1	3	**7**	2
W	-2	-3	-3	-4	-3	-2	-4	-4	-3	-2	-2	-3	-3	-1	-3	-2	-3	1	2	**11**

Figure 2. The Blosum50 scoring matrix

Words which score at least threshold value T with the scoring matrix when aligned with the words extracted from the query sequence are regarded to be similar to these extracted words. Similar words for each extracted word are found and then recorded with the location address of the corresponding extracted word in the query sequence tagged to them. This process is illustrated below with the first extracted word shown above (i.e. LVN) using the Blosum50 scoring matrix for the case where T is 12:

```
        Word 0: L    V   N
                4 + 4 + 6 = 14
Query word 1: L    V   N

         Word 0: L    V    N
                2 + 4 + 6 = 12
Query word 2: M   V    N

         Word 0: L    V    N
                4 + 4 + 1 = 9
Query word 3: L   V    S
```

Query word 1 and query word 2 score 14 and 12 respectively when aligned with the first extracted word (LVN) from the query sequence. Since score values are over or equal to 12, query word 1 and query word 2 are recorded with the location address of the first extracted word in the query sequence, which is 0. However, query word 3 is discarded since it scores less than 12 when aligned with the extracted word. All recorded similar words are used in step 2 of the BLAST algorithm.

B. Step 2: Scanning the subject sequences

In this step, all subject sequences in the database are scanned one by one to find the possible exact matches of the query words which were recorded in step 1. Each match is referred to as hit or hotspot. Each hit is recorded in a list for the third step of the BLAST algorithm with the identity of the corresponding query word and the location address where the hit occurred in the subject sequence. Considering the fact that current databases contains tens of thousands of subject sequences and that each subject sequence comprises hundreds/thousands of residues, it is obvious that this sequence database scanning process is a massively time consuming task.

C. Step 3: Extension of the matches

In this last step of the basic BLAST algorithm, we utilize the list of matches (hits) obtained in step 2 to form local alignments between the query sequence and the subject sequences in the database. Each entry in the list of hits contains the location address of a match in the subject sequence and the location address of the corresponding query word in the query sequence. Starting from these 2 location addresses, each of the hits in the list is extended on the query and corresponding subject sequence in both directions without allowing any gaps. In this extension, pairs of residues along the query and subject sequence are scored with a scoring matrix (e.g. Blasoum50). This process is illustrated in figure 3 with the following subject sequence:

GVCRRPLKC

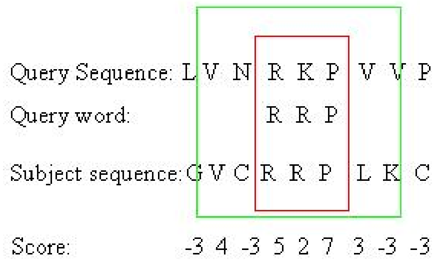

Query Sequence: L V N R K P V V P
Query word: R R P
Subject sequence: G V C R R P L K C
Score: -3 4 -3 5 2 7 3 -3 -3

Figure 3. Step 3: Extension of matches

In figure 3, the red box shows a hit where query word RRP is matched in the subject sequence. The query word RRP is similar to RKP word in the query sequence. The green box in figure 3 shows the extension which started from the edges of the red box. As the extension proceeds in a 1 residue pair at a time in both directions and without allowing for any gaps, pairs of residues along the extension are scored using a scoring matrix (BLOSUM50 in our case). These score terms are added up after each extension step and the extension is terminated when this total score falls a certain cut-off distance below the best total score obtained so far. Then, the extension goes back to its state which yielded the highest total score. As a result of this extension step, the related subject sequence is locally aligned to the query sequence (without gaps).

D. BLAST with two-hit method

The third step of the BLAST algorithm, i.e. the extension of the matches on the query and subject sequences, generally accounts for a very high percentage of the BLAST algorithm's execution time. Hence, the two-hit method was devised to reduce the time spent in this extension step. The central idea of the two-hit method is to start extension only when there are two non-overlapping hits on the same diagonal within distance A of each other. This is illustrated in figure 4 where only two non-overlapping hits on the same diagonal line which are close enough to each other are extended.

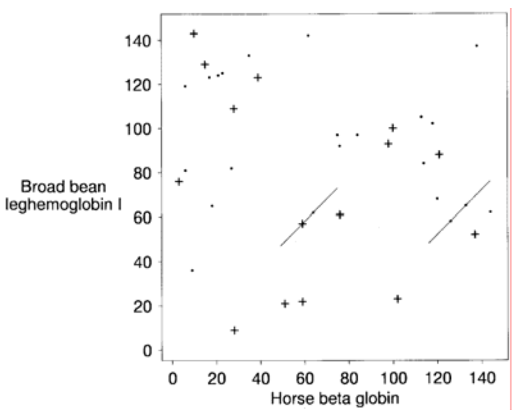


Figure 4. Ungapped extension of two close hits on the sane diagonal lines [10]

In other words, if the distance between any two non-overlapping hits on the subject sequence is equal to the distance between the locations of the corresponding query words in the query sequence, then ungapped extension is triggered in both directions starting from both hits. The rest of the process is the same as explained in subsection II.C and the result is a local ungapped alignment of the query and subject sequences. This process is illustrated in figure 5 where A is equal to 5.

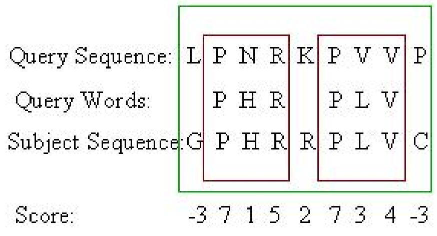

Query Sequence: L P N R K P V V P
Query Words: P H R P L V
Subject Sequence: G P H R R P L V C
Score: -3 7 1 5 2 7 3 4 -3

Figure 5. Extension with the two-hit method

In figure 5, the red boxes show two non-overlapping hits on the query and subject sequences within a distance of 4. Since the distance between the query words in the query sequence is equal to the distance between the two hits on the subject sequence, and since this distance between the two hits is less than 5, and bigger than 2, ungapped extension is started from the edges of the left and right hand sides of the red boxes respectively (see the green box in figure 5).

To maintain the sensitivity of the general algorithm, the threshold value T used in the query pre-processing step of the algorithm is reduced. Hence, the number of query words recorded in this step will increase. As a result, while scanning the subject sequences in step 2 we will potentially find more hits than before. However, only a small fraction of these hits will have an associated second hit. Therefore, ungapped extension will be triggered less frequently compared to the case in the original BLAST algorithm. The total execution time of BLAST is thus reduced.

E. Gapped BLAST

Gapped BLAST is an advancement of BLAST with the two-hit method, which is faster and gives better alignments and alignment scores. In addition to the steps outlined above, gapped alignment is triggered in gapped BLAST if local ungapped alignment obtained as a result of ungapped extension has a sufficiently high score. If this is the case, the central pair of the local ungapped alignment is used as a seed from which the gapped alignment is run both backwards and forwards, as illustrated in figure 6. The gapped alignment algorithm utilized in Gapped BLAST is a modified version of the Needleman-Wunsch algorithm where the alignment is pruned when alignment scores fall a certain cut-off distance below the best score so far. The Needleman-Wunsch algorithm with linear and affine gap models is explained in subsections II.F and II.G below, respectively. The necessary modifications of the original Needleman-Wunsch algorithm needed in the Gapped BLAST algorithm are explained in subsection II.H.

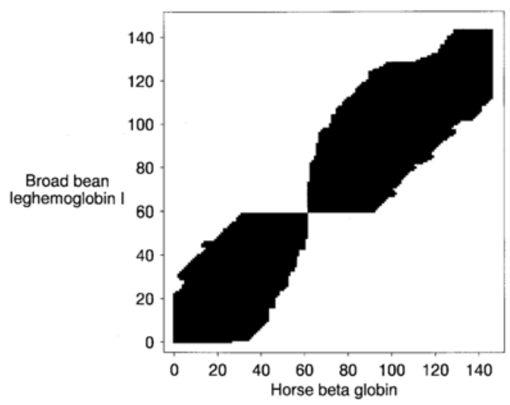


Figure 6. Gapped alignment started from the central pair of the local ungapped alignments in both directions [10]

F. The Needleman-Wunsch algorithm with the Linear Gap Model

The Needleman-Wunsch algorithm is a dynamic programming algorithm which finds optimal global gapped alignment between two sequences [6]. In Gapped BLAST, however, it is used for local alignment purposes, after a slight modification as will be explained in subsection II.H below. In this section, we will present the original Needleman-Wunsch algorithm where a linear gap model is assumed.

Assuming we have two sequences $X = x_1x_2\ldots x_M$ and $Y = y_1y_2\ldots y_N$, whose lengths are M and N respectively, a dynamic programming score matrix F is built where each cell F (i, j) represents the best alignment between $x_1x_2\ldots x_i$ segment of X and $y_1y_2\ldots y_j$ segment of Y.

The boundary cells of Matrix F are set by the following set of equations:

$$F(0, 0) = 0 \quad (1)$$

$$F(i, 0) = -i*d \text{ where } i=1, 2….M \quad (2)$$

$$F(0, j) = -j*d \text{ where } j=1, 2….N \quad (3)$$

The following equation is used to compute the values of each of the remaining cells of matrix F:

$$F(i,j) = max \begin{Bmatrix} F(i-1,j-1)+s(x_i,y_j) \\ F(i-1,j)-d \\ F(i,j-1)-d \end{Bmatrix} \quad (4)$$

Here, we aim to find best alignment between $x_1x_2….x_i$ and $y_1y_2…..y_j$ 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)), between $x_1x_2….x_{i-1}$ and $y_1y_2…..y_j$ (i.e. F(i-1, j)) and between $x_1x_2….x_i$ and $y_1y_2…..y_{j-1}$ (i.e. F(i, j-1)). There are three alternatives:

- An alignment between x_i and y_j: In this case, the new score F(i, j) is F(i-1, j-1) + s(x_i, y_j) where s(x_i, y_j) is the scoring matrix score for x_i and y_j.
- An alignment between x_i and a gap in Y: In this case, the new score F(i, j) is F (i-1, j)-d where d is the gap penalty.
- An alignment between a gap in X and y_j: In this case, the new score F(i, j) is F (i, j-1)-d where d is the gap penalty.

One of these three alternatives (see figure 7) yields the largest score and is the best alignment between $x_1x_2….x_i$ and $y_1y_2…..y_j$.

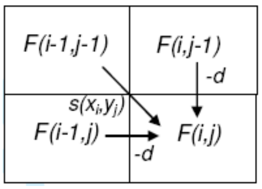
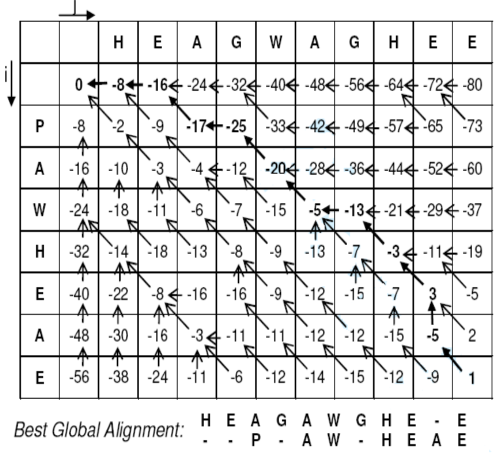


Figure 7. Illustration of the Needleman-Wunsch dynamic programming equations

Note that a pointer to the cell from which F (i, j) was derived (i.e. above, left, above-left) is stored in each cell. Once the value of the last cell of matrix F (i.e. F (M, N)) is computed, the best global alignment between X and Y is obtained by tracking back from this cell, using the aforementioned pointers, and applying the following procedure:

- If cell (i, j) was derived from cell (i-1, j-1), the pair of symbols x_i and y_j is added to the front of the current alignment.
- If cell (i, j) was derived from cell (i-1, j), x_i and a gap in Y are added to the front of the current alignment.
- If cell (i, j) was derived from cell (i, j-1), a gap in X and y_j are added to the front of the current alignment.

This is illustrated in figure 8 for 2 protein sequences. In this figure, the trace-back starts from F(M, N) = F(7, 10) and moves backward to the cell from which the current cell was derived until F(0, 0) is reached, while applying the aforementioned procedure at every step of the trace-back. The resulting global alignment of these 2 sequences can be seen at the bottom of figure 8.



		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

Figure 8. Illustration of the Needleman-Wunsch algorithm

G. *The Needleman-Wunsch algorithm with the Affine Gap Model*

The Needleman-Wunsch algorithm with the affine gap model is similar to the one with the linear gap model. However, in this case, we have three new matrixes namely I_z, I_x and I_Y to compute. The following equations are used to compute the values of I_z, I_x and I_Y where d is the penalty associated with the gap opening and e is penalty associated with the gap extension:

$$I_z(i,j) = max \begin{Bmatrix} I_z(i-1,j-1)+s(x_i,y_j) \\ I_x(i-1,j-1)+s(x_i,y_j) \\ I_y(i-1,j-1)+s(x_i,y_j) \end{Bmatrix} \quad (5)$$

$$I_x(i,j) = max \begin{Bmatrix} I_z(i-1,j)-d \\ I_x(i-1,j)-e \end{Bmatrix} \quad (6)$$

$$I_y(i,j) = max \begin{Bmatrix} I_z(i,j-1)-d \\ I_y(i,j-1)-e \end{Bmatrix} \quad (7)$$

The values of the dynamic programming matrix cells F(i, j) are equal to the maximum of $I_Z (i, j)$, $I_X (i, j)$ and $I_Y (i, j)$ as shown in equation 8.

$$F(i,j) = max \begin{Bmatrix} I_z(i,j) \\ I_x(i,j) \\ I_y(i,j) \end{Bmatrix} \quad (8)$$

Note that the pointer to the above-left cell is stored in the cell if *F(i, j)* is set to equal I_Z *(i, j)* whereas the pointer to the left cell is stored if *F(i, j)* is set to equal

$I_x(i, j)$. Finally, the pointer to the above cell is stored if $F(i, j)$ is set to equal $I_Y(i, j)$.

H. Modified Needleman-Wunsch algorithm

The Needleman-Wunsch algorithm presented above is used for finding global gapped alignments between two sequences. Gapped BLAST however requires some modifications to the original Needleman-Wunsch algorithm. First, no computations are done for the dynamic programming matrix cells which are adjacent to cells whose F(i, j) values are a certain cut-off value below the highest cell value computed so far. Second, the trace-back procedure may start at any cell which has the highest value F(i, j) among all the cells, rather than bottom rightmost cell. In this way, we have a local gapped alignment at the end of the trace-back procedure.

III. Hardware Implementation of Gapped BLAST with the Two-Hit Method

Figure 9 shows a hardware architecture which implements gapped BLAST algorithm with the two-hit method. Each block in the architecture implements one step of the algorithm as described in the above sections, except for the pre-processing query sequence step which is implemented by high level application software running on the host computer. The architecture consists of 8 *HitFinderTwoHit* blocks, 2 *UngappedExtender* blocks and 1 *GappedExtender block* all of which are running in parallel. There are also 8 32K x 5 bits subject sequence memories each of which holds a number of subject sequences. Note that each subject sequence memory belongs to one *HitFinderTwoHit* block. Each *HitFinderTwoHit* block is composed of 5 *HitFinder* blocks and 1 *TwoHitMethod* block. Each *HitFinder* block implements step 2 outlined in subsection II.B and scans its assigned subject sequence memory to find exact matches of the query words in the subject sequences. Each *TwoHitMethod* block performs the two-hit method procedure on hits coming from the 5 *HitFinder* blocks which are in the same *HitFinderTwoHit* block as the *TwoHitMethod* block. Besides these, each *UngappedExtender* block implements step 3 mentioned in subsection II.C and extends the two hits found by its 4 allocated *TwoHitMethod* blocks without allowing gaps, in order to obtain local ungapped alignments. Finally, a single *GappedExtender* block implements the modified Needleman-Wunsch algorithm to produce local gapped alignments from local ungapped alignments obtained in 2 *UngappedExtender* blocks.
The high level application software and all of the blocks which constitute the architecture shown in figure 9 are detailed in the following subsections.

A. High Level Application Software

Figure 10 shows the organization of our Gapped BLAST FPGA implementation. Application software running on the host has many duties, the most important of which is the query sequence pre-processing as explained in section II.A. In brief, the application software finds 3 letter long query words which score at least a threshold value T when aligned with words extracted from the query sequence. Then, the location address of each of these query words in the query sequence is placed at a vacant position in an upper word list and a lower word list pair depending on the 2 most significant letters and 2 least significant letters of the query word, respectively. Note that there are 5 upper word and lower word list pairs.

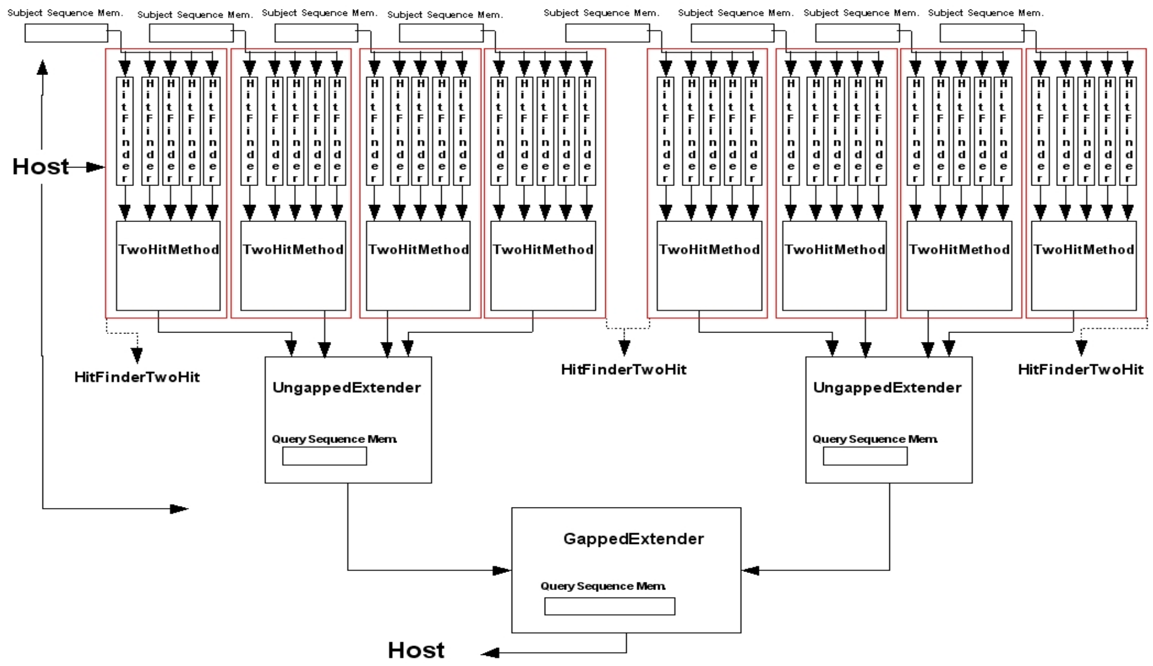


Figure 9. Hardware architecture for the Gapped BLAST algorithm with the two-hit method

As it can be seen in figure 10, there are various FPGA configuration bit files for different threshold and cut-off value parameters. The first task of the application software is to pick the proper bit file, depending on the user-supplied algorithm parameters, from a database of FPGA configurations and load it on to the FPGA chip. Afterwards, the application software runs the hardware configuration in 4 modes. In mode 1, the application software sends one of the 5 upper word and lower word list pairs to each of the 5 *HitFinder* blocks in every *HitFinderTwoHit* block. In mode 2, a number of subject sequences are sent to the 8 available subject sequence memories on FPGA, depending on the subject sequence lengths. In mode 3, the application software sends a query sequence to the FPGA to be stored in memories within the 2 *UngappedExtender* blocks and the single *GappedExtender* block. Finally, the execution of the hardware configuration is launched in mode 4. After some time, the FPGA starts sending the high scoring subject sequences to host with their alignment scores to be printed onto the screen. By repeating these steps several times for different subject sequences, we can align a query sequence to all subject sequences in a sequence database. Note that each iteration is called as "pass".

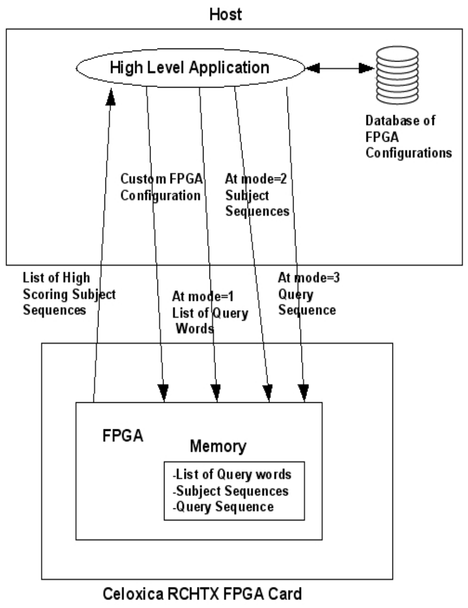

Host
High Level Application
Database of FPGA Configurations
Custom FPGA Configuration
At mode=2 Subject Sequences
List of High Scoring Subject Sequences
At mode=1 List of Query Words
At mode=3 Query Sequence
FPGA
Memory
-List of Query words
-Subject Sequences
-Query Sequence
Celoxica RCHTX FPGA Card

Figure 10. Organization of our Gapped BLAST system

B.

A.

B. HitFinder Block

Figure 11 shows a simplified inner structure of a *Hitfinder* block. The architecture of this block is modified version of one shown in figure 7 of [18].The difference is that as opposed to [18],we added the positions of the query words in the query sequence into the memory content of the Hit Finder to increase the sensitivity of the hit finding process. Furthermore, our design implements the two-hit method (detailed in the next section) which is not the case with [18]. Lastly, our core includes a unit for gapped alignment for the purposes of implementing Gapped BLAST in contrast to [18] which just implements original BLAST.

The major aim of this block is to scan each three letter long word of the subject sequences in order to find exact matches of the query words, as explained in subsection II.B. It is comprised of an upper word list memory, a lower word list memory, a shift register, a FIFO buffer and some control logic. Note that every *Hitfinder* block is assigned to a subject sequence memory whose address register (*Counter*) is unique in the *HitFinderTwoHit* block.

At every clock cycle, 5-bit long residues of a subject sequence are shifted into the shift register (*ShiftReg*) from the assigned subject sequence memory and the address register of the subject sequence memory is incremented by one. The shift register is 15 bits long and hence it can hold 3 subject sequence residues at the same time. At every clock cycle, the 10 most significant bits and the 10 least significant bits of the shift register content are used as addresses for the upper word list memory and the lower word list memory respectively (see figure 11). If the resulting outputs of these memories are valid entries and are equal to each other, this means that a three-letter long word of the subject sequence which is currently held in the shift register matches exactly a query word whose location address in query sequence is given in the outputs of the word list memories. In this case, we have a hit condition which needs to be recorded for the following steps of the algorithm. Hence, we register the address of the query word in the query sequence and the location address of the hit in the subject sequence to a FIFO buffer named *Hit FIFO* with 3 control bits. These entries to *Hit FIFO* are processed by the *TwoHitMethod* block assigned to the *Hitfinder* block (see figure 9).

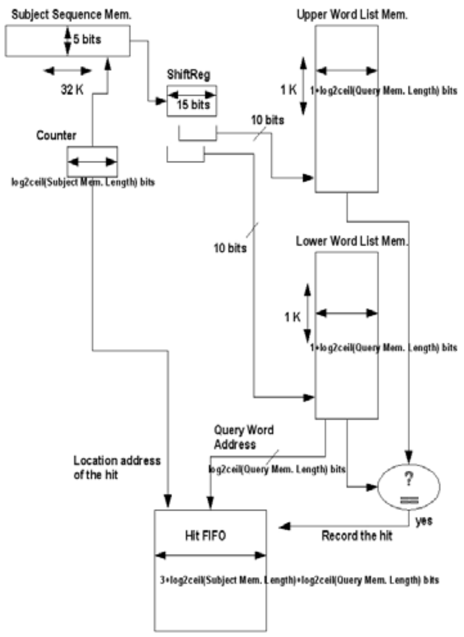

Subject Sequence Mem.
5 bits
32 K
ShiftReg
15 bits
Upper Word List Mem.
1 K
1+log2ceil(Query Mem. Length) bits
10 bits
Counter
log2ceil(Subject Mem. Length) bits
10 bits
Lower Word List Mem.
1 K
1+log2ceil(Query Mem. Length) bits
Query Word Address
log2ceil(Query Mem. Length) bits
Location address of the hit
?
==
yes
Record the hit
Hit FIFO
3+log2ceil(Subject Mem. Length)+log2ceil(Query Mem. Length) bits

Figure 11. Simplified inner structure of the *Hitfinder* block

C. TwoHitMethod Block

Figure 12 shows a simplified inner structure of the *TwoHitMethod* block. Its aim is to find two non-overlapping hits on the same diagonal within distance A of each other as explained in subsection II.D above. In this architecture, there are two FIFOs of the same length and same width namely *Hit FIFO 1* and *Hit FIFO 2* to which the same hit entries from the *Hit FIFO*s of the 5 *Hitfinder* blocks (which belong to the same *HitFinderTwoHit* block) are stored one by one in turn starting from the *Hit FIFO* in the first *Hitfinder* block. The processing of hit entries commences when there are more than two hit entries in the FIFOs. For instance, the a^{th} hit entry of *Hit FIFO 1* and b^{th} hit entry of *Hit FIFO 2* are taken and the hit addresses of these entries are subtracted from each other. If the result is less than 3, we continue with the processing of the a^{th} hit entry in *Hit FIFO 1* and $(b+1)^{th}$ hit entry in *Hit FIFO 2* in the next clock cycle. On the other hand, if the result is bigger than threshold value A, we continue with the processing of the $(a+1)^{th}$ hit entry in *Hit FIFO 1* and $(a+2)^{th}$ hit entry in *Hit FIFO 2* in the next clock cycle. However, if the result of this subtraction is between 3 and threshold value A inclusive, we subtract the query word addresses in the hit entries. If the second subtraction result is not equal to the first one, this means that the two hits are not on the same diagonal, and hence we continue with the processing of the a^{th} hit entry in *Hit FIFO 1* and $(b+1)^{th}$ hit entry in *Hit FIFO 2* in the next clock cycle. If the two results are the same, however, this means that we have two close enough non-overlapping hits on the same diagonal which need to be recorded for the subsequent steps of the algorithm. The two hit cases are recorded to two FIFOs namely *TwoHit FIFO1* and *TwoHit FIFO* 2. The address of the first hit and the distance between the two hits (Result 2 in figure 12) are stored in *TwoHit FIFO1* with 2 control bits, whereas the address of the first query word is stored in *TwoHit FIFO* 2. These two-hit entries to the *TwoHit FIFOs* are subsequently processed by the assigned *UngappedExtender* block.

D. UngappedExtender Block

The *UngappedExtender* block implements the ungapped extension step of the Gapped BLAST algorithm as explained in subsection II.C above. Each of the two *UngappedExtender* blocks read *Twohit FIFO*s of its 4 assigned *TwoHitMethod* blocks in turn. When the *UngappedExtender* block detects a two-hit entry in the *Twohit FIFOs* of one *TwoHitMethod* block, the hit address of the first hit, the address of the first query word in the query sequence and the distance between the two hits are all extracted from that entry to compute the start (seed) points of the outward ungapped extension in both directions, on both query and related subject sequence. Note that first residue pair of the first hit and the last residue pair of the second hit are the seed points of the outward ungapped extension on the query and related subject sequence,. Afterwards, the inward ungapped extension starts from one start point to the other start point where the residue pairs along the extension are scored against a scoring matrix, with the intermediate scores accumulated. When the inward ungapped extension ends, the outward ungapped extension is launched in both directions. Here again, the residue pairs along the extension are scored, with the intermediate score terms accumulated, and added up with the total score obtained from the inward ungapped extension. The outward ungapped extension terminates either when the currently computed grand total score falls a certain cut-off value below the highest grand total score obtained so far, or when the extension reaches end of the query or subject sequences in either direction. In this case, the ungapped extension retracts to its previous state which yielded the highest grand total score. If this highest grand total score exceeds a certain threshold value, the end points of this high scoring ungapped extension in both directions on both query and subject sequences are registered to two *UngappedResult FIFOs* to be read and processed by the single *GappedExtender* block for the purpose of gapped alignment.

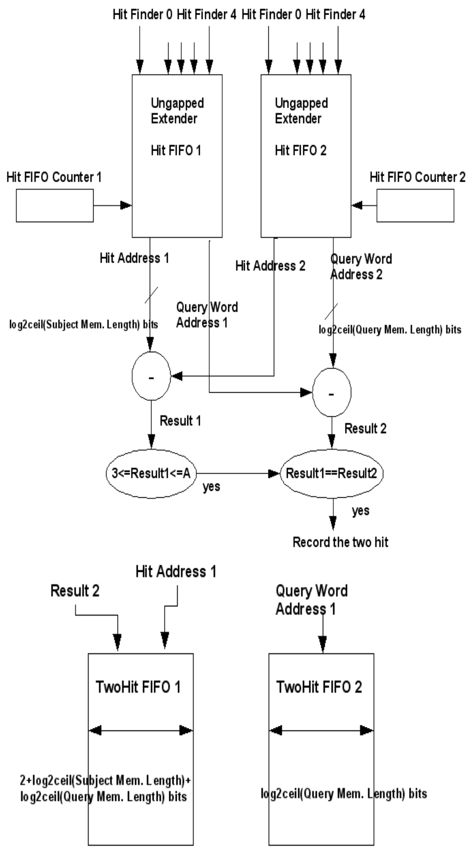


Figure 12. Simplified inner structure of TwoHitMethod block

E. GappedExtender block

The *GappedExtender* block implements the gapped alignment step using the modified Needleman-Wunsch algorithm with the affine gap model. Here, only the gapped alignment score is computed. The final alignment, i.e. with trace-back, is not done on FPGA because of its excessive memory requirement. The *GappedExtender* block reads *UngappedResult FIFOs* of the two *UngappedExtender* blocks in turn to obtain the edge points of the high scoring ungapped alignments produced by these two *UngappedExtender* blocks.

These edge points are used to compute the central residue pair of the ungapped alignment from which the gapped alignment on the query and related subject sequence is launched in both directions. Figure 13 shows one of the two linear systolic arrays in the *GappedExtender* block which run independently in parallel to perform the modified Needleman-Wunsch algorithm on each side of the seed residue pair. This architecture is deducted from the data dependency graph of the Needleman-Wunsch algorithm as presented in section II above [12].

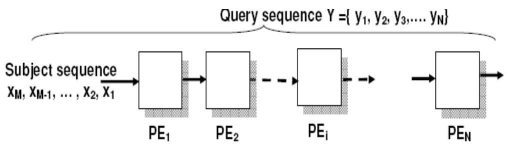


Figure 13. Linear systolic array for gapped alignment

The linear systolic array consists of pipelined basic processing elements (PE) each of which performs the dynamic programming equations presented in subsections II.G above. Before the operation of the array, the query sequence residues at one side of the seed residue pair are shifted through the array. At the end of this shift, each PE holds one query residue. Following this, the subject sequence residues at the same side of the seed residue pair are shifted systolically through the array during which each PE generates value of one dynamic programming matrix cell every clock cycle. However, the direction of the cell from which the current result has been derived is not saved since trace-back will not be performed in hardware. Each PE generates one column of the dynamic programming matrix after M cycles where M is equal to the number of subject sequence residues. However, each PE is one cycle behind its predecessor PE due to the fact that computations in PE_{i+1} depend on the computation results in PE_i. Figure 14 illustrates the execution of the recursive equations of the original Needleman-Wunsch algorithm on the linear array architecture where diagonal lines cross the matrix cells of dynamic programming matrix whose values are computed at the t^{th} clock cycle.

The linear array architecture keeps record of the maximum value in the dynamic programming matrix at each PE, calculating its *maximum-so-far* value and broadcasting it to the next PE. The gapped extension in the linear array architecture terminates when the end of the query or subject sequence is reached in either side, or when the current result in PE_1 is a certain cut-off value below its *maximum-so-far*. Once both of the linear array architectures in the *GappedExtender* block terminate, their maximum values are added up to obtain the score of the gapped alignment. If this score exceeds certain threshold value, the address of the subject sequence in the related subject sequence memory is sent to the host to allow for the subject sequence to be truly aligned with the query sequence by the high level application software running on the host.

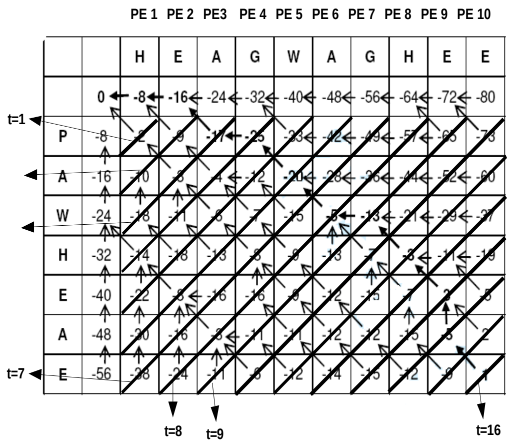


Figure 14. Illustration of the execution of the original Needleman-Wunsch algorithm on the linear systolic array architecture

Note that number of PEs in the linear array architectures should be equal to the number of residues in the query sequence in order to correctly implement the modified Needleman-Wunsch algorithm. However, considering the amount of resources in today's FPGAs, this is impossible since there could be hundreds or even thousands of residues in the query sequence. To solve this problem, the algorithm is partitioned into small alignment steps which are mapped onto a fixed size linear systolic array as shown in figure 15 [14] [15]. In this architecture, the alignment process is performed in a number of passes depending on the length of the query sequence, where a FIFO is used to store intermediate results and subject sequence residues from each pass before they are fed back to the input of the array for the next pass. In our implementation, each of the linear arrays in the *GappedExtender* block has 4 processing elements. This could be extended at will, resource permitting.

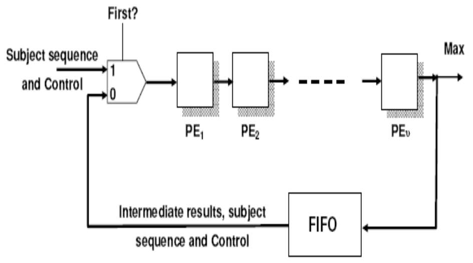


Figure 15. Partitioning and mapping of the modified Needleman-Wunsch algorithm on a fixed size systolic array

IV. Results

Our Gapped BLAST design was captured in the Handel C language to which no specific resource inference or placement constraints were applied. Hence, it can be directly targeted to a variety of FPGA platforms (e.g. Xilinx and Altera FPGAs). The resulting core was compiled into EDIF by Agility's DK5 SP2 suite from which FPGA bitstreams were generated using Xilinx ISE9.2 tool.

The hardware implementation of the core was achieved on a Celoxica RCHTX FPGA board [17] which has a Xilinx Virtex 4 (xc4vlx160ff1148-11) FPGA and off-chip memory fitted on it. In our implementation, however, the off-chip memory was not used. The operation of the core was tested on the Swiss-Prot protein sequence database [16] with various query protein sequences.

We have also implemented Gapped BLAST with the two-hit method algorithm in C in order compare our hardware implementation with a pure software implementation. Table 1 presents timing performance figures of both hardware and software implementations for 9 random query protein sequences of various lengths searched in the Swiss-Prot database. The FPGA hardware was clocked at 15 MHz. The software implementation was executed on an Intel Centrino Duo 2.2 GHz PC with 2 GB RAM. The same threshold and cut-off values were used in both hardware and software implementations at every step of the algorithm.

As it can be seen from table 1, our FPGA implementation result in substantial speed-up compared to software, ranging from 44x to 20x (the speed-up figure depends on the query sequence). The reason behind this high speed-up figure of the FPGA implementation, despite the huge difference in clock frequency, is due to the high level of process parallelism on FPGA.

Table 1. Timing performance figures of hardware and software implementations for 9 random protein sequences queried in Swiss-Prot protein sequence database

	No of Residues in Query Sequence	No of Query words	FPGA Execution time (sec)	Software execution time (sec)	FPGA Speed-up
1. Query Sequence	111	116	4.45	91.56	20.58
2. Query Sequence	214	98	5.01	131.93	26.34
3. Query Sequence	368	136	4.32	137.42	31.81
4. Query Sequence	459	263	5.88	211.42	35.96
5. Query Sequence	565	137	5.73	181.48	31.67
6. Query Sequence	635	140	5.36	194.45	36.28
7. Query Sequence	746	117	6.83	233.25	34.15
8. Query Sequence	864	240	7.01	311.23	44.40
9. Query Sequence	985	53	5.33	194.12	36.42

V. Conclusion

In this paper, the detailed FPGA implementation of the Gapped BLAST with two-hit method algorithm has been presented. To our knowledge this is the first FPGA implementation of this algorithm ever reported in the literature. The hardware architecture is composed of various blocks each of which performs a specific step of the algorithm in parallel. Moreover, the FPGA core is parameterized in terms of the sequence lengths, match score, gap penalties, cut-off and threshold values. The resulting implementation outperforms an equivalent desktop-based software implementation by at least one order-of magnitude. Furthermore, it was designed in the Handel-C language which makes it FPGA-platform-independent. As a result, the same core can be ported to other FPGA architectures from different vendors.

The work presented in this paper is part of a bigger project which seeks to harness the computational performance and re-configurability features of FPGAs in the field of Bioinformatics and computational biology. Future work includes the extension of this work to the Position Specific Iterated BLAST (PSI-BLAST) algorithm, as well as other sequence analysis techniques based on Hidden Markov Models.

VI. 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] 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.

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

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

[6] 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

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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] The Handel-C Language Reference Manual, Agility Plc, http://www.agilityds.com

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

[15] 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

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

[17] RCHTX FPGA Board Reference Manual, Celoxica Plc, http://www.celoxica.com

[18] Sotiriades, E.,Dollas,’A General Reconfigurable Architecture for the BLAST Algorithm’, Journal of VLSI Signal Processing 48, 189–208, 2007