High Performance FPGA-based Core for BLAST Sequence Alignment with the Two-Hit Method

Server Kasap, Khaled Benkrid, *Senior Member, IEEE,* Ying Liu

***Abstract*--This paper presents the design and implementation of a high performance FPGA-based core for BLAST sequence alignment with the two-hit method. BLAST with two-hit is a very widely used heuristic biological sequence alignment algorithm, and this paper is the first reported FPGA implementation of it, to our knowledge. The architecture of our core is parameterized in terms of the 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 with the others. This results in a high performance and efficient FPGA implementation, which outperforms equivalent software implementations by one order of magnitude or more. Real hardware implementations show that our core is 52 times faster than equivalent software implementations, on average. 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 same code can be easily ported to different FPGA families and architectures.**

I. Introduction

In Bioinformatics and Computational biology (BCB), biological sequence alignment is a very common task where subject sequences from a large database are aligned to a query sequence to find similarities between the query sequence and the sequences in the database [1]. Obtaining information about a newly discovered biological sequence (i.e. Protein, DNA or RNA) from other known sequences is a major application of this operation. For example, if a new sequence is found to be similar to a known cancerous sequence, then information regarding the functionality of the new sequence can be deduced. This is obviously useful in early disease diagnosis and drug engineering. Furthermore, biological sequence alignment can be utilized in the study of evolutionary development and history of species [1] [2].

Sequence alignment is a computationally intensive operation, however. This is exacerbated by the exponential growth in biological sequence databases. Therefore, desktop computer systems cannot, usually, perform this task within acceptable execution timeframes. Hence, faster computing platforms are required.

Manuscript received July 5, 2008.

Server Kasap is with the University of Edinburgh, Mayfield Road, Edinburgh EH9 3JL, Scotland, UK. He is a PhD student in School of Electronics and Engineering (E-mail: s.kasap@ed.ac.uk).

Khaled Benkrid is with the University of Edinburgh, Mayfield Road, Edinburgh EH9 3JL, Scotland, UK. He is a lecturer in the School of Engineering and Electronics (e-mail: k.benkrid@ed.ac.uk).

Ying Liu is with the University of Edinburgh, Mayfield Road, Edinburgh EH9 3JL, Scotland, UK. She is a PhD student in School of Electronics and Engineering (E-mail: y.liu@ed.ac.uk).

Recently, high performance reconfigurable hardware in the form of Field Programmable Gate Arrays (FPGAs) has been proposed as an efficacious and efficient implementation platform for sequence alignment algorithms [3] [4] [5]. Indeed, their ASIC-like performance coupled with their reprogrammability feature make FPGAs capable of providing high speed-ups compared to general purpose processors, with the added convenience of reprogrammability.

There are various biological sequence alignment algorithms in the literature. Some of these are exhaustive and give optimal alignments (e.g. Needleman-Wunsch [6], Smith-Waterman [7]) and others 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 heuristic local alignment algorithm. It is much faster than ordinary exhaustive dynamic programming algorithms, although it produces local alignments which are not always optimal. The design and implementation of BLAST with two-hit method [10] (a variant of BLAST) is presented in this paper. To our knowledge, this is the first reported FPGA implementation of this algorithm variant. It results in 52x speed-up over equivalent software implementations on average. Besides, the design was captured in a FPGA-platform-independent language, namely Handel-C language [13], which makes it portable across a number of FPGA architectures (e.g. from Xilinx or Altera).

In the remainder of this paper, essential background information on the general BLAST algorithm will be presented first. Following that, the design and implementation of our FPGA core for BLAST with the two-hit method will be elaborated. After that, timing performance of our core implementation is presented and compared with equivalent software implementation running on desktop computers. 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:

- 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. It 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 one of these variants, namely BLAST with two-hit method which is described in subsection II.D.

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.

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 two 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. BLOSUM50). This process is illustrated in figure 1 with the following subject sequence:

GVCRRPLKC

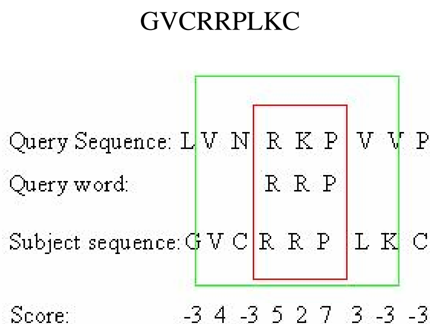

Figure 1. Step 3: Extension of matches

In figure 1, the small 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 big box in figure 1 shows the extension which started from the edges of the small 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 the extension only when there are two non-overlapping hits on the same diagonal within distance A of each other. This is illustrated in figure 2 where only two non-overlapping hits on the same diagonal line which are close enough to each other are extended.

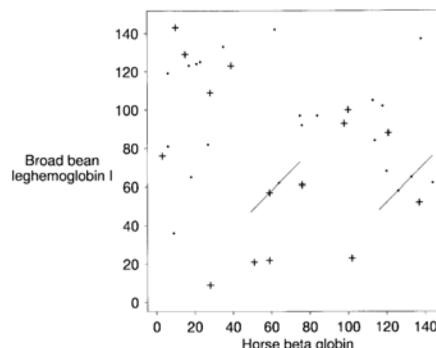

Figure 2. Ungapped extension of two close hits on the same diagonal lines [10]

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 3 where A is equal to 5.

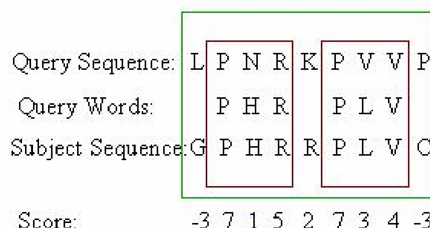

Figure 3. Extension with the two-hit method

In figure 3, the small 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 small boxes respectively (see the big box in figure 3).

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.

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

Figure 4 shows our hardware architecture which implements the 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 a host computer. The architecture consists of 12 *HitFinderTwoHit* blocks, 3 *UngappedExtender* blocks and 1 *Collector* block all of which are running in parallel. There are also 12 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 of which 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 four allocated *TwoHitMethod* blocks without allowing gaps, in order to obtain local ungapped alignments. Finally, a single *Collector* block collects high-scoring local ungapped alignments obtained in 3 *UngappedExtender* blocks and sends their details to the host.

The high level application software and all of the blocks which constitute the architecture shown in figure 4 are detailed in the following subsections.

A. High Level Application Software

Figure 5 shows the organization of our FPGA implementation for BLAST with two-hit method. There is application software running on the host computer which has many duties, the most important of which is the query sequence pre-processing as explained in section II.A. Besides running application software, host computer stores sequence database (e.g. Swiss-Prot) which is read as required by application software. 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.

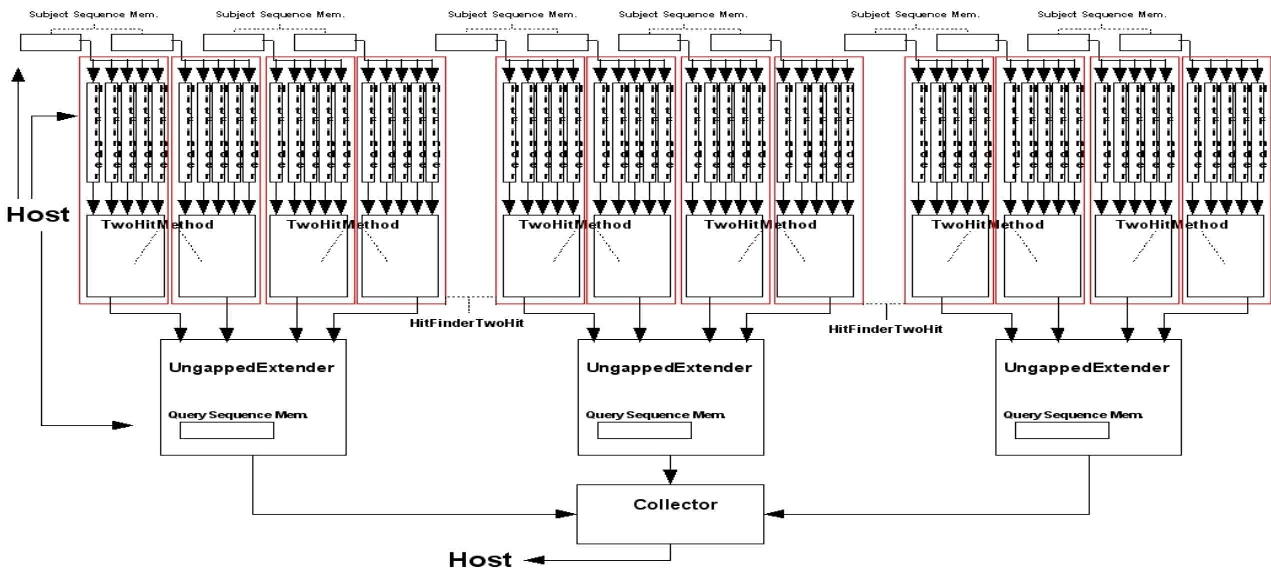

Figure 4. Hardware architecture for the BLAST algorithm with the two-hit method

As it can be seen in figure 5, 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 implementation in 4 modes. In mode 1, the application software sends each 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 read from the database on host are sent to the 12 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 3 *UngappedExtender* blocks. Finally, the execution of the hardware implementation is launched in mode 4.

After some time, the FPGA starts sending the high scoring subject sequences with their alignment scores. Then, application software prints these ungapped alignments onto the screen. This completes the first iteration of the operation. In the following iterations, different set of subject sequences are sent to the FPGA to be processed. Iterations terminate when there is no more subject sequence in the database awaiting to be sent to the FPGA.

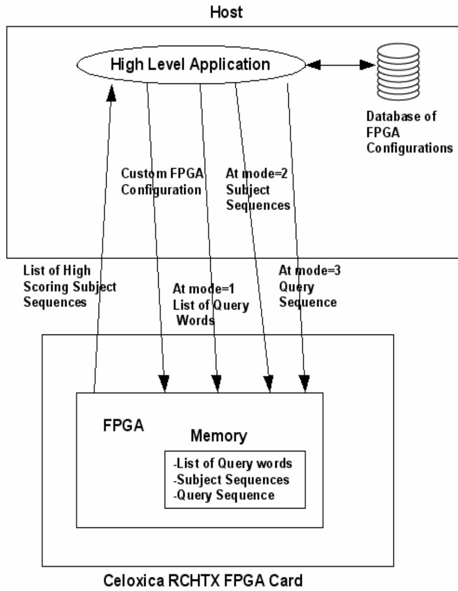

Figure 5. Organization of our BLAST system

B. HitFinder Block

Figure 6 shows a simplified inner structure of a *Hitfinder* block. The architecture of this block is a modified version of the one shown in figure 7 of [18]. 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 6). 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 4).

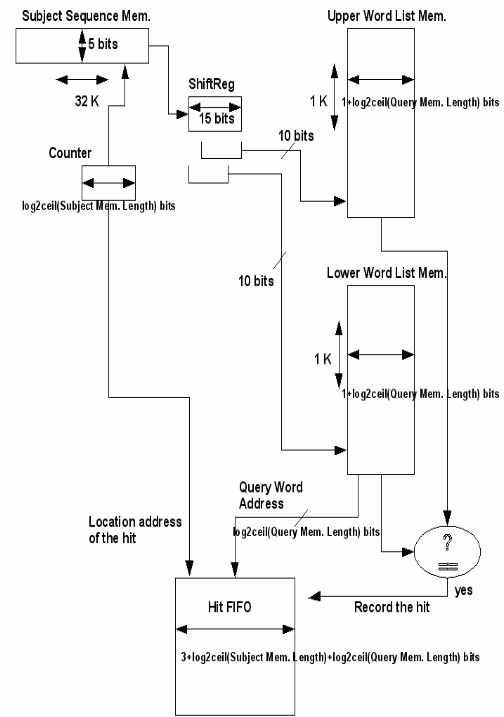

Figure 6. Simplified inner structure of the *Hitfinder* block

C. TwoHitMethod Block

Figure 7 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 ath 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 ath 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 ath 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 7) 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.

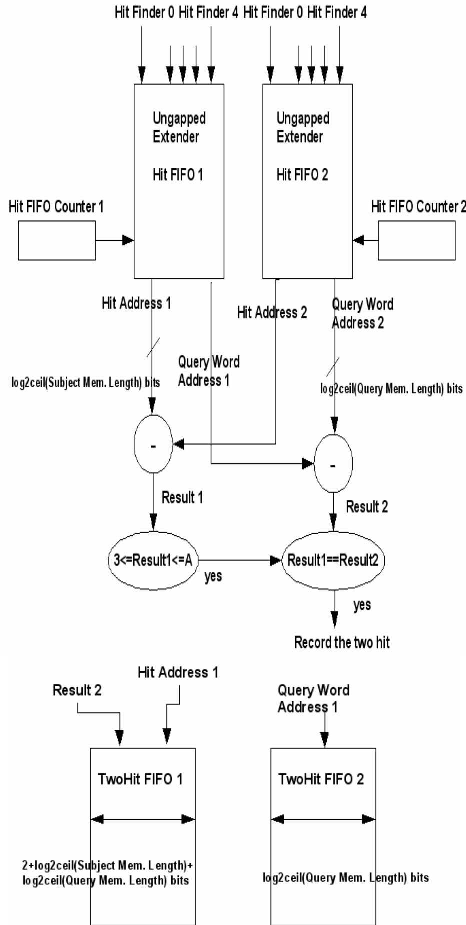

Figure 7. Simplified inner structure of TwoHitMethod block

D. UngappedExtender Block

The *UngappedExtender* block implements the ungapped extension step of the BLAST algorithm as explained in subsection II.C above. Each of the three *UngappedExtender* blocks reads *TwoHit FIFO*s of its four 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 the 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* with the score to be read by the single *Collector* block which sends these points as well as the score of the ungapped extension to the host.

IV. Results

Our 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 or 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 chip 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 BLAST with the two-hit method algorithm in C in order to compare our hardware implementation with a pure software implementation. Table 1 presents timing performance figures of both hardware and software implementations for 8 random query protein sequences of various lengths searched in the Swiss-Prot database. The FPGA hardware was clocked at 20 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.

Table 1. Timing performance figures of hardware and software implementations for 8 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	3.49	78.85	22.59
2. Query Sequence	368	136	3.50	137.98	39.42
3. Query Sequence	459	263	3.52	209.84	59.61
4. Query Sequence	565	137	3.45	177.57	51.47

5. Query Sequence	635	140	3.46	179.45	51.86
6. Query Sequence	746	117	3.57	209.25	58.61
7. Query Sequence	864	240	3.52	286.47	81.38
8. Query Sequence	985	53	3.48	197.87	56.86

As it can be seen from table 1, our FPGA implementation results in substantial speed-up compared to software, ranging from 81x to 22x (the speed-up figure depends on the query sequence). Note that the FPGA execution times fluctuate around 3.5 seconds hence showing experimentally that it is predominantly dependent on the size of the database rather than on the size of the query sequence or number of query words.

On average, our core is 52 times faster than equivalent software implementations. 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. Besides, the complete design, implementation and testing was achieved in less than 5 months by a first year PhD student. This shows that reconfigurable technology can be an efficacious and efficient platform for high performance biological sequence analysis.

V. Conclusion

In this paper, the detailed FPGA implementation of the BLAST algorithm with two-hit method has been presented. This is the first FPGA implementation of this variant of BLAST ever reported in the literature, to our knowledge. 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 equivalent desktop-based software by 52 times on average. Furthermore, our core was designed in the Handel-C language, thus making it FPGA-platform-independent. This means that our core can be ported to other FPGA architectures from different vendors very easily. Finally, it is worth mentioning that the whole design, implementation and testing design took less than 5 person-months to achieve, which shows that FPGAs can be an economic platform for high performance biological sequence alignment.

The work presented in this paper is part of a bigger project where the computational performance and re-configurability features of FPGAs are harnessed in the field of bioinformatics and computational biology. Future work includes the extension of this core to support the Gapped BLAST and Position Specific Iterated BLAST (PSI-BLAST) algorithms.

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 PCI 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