Chapter 8
Reconfigurable Hardware for DNA Matching*

Abstract. DNA sequence matching is used in the identification of a relationship between a fragment of DNA and its owner by mean of a database of DNA registers. A DNA fragment could be a hair sample left at a crime scene by a suspect or provided by a person for a paternity exam. The process of aligning and matching DNA sequences is a computationally demanding process. In this chapter, we propose a novel parallel hardware architecture for DNA matching based on the steps of the BLAST algorithm. The design is scalable so that its structure can be adjusted depending on size of the subject and query DNA sequences. Moreover, the number of units used to perform in parallel can also be scaled depending some characteristics of the algorithm. The design was synthesized and programmed into FPGA. The trade-off between cost and performance were analyzed to evaluate different design configuration.

8.1 Introduction

Bioinformatics is a field of biological science which deals with the study of methods for storing, retrieving and analyzing biological data such as DNA. It also involves finding the genes in the DNA sequences of various organisms, developing methods to predict the structure and/or function of newly discovered proteins and structural RNA sequences, clustering protein sequences into related families. Specifically, it includes solving the problem of aligning similar proteins in general and DNA in particular 10.

One of the main challenges in bioinformatics consists of aligning DNA. DNA stripes are long sequences of DNA bases, which are represented as A (Adenine), C (Cytosine), G (Guanine) and T (Thymine). In this sense, algorithms are specifically developed to reduce time spent in DNA alignment and matching, evaluating similarity degree between the subject and the query sequence. These algorithms are usually based on dynamic programming, which work well providing a fair tradeoff

* This chapter was developed in collaboration with Edgar José Garcia Neto Segundo.

between time and cost for short sequences. However, commonly these algorithm take exponentially more time as DNA sequences get longer.

The major advantage of the methods based on dynamic programming are the commitment to discover the best match. However, that commitment requires huge computational resources [7, 4]. DNA matching algorithms based on heuristics [8] emerged as an alternative to dynamic programming in order to reduce the required high computational cost. Instead of aiming at the best alignment(s), heuristics-based methods attempt to find a set of acceptable or pseudo-optimal matches. Ignoring unlikely alignments, these techniques have improved the performance of DNA matching [3, 5, 10]. Among heuristics-based methods, BLAST [1, 2] and FASTA [9, 7] stand out. Both of these algorithms have well defined procedures for the three main stages of aligning algorithms, which are seeding, extending and evaluating. BLAST is the fastest algorithm known so far [1, 2, 6]. In this chapter, we focus of this algorithm and propose a massively parallel architecture suited as an ASIC for DNA matching using BLAST. The main objective of this work is the acceleration of the aligning and matching procedures.

This chapter is organized as follows: First, in Section 8.2, we sketch briefly the steps used in the BLAST algorithm; Thereafter, in Section 9.2, we detail the proposed parallel architecture, pointing out specifically its scalability characteristics; Subsequently, in Section 9.8, we describe the setup used to implement the proposed architecture on FPGAs and evaluate the performance of the design; Finally, in Section 8.5, we draw some concluding remarks and point out directions for future work.

8.2 BLAST Algorithm

The BLAST (Basic Local Alignment Search Tool) [1] algorithm is a heuristic search-based method that seeks words in the subject sequence s of length w that score at least T, called the *alignment threshold*, when aligned with the query sequence t. The scoring process is performed according to predefined criteria that are usually prescribed by geneticists. This task is called *seeding*, where BLAST attempts to find regions of similarity to begin its matching procedure. This step has a very powerful heuristic advantage, because it only keeps pairs whose matching score is larger than the pre-defined threshold T. Of course, there is some risk of leaving out some worthy alignments. Nonetheless, using this strategy, the search space decreases drastically, and hence accelerating the convergence of the matching process.

After identifying all possible alignments locations or *seeds*, the algorithm proceeds with the *extension stage*. It consists of extending the found alignments to the right and left within both the subject and query sequences, in an attempt to find a locally optimal alignment. Some versions of BLAST introduce the use of a wildcard symbol (_), called the *gap*, which can be used to replace any mismatch [7, 10]. Here, we do not allow gaps. Finally, BLAST try to improve score of high scoring pairs, HSP, through a second extension process and the dismissal of a pair is done when the corresponding score does not reach a new pre-defined threshold. HSPs that meet this criterion will be reported by BLAST as final results, provided that they do not exceed the cutoff prescribed value, which specifies for number of descriptions and/or alignments that should be reported. This last step is called *evaluation*. In the implementation presented in this chapter, we do not assess the results provided by the extension stage. We simply provide all of them as a final result of the alignment process.

BLAST employs a measure based on a well-defined mutation scores. It directly approximates the results that would be obtained by any dynamic programming algorithm for optimizing this measure. The method allows for the detection of weak but biologically significant similarities. The algorithm is more than one order of magnitude faster than existing heuristic algorithms. Compared to other heuristics-based methods, such as FASTA [7], BLAST performs DNA and protein sequence similarity alignment much faster but it is considered to be equally sensitive.

The BLAST algorithm proceeds through three main steps: *(i)* seeding, which allows to find and mark all seeds. These are subsequences of size w that can be considered as alignment points. Algorithm 8.4 describe the work as it should be done during this step; *(ii)* extension, which extends at most, i.e. with respect to the limits of the subject and query sequences, all the marked seeds and marks all those extensions that scored more that the prescribed threshold T. The extension is done in both directions, i.e. to the right of the seed location in the subject and query sequences as well as to the its left; Algorithm 8.2 describes the extension done to the right of the seed. Note that the algorithm does the extension to the left (Algorithm 8.4) is similar to the one presented with the exception that sequence counters i and j are decremented and the base are appended to the left; *(iii)* assessment, which selects some of the alignments, as found by the extension stage, and applies some biological parameters to extract some few promising alignment to be considered further in the DNA matching biological process. This last step, as described in Algorithm 8.3, is not treated any further in this chapter.

Algorithm 8.1. Seeding procedure

```
Require: Subject and query sequences $s$ and $t$ respectively
Ensure: Matrix of seed location $hits$
 1: let $s = [s_0, s_1, \ldots, s_i, \ldots, s_{m-1}]$
 2: let $t = [t_0, t_1, \ldots, t_j, \ldots, t_{n-1}]$
 3: $sws \leftarrow [sw_0, sw_1, \ldots, sw_i, \ldots, sw_{m-w}]$, wherein    $sw_i = [s_i, s_{i+1}, \ldots, s_{i+w-1}]$
 4: $tws \leftarrow [tw_0, tw_1, \ldots, tw_j, \ldots, tw_{n-w}]$, wherein    $tw_j = [t_j, t_{j+1}, \ldots, t_{j+w-1}]$
 5: for $i = 0 \rightarrow (m-w)$ do
 6:    for $j = 0 \rightarrow (n-w)$ do
 7:       if $tw_i = sw_j$ then
 8:          $hits[i,j] \leftarrow 1$
 9:       else
10:          $hits[i,j] \leftarrow 0$
11:       end if
12:    end for
13: end for
```

Algorithm 8.2. Extension procedure (right)

```
Require: Sequences $s$ and $t$, seed offsets $i$ and $j$ respectively
Ensure: Extension score $\sigma$
 1: $E_s \leftarrow sw_i \boxplus s_{i+w}$; $k \leftarrow i$
 2: $E_t \leftarrow tw_j \boxplus t_{j+w}$; $\ell \leftarrow j$
 3: repeat
 4:     $k \leftarrow k+1$; $E'_s \leftarrow E_s$; $E_s \leftarrow E_s \boxplus s_{k+w+1}$
 5:     $\ell \leftarrow \ell+1$; $E'_t \leftarrow E_t$; $E_t \leftarrow E_t \boxplus t_{\ell+w+1}$
 6: until $(s_{k+w+1} \neq t_{\ell+w+1})$ or $(k > m-1)$ or $(\ell > n-1)$
 7: if $s_{k+w+1} = t_{\ell+w+1}$ then
 8:     $\sigma \leftarrow Scores(E_s, E_t)$
 9: else
10:     $\sigma \leftarrow Scores(E'_s, E'_t)$
11: end if
```

Algorithm 8.3. Assessment procedure

```
Require: Offsets $i$, $j$, threshold $T$ and extension score $\sigma$
Ensure: Matrix $hits$ updated
 1: if $\sigma \geq T$ then
 2:     $hits[i, j] \leftarrow \sigma$
 3: else
 4:     $hits[i, j] \leftarrow 0$
 5: end if
```

Algorithm 8.4. Extension procedure to the let

```
Require: $s$, $t$ and $hits$ as a results of seeding;
Ensure: $hits$ updated
 1: $E_s \leftarrow s_{i+w} \boxplus sw_i$; $k \leftarrow i$
 2: $E_t \leftarrow t_{j+w} \boxplus tw_j$; $\ell \leftarrow j$
 3: repeat
 4:     $k \leftarrow k-1$; $E'_s \leftarrow E_s$; $E_s \leftarrow s_{k+w+1} \boxplus E_s$
 5:     $\ell \leftarrow \ell-1$; $E'_t \leftarrow E_t$; $E_t \leftarrow t_{\ell+w+1} \boxplus E_t$
 6: until $(s_{k+w+1} \neq t_{\ell+w+1})$ or $(k < 0)$ or $(\ell < 0)$
 7: if $s_{k+w+1} = t_{\ell+w+1}$ then
 8:     $\sigma \leftarrow Scores(E_s, E_t)$
 9: else
10:     $\sigma \leftarrow Scores(E'_s, E'_t)$
11: end if
```

8.3 Proposed Architecture

The overview of the proposed architecture is depicted in Fig. 8.1. The Hardware HBLAST implements the BLAST algorithm, as described in Section 8.2. Besides the clock signal, it receives as input the subject and query sequences of m and n

bases respectively. Note that, in general, we have $m \ll n$. HBLAST also expects the configuration of three parameters: w, which determine the seed size, T, which sets up the required threshold value for alignment acceptance during extension, and p, which dictates the number of extension processor that will be used in parallel as it will be show later.

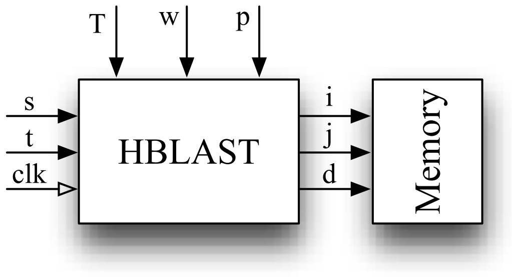

Fig. 8.1 Interface of the proposed design

As there are 4 DNA bases (A, C, G, and T), we need 2 bits to represent each base distinctively (00, 01, 10, 11). Instead of representing the subject and query sequences 2 registers of $2 \times m$ and $2 \times m$ bits respectively, we opted to use 2 registers of m bits to store subject sequence and 2 registers of n bits to hold the query sequence: one register of the pair holds the MSB of the DNA bases that form the sequence and the other the LSB. These two ways of storing the DNA sequences require the same number of flip-flops, but the second way improves the matching time as the two bits of a base can be compared in parallel without much increase in control, as they are provide by two distinct registers. The macro-architecture of HBLAST is given in Fig. 8.2. It includes a Seeding Unit that takes care of finding and bookkeeping all the seeds, with respect to s and t, and an Extension Unit that extends the seeds found.

A Global Controller synchronizes the work in pipeline of the seeding and extension units: seeds are handled by the Extension Unit as they come. There is no need to complete the seeding step before starting the extension work. A Scheduler arbitrates the use of the shared data and control buses between the Seeding and Extension Units. This is necessary because the Seeding Unit is, in turn, structurally formed by $q = n - w + 1$ concurrent sub-units and the Extension Unit is formed by p extension processors that act in parallel to accelerate the alignment process. The structural parallelism within the Seeding and Extension Unit is depicted in Fig. 8.3. The work of the q seeding components (Seeding$_i$) and the p extension components (Extension Processor$_j$) is harmonized by a respective stage controller, i.e. the Seeding Controller and the Extension Controller respectively.

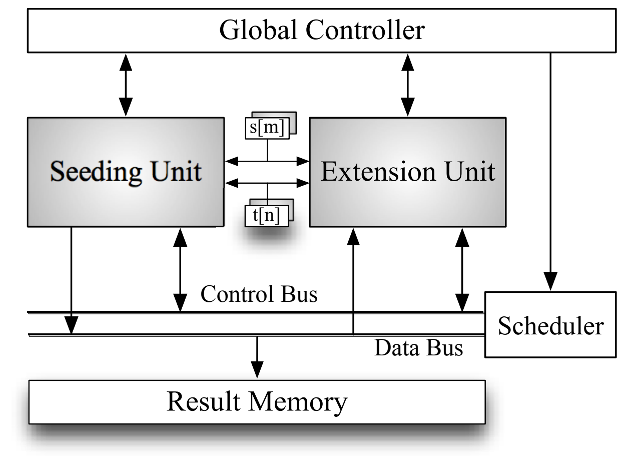

Fig. 8.2 Proposed macro-architecture

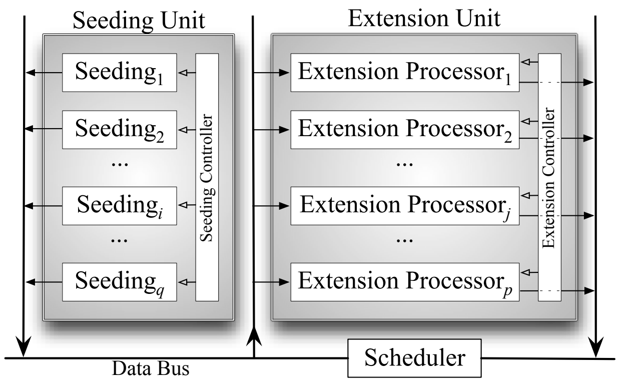

Fig. 8.3 Structural parallelism in the seeding and extension units

8.3.1 Seeding Unit

The design uses $q = n - w + 1$ concurrent Seeding components. Fig. 8.4 describes the corresponding micro-architecture along with the interface withe the Scheduler and the Seeding Controller. Each of these Seeding components includes 2 Matching Units: one for the comparison of the MSBs of subject and query DNA sequences and the other for the LSBs. The Matching Unit is a mere array of w XNOR gates whose results are summarized by an AND gate, as shown in the circuit of Fig. 8.5(a).

When a match of a target (*w* consecutive bits of *s*) and a word (*w* consecutive bits of *t*) is declared, i.e. the result of the both Matching units (MSB and LSB) are both 1, the stamp formed by the offset of the target and word is pushed down the FIFO. Note that there is one FIFO per Seeding Unit. The stamps are later popped to be considered for extension. Once a FIFO (or a Seeding Unit) is selected by the Scheduler to feed a requesting Extension Processor, the Write Logic of Fig. 8.5(b) allows the output stamp of the FIFO to be written into the Data Bus so as to be forwarded to the Extension Processor.

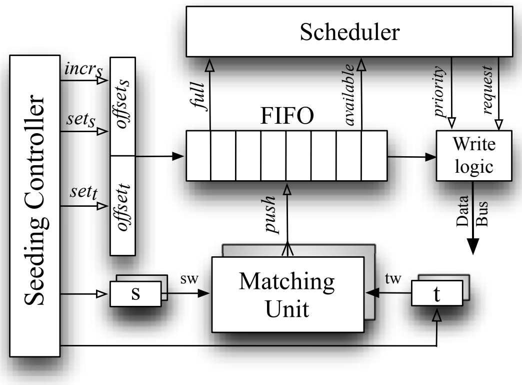

Fig. 8.4 Seeding unit micro-architecture

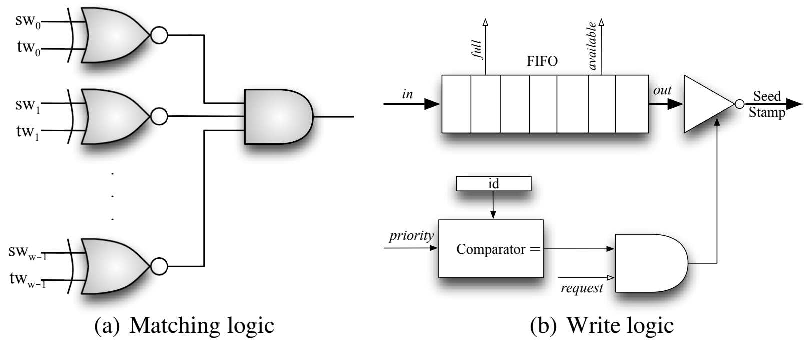

Fig. 8.5 Matching and write logic micro-architectures

8.3.2 Extension Unit

The Extension Unit includes p Extension Processors as shown in Fig. 8.6. The number of included processor is defined as external parameter. This number does not necessarily coincide with that of Seeding components as many seeds do not require much extension work. Some seeds are discarded in the first base extension. Note that it is intended that $p \ll q$. For this purpose, among others, a Scheduler is used to distributed the identified seeds (in the FIFOs) as soon as a processors becomes idle. When a processor completes the extension of a given seed and requests a new one to work with, the Scheduler that is made aware of the request, selects the FIFO that is already full, if any. Otherwise, it selects the FIFO that has less available space. In the case there two or more FIFOS with the same available space, the one with the smallest identifier is given precedence. Note that the work of a Seeding component is suspended when its respective FIFO becomes full. Thus the strategy adopted by the Scheduler in selecting the FIFO that is to serve the requesting extension processor aims at minimizing the number of halted Seeding components. As soon as an interruption is received by the

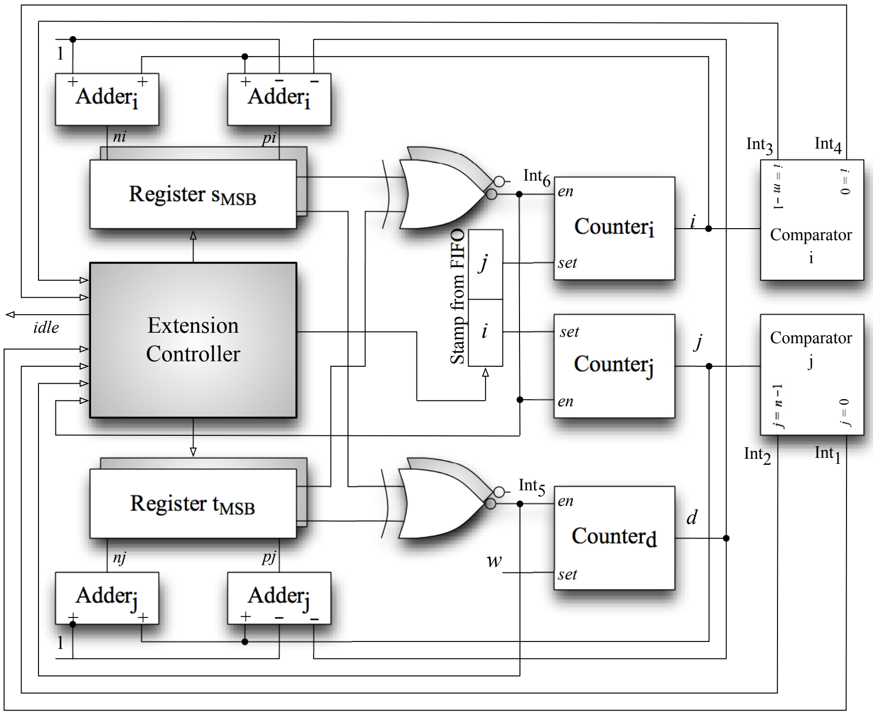

Fig. 8.6 Micro-architecture of the Extension Processor

Extension to the right and left are done parallel. The extension processor includes 4 adders that compute the new offset as well as the length of the matched subsequence. During the extension to the right, 2 adders compute $ni \leftarrow i+1$ and $nj \leftarrow j+1$ while during the extension to the left, the other 2 adders compute $pi \leftarrow i-d-1$ and $pj \leftarrow j-d-1$. These new indices allow the processor to have access to the new bases at the immediate left and right to the d bases already matched. At first, we have $d = w$, then d is incremented at every successful match. The actual update of indices i, j and d is done, by the counters, only once the match is declared. When a mismatch occurs, an interrupt (Int_6 or Int_5) is triggered to abandon the current seed. Two other interrupts can occur when the either all bases to seed's right or left on t (Int_1 or Int_2) or t those to the seed's right or left on s were treated. When interruption occurs, the Extension Controllers enables the writing of the triplet (i, j, d) into the Result Memory and signals to the Global Controller that the Extension Processor in question is idle and thus generates s request for a new seed to work pass it through to the processor.

8.3.3 The Controllers

The design includes 4 controllers: the Global Controller, the Seeding Controller and the Extension Controller and the Scheduler. Controllers are implemented as finite state machines.

The Global Controller is responsible mainly for the synchronization of the pipeline between the seeding and extension stages. Besides, it allows for the initialization of all components, the load of the DNA subject and query sequences into the corresponding registers and enabling the writing operation of the final results into the Result Memory.

The actions imposed by the seeding Controller guarantee the logic distribution of the DNA sequences into targets words son as to allow for the matching process to perform correctly. The main task of this controller consist of maintaining the content of register s and t coherent all the time by synchronizing the required shifting operations.

The Extension Controller is responsible for the correct performance of the p Extension Processors. It handles the interruption signals send by the Extension Processors and controls the injection of the bits that represent the bases that need to be considered during extension to the right and/or left, depending on the status of the triggered interruptions.

The Scheduler is responsible for controlling the use of the Data Bus as to forward an give seed stamp to an identified Extension Processor. It also selects the FIFO that needs to provide the seed stamp to be treated next when the Extension Controller signals that one of the Extension Processor became idle.

8.4 Performance Results

The MicroBlaze$^{\text{TM}}$ and the co-processor HBLAST were synthesized in a Xilinx Virtex 5 FPGA xc5vfx70t. The MicroBlaze is an embedded processor soft core, which is a reduced instruction set computer optimized for implementation on Xilinx$^{\text{TM}}$ FPGAs.

Without the proposed HBLAST co-processor, the MicroBlaze processor performs all the alignment process. In this case, the BLAST algorithm were implemented in ANSI/C++. The MicroBlaze has a communication interface for point-to-point, called Fast Simplex Link (FSL), which allows for an efficient connection with an external co-processor. In the remainder of this section, we will first introduce the performance figures of the HBLAST proposed design in terms of area and time requirements, then we compare the performance of the Microblaze-based implementation (software implementation) and that occasioned by the use of HBLAST as a co-processor (hardware implementation).

Table 8.1 shows the impact of varying the number of bases in the subject and query sequences on both area and time requirements. Note that in case 4, wherein $m = 100$ and $n = 25$, the hardware resources available on the used FPGA were exhausted and thus no time figure is given in this case. Fig. 8.7 illustrates graphically this impact.

Table 8.1 Hardware area and time requirements for diffrent configuration of m and n

#	m	n	FFs	%	LuTs	%	Slices	%	Time
1	20	10	7887	18	7811	17	3300	29	12.59
2	60	20	33418	75	33124	74	10900	97	14.91
3	100	10	27767	62	28307	63	9547	85	19.57
4	100	25	49907	111	50952	114	12411	111	—

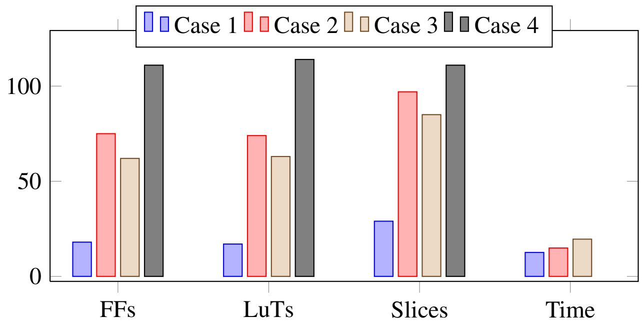

Fig. 8.7 Impact of the number of seed bases on the area and time requirements

Table 8.2 shows the impact of the value chosen for the seed size w. It is possible to note that adjusting the setting of this parameter can be a way to remedy to the case when the hardware are required is slightly above the available resources. Note that in this case, we set $m = 20$, $n = 10$ and $p = 2$. A graphical illustration of this effect is shown in Fig. 8.8.

Table 8.2 Hardware area and time requirements as w increases

w	FFs	%	LuTs	%	Slices	%	Time
3	7887	18	7811	17	3300	29	12.59
4	7205	16	7278	16	3258	29	7.64
5	6523	15	6689	15	2963	26	5.71
6	5841	13	4553	10	2631	23	4.86

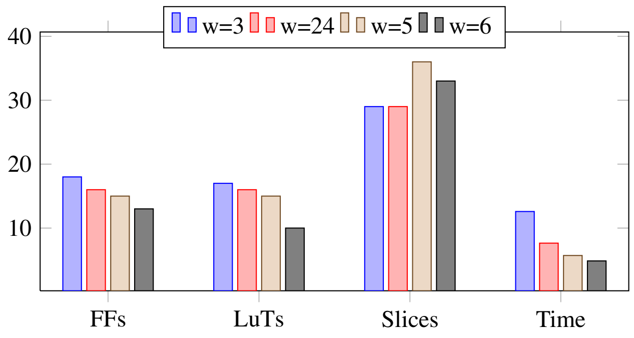

Fig. 8.8 Impact of the number of seed bases on the area and time requirements

In order to verify the improvement in terms of performance, if any, *vs.* the increase in terms of hardware area requirements occasioned by the use of more extension processors, we set $m = 20$, $n = 10$ and $w = 3$ and varied the number of processors p. Table 8.3 shows the impact as p increases. Fig. 8.9 illustrates graphically this impact.

Table 8.3 Hardware area and time requirements as p increases

p	FFs	%	LuTs	%	Slices	%	Time
1	6435	14	6622	15	3109	28	17.63
2	7887	18	7811	17	3300	29	12.59
3	7887	18	7804	17	3438	31	10.08
4	8004	18	7989	18	3672	33	8.50

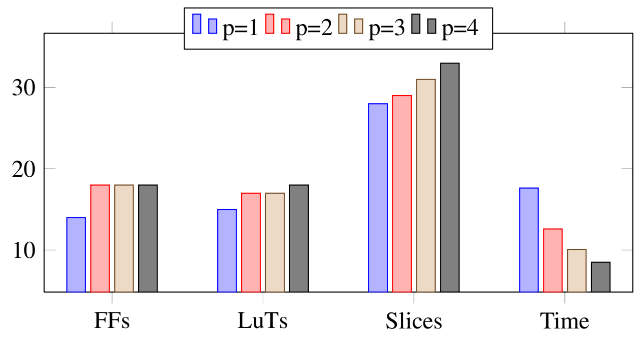

Fig. 8.9 Impact of the number of processor on the area and time requirements

Table 8.4 shows the time requirements of the MicroBlaze software implementation and the HBLAST hardware implementation. The operation frequency of processor MicroBlaze is 50 MHz while HBLAST runs at different frequencies as shown the penultimate column of Table 8.4. Fig. 8.10 illustrates, in a logarithmic scale, the comparison of the MicroBlaze and HBLAST performances, as well as the speedup achieved by using HBLAST. The average speedup is about 60×.

Table 8.4 Microblaze *vs* HBLAST time comparison

Case	m	n	Microblaze		HBLAST		
			#Cycles	Time	#Cycles	Freq.	Times
1	20	10	32411	528.59	772	61.3	12.59
2	60	20	54393	996.21	814	54.6	14.91
3	100	10	54919	1065.04	1009	51.5	19.58
4	100	25	255454	5109.08	3206	50.0	64.12

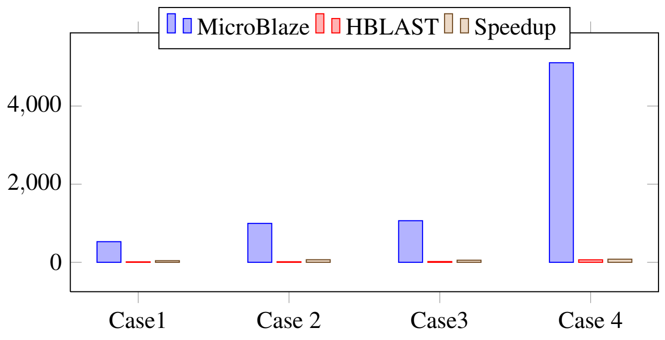

Fig. 8.10 Impact of the number of processor on the area and time requirements

8.5 Summary

This chapter presents a parallel architecture of the BLAST algorithm implemented as a hardware co-processor to the MicroBlaze processor. BLAST is used to align DNA sequences. The FPGA used is a Xilinx Virtex 5 FPGA (xc5vfx70t). The proposed architecture exploits the parallelism of identifying the seeds using a strategic partitioning of the subject and query sequences into words of a configurable size in terms of bases. It also explores further parallelism as it includes many extension processors to investigates the seeds found. The seeding and extension processes are carried on in a pipelined fashion.

Moreover, the design is easily scalable to new configuration parameter, which consist of the seed size *w* in terms of number of bases and the number of extension processors *p*. This adjustment cab be done according to speed *vs.* cost constraints.

A thorough analysis of the impact of each of the algorithm parameters has been done to evaluate the impact in terms of hardware are and time requirements. A comparison of the software-based and the proposed hardware design showed that a speedup of 60$\times$ is achieved in average.

Future work will be directed at completing the assessment step and analyzing the impact on the whole design, as well as the use of real-world cases DNA alignment and matching.

References

1. Altschul, S., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of Molecular Biology 215(3), 403–413 (1990)
2. Mount, D.W.: Steps used by the BLAST algorithm. Cold Spring Harbor Protocols: Molecular Biology (2007), doi:10.1101/pdb.ip41
3. Needlman, S., Wunsh, S.: A general method applicable to the search of similarities in Amino-Acid sequence of two protein. Journal of Molecular Biology 1(48), 443–453 (1970)
4. Garcia Neto Segundo, E.J., Nedjah, N., de Macedo Mourelle, L.: A parallel architecture for DNA matching. In: Xiang, Y., Cuzzocrea, A., Hobbs, M., Zhou, W. (eds.) ICA3PP 2011, Part II. LNCS, vol. 7017, pp. 399–407. Springer, Heidelberg (2011)
5. Giegerich, R.A.: Systematic approach to dynamic programming in bioinformatics. Bioinformatics 8(16), 665–677 (2000)
6. Kasap, S., Benkrid, K.: High performance phylogenetic analysis with maximum parsimony on reconfigurable hardware. IEEE Transactions on Very Large Scale Integration VLSI Systems 99(5), 796–808 (2011)
7. Pearson, W.: Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms. Genomics 3(11), 635–650 (1991)
8. Rubin, E., Pietrokovski, S.: Heuristic methods for sequence alignment. Advanced Topics in Bioinformatics, Weizmann Institute of Science (2003)
9. Shaper, E.G., et al.: Sensitivity and selectivity in protein similarity searches: a comparison of Smith-Waterman in hardware to BLAST and FASTA. Genomics 2(38), 179–191 (1996)
10. Waterman, M.S.: Introduction to Computational Biology. CRC Press (1995)