High Performance Biosequence Database Scanning on Reconfigurable Platforms

Timothy Oliver and Bertil Schmidt
School of Computer Engineering
Nanyang Technological University
Singapore 639798
Tim.oliver@pmail.ntu.edu.sg, asbschmidt@ntu.edu.sg

Abstract

Molecular biologists frequently compare an unknown protein sequence with a set of other known sequences (a database scan) to detect functional similarities. Even though efficient dynamic programming algorithms exist for the problem, the required scanning time is still very high, and because of the rapid database growth finding fast solutions is of highest importance to research in this area. In this paper we present a new approach to biosequence database scanning on reconfigurable hardware platforms to gain high performance at low cost. To derive an efficient mapping onto this type of architecture, we have designed fine-grained parallel processing elements (PEs). Since our solution is based on reconfigurable hardware, we can design PEs that are tailored towards the parameters of a query. This results in an implementation with significant runtime savings on a standard off-the-shelf FPGA.

1. Introduction

Scanning protein sequence databases is a common and often repeated task in molecular biology. The need for speeding up this treatment comes from the exponential growth of the biosequence banks: every year their size scaled by a factor 1.5 to 2. The scan operation consists of finding similarities between a particular query sequence and all sequences of a bank. This operation allows biologists to point out sequences sharing common subsequences. From a biological point of view, it leads to identify similar functionality.

Comparison algorithms whose complexities are quadratic with respect to the length of the sequences detect similarities between the query sequence and a subject sequence. One frequently used approach to speed up this time consuming operation is to introduce heuristics in the search algorithm [1]. The main drawback of this solution is that the more time efficient the heuristics, the worse is the quality of the result [17].

Another approach to get high quality results in a short time is to use parallel processing. There are two basic methods of mapping the scanning of sequence databases to a parallel processor: one is based on the systolisation of the sequence comparison algorithm, the other is based on the distribution of the computation of pairwise comparisons. Systolic array architectures have been proven as a good candidate structure for the first approach [5,12,18], while more expensive supercomputers and networks of workstations are suitable architectures for the second [7,15].

Special-purpose systolic arrays provide the best area/performance ratio by means of running a particular algorithm [14]. Their disadvantage is the lack of flexibility with respect to the implementation of different algorithms. Several massively parallel SIMD architectures have been developed in order to combine the speed and simplicity of systolic arrays with flexible programmability [3,6,19]. However, because of the high production costs involved, there are many cases where announced second-generation architectures have not been produced. The strategy to high performance sequence database scanning used in this paper is based on FPGAs. FPGAs provide a flexible platform for fine-grained parallel computing based on reconfigurable hardware. Since there is a large overall FPGA market, this approach has a relatively small price/unit and also facilitates upgrading to FPGAs based on state-of-the-art technology. Taking full advantage of hardware reconfiguration, we present PE designs that are tailored towards particular query parameters. We will show how this leads to a high-speed implementation on a Virtex II XC2V6000. The implementation is also portable to other FPGAs.

This paper is organised as follows. In Section 2, we introduce the basic sequence comparison algorithm for database scanning. Section 3 highlights previous work in parallel sequence comparison. The parallel algorithm and its mapping onto a reconfigurable platform are explained in Section 4. The performance is evaluated and compared to previous implementations in Section 5. Section 6 concludes the paper.

2. Sequence Comparison Algorithm

Surprising relationships have been discovered between protein sequences that have little overall similarity but in which similar subsequences can be found. In that sense, the identification of similar subsequences is probably the most useful and practical method for comparing two sequences. The Smith-Waterman algorithm [20] finds the most similar subsequences of two sequences (the local alignment) by dynamic programming.

The algorithm compares two sequences by computing a distance that represents the minimal cost of transforming one segment into another. Two elementary operations are used: substitution and insertion/deletion (also called a gap operation). Through series of such elementary operations, any segments can be transformed into any other segment. The smallest number of operations required to change one segment into another can be taken into as the measure of the distance between the segments.

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

Figure 1. Example of the Smith-Waterman algorithm to compute the local alignment between two DNA sequences ATCTCGTATGATG and GTCTATCAC. The matrix *H*(*i*,*j*) is shown for the linear gap cost $\alpha = 1$, and a substitution cost of +2 if the characters are identical and −1 otherwise. From the highest score (+10 in the example), a traceback procedure delivers the corresponding alignment (shaded cells), the two subsequences TCGTATGA and TCTATCA.

Consider two strings $S1$ and $S2$ of length $l1$ and $l2$. To identify common subsequences, the Smith-Waterman algorithm computes the similarity $H(i,j)$ of two sequences ending at position i and j of the two sequences $S1$ and $S2$. The computation of $H(i,j)$ is given by the following recurrences:

$H(i,j) = \max\{0, E(i,j), F(i,j), H(i-1,j-1)+Sbt(\mathrm{S1}_i,\mathrm{S2}_j)\}$, for $1 \le i \le l1$, $1 \le j \le l2$.

$E(i,j) = \max\{H(i,j-1)-\alpha, E(i,j-1)-\beta\}$, $0 \le i \le l1$, $1 \le j \le l2$.

$F(i,j) = \max\{H(i-1,j)-\alpha, F(i-1,j)-\beta\}$, $1 \le i \le l1$, $0 \le j \le l2$.

where Sbt is a character substitution cost table. Initialization of these values are given by $H(i,0) = E(i,0) = H(0,j) = F(0,j) = 0$ for $0 \le i \le l1$, $0 \le j \le l2$. Multiple gap costs are taken into account as follows: α is the cost of the first gap; β is the cost of the following gaps. This type of gap cost is known as *affine gap penalty*. Some applications also use a *linear gap penalty*, i.e. $\alpha = \beta$. For linear gap penalties the above recurrence relations can be simplified to:

$H(i,j) = \max\{0, H(i,j-1)-\alpha, H(i-1,j)-\alpha, H(i-1,j-1) + Sbt(\mathrm{S1}_i,\mathrm{S2}_j)\}$, for $1 \le i \le l1$, $1 \le j \le l2$.

$H(i,0) = H(0,j) = 0$ for $0 \le i \le l1$, $0 \le j \le l2$.

Each position of the matrix H is a similarity value. The two segments of $S1$ and $S2$ producing this value can be determined by a backtracking procedure. Fig. 1 illustrates an example.

3. Previous Work

A number of parallel architectures have been developed for sequence analysis. In addition to architectures specifically designed for sequence analysis, existing programmable sequential and parallel architectures have been used for solving sequence alignment problems.

Special-purpose hardware implementations can provide the fastest means of running a particular algorithm with very high PE density. However, they are limited to one single algorithm, and thus cannot supply the flexibility necessary to run a variety of algorithms required analyzing DNA, RNA, and proteins. P-NAC was the first such machine and computed edit distance over a four-character alphabet [16]. More recent examples, better tuned to the needs of computational biology, include BioScan, BISP, and SAMBA [5,12,18].

An approach presented in [19] is based on instruction systolic arrays (ISAs). ISAs combine the speed and simplicity of systolic arrays with flexible programmability. Several other approaches are based on the SIMD concept, e.g. MGAP [3], Kestrel [6], and Fuzion [19]. SIMD and ISA architectures are programmable and can be used for a wider range of applications, such as image processing and scientific computing. Since these architectures contain more general-purpose parallel processors, their PE density is less than the density of special-purpose ASICs. Nevertheless, SIMD solutions can still achieve significant runtime savings. However, the costs involved in designing and producing SIMD architectures are quite high. As a consequence, none of the above solutions has a successor generation, making upgrading impossible.

Reconfigurable systems are based on programmable logic such as field-programmable gate arrays (FPGAs) or custom-designed arrays. They are generally slower and have lower PE densities than special-purpose architectures. They are flexible, but the configuration must be changed for each algorithm, which is generally more complicated than writing new code for a programmable architecture. Several solutions including Splash-2 [13] and Decipher [21] are based on FPGAs while PIM has its own reconfigurable design [8]. Solutions based on FPGAs have the additional advantage that they can be regularly upgraded to state-of-the-art-technology. This makes FPGAs a very attractive alternative to special-purpose and SIMD architectures.

Compared to the previously published FPGA solutions, we are using a new partitioning technique for varying query sequence lengths. The design presented in [22] is closest to our approach since it also uses a linear array of PEs on a reconfigurable platform. Unfortunately, it only allows for linear gap penalties and global alignment, while our implementation considers both linear and affine gap penalties and is able to compute local alignments.

4. Mapping of Sequence Comparison on a Reconfigurable Platform

The dynamic programming calculation presented in Section 2 can be efficiently mapped to a linear array of PEs. A common mapping is to assign one PE to each character of the query string, and then to shift a subject sequence systolically through the linear chain of PEs (see Figure 2). If M is the length of the first sequence and K is the length of the second, the comparison is performed in $M+K-1$ steps on M PEs, instead of $M\times K$ steps required on a sequential processor. In each step the computation for dynamic programming cells along a single diagonal in Figure 1 is performed in parallel.

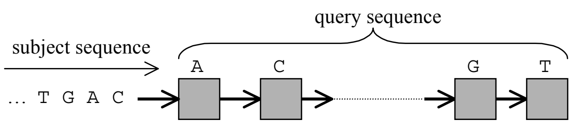
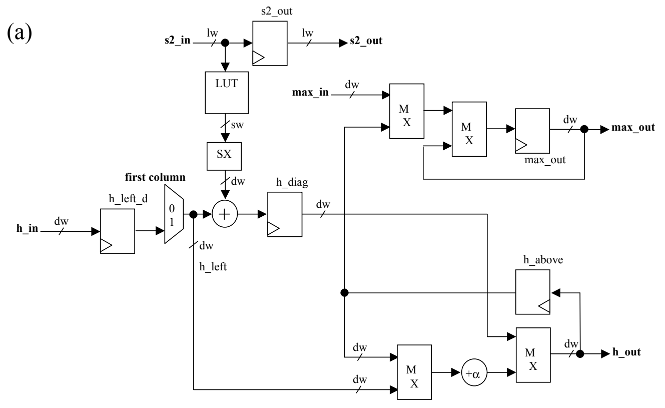


Figure 2. Sequence comparison on a linear processor array: the query sequence is loaded into the processor array (one character per PE) and a subject sequence flows from left to right through the array. During each step, one elementary matrix computation is performed in each PE.



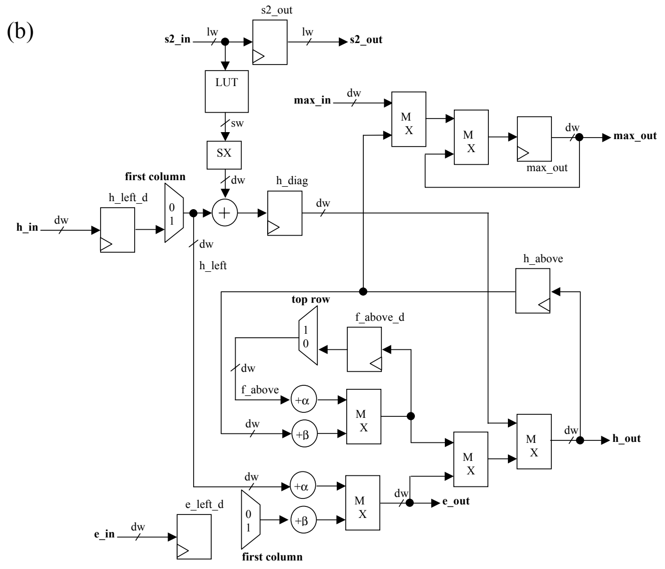


Figure 3. (a) Shows linear gap penalty PE. (b) Shows affine gap penalty PE. Data width (*dw*) is scaled to the required precision (usually *dw*=16 is sufficient). The LUT depth is scaled to hold the required number of substitution table rows. Substitution width (*sw*) is scaled to accommodate the dynamic range required by the substitution table. Look-up address width (*lw*) is scaled in relation to the LUT depth. Each PE has local memory to store $H(i,j-1)$, $H(i-1,j)$ and $H(i-1,j-1)$. The PE holds a column of the substitution table in its LUT. The look-up of $Sbt(S1_i,S2_j)$ and addition to $H(i-1,j-1)$ is done in one cycle. The score is calculated in the next cycle and passed to the next PE in the array. The PE keeps track of the maximum score computed so far and passes it to the next PE in the array. The affine gap penalty PE has additional storage for $E(i,j-1)$ and $F(i-1,j)$ and additional score

computation circuitry. Additions are performed using saturation arithmetic.

Taking advantage of having a reconfigurable hardware platform, we can tailor the individual PE design towards different gap penalty functions. This approach allows us to include only as much computational hardware and local memory as required. Figure 3 shows our designs for linear gap penalties and for affine gap penalties.

Assuming, we are aligning the sequences $A = a_1a_2\ldots a_M$ and $B = b_1b_2\ldots b_K$, on a linear processor array of size M with affine gap penalties, where A is the query sequence and B is a subject sequence of the database. As a preprocessing step, symbol a_i, is loaded into PE i, $1 \leq i \leq M$. After that the row of the substitution table corresponding to the respective character is loaded into each PE as well as the gap penalties α and β. B is then completely shifted through the array in $M+K-1$ steps as displayed in Figure 2. In iteration step k, $1 \leq k \leq M+K-1$, the values $H(i,j)$, $E(i,j)$, and $F(i,j)$ for all i, j with $1 \leq i \leq M$, $1 \leq j \leq K$ and $k=i+j-1$ are computed in parallel in all PEs $1 \leq i \leq M$, within a single clock cycle. For this calculation PE i, $2 \leq i \leq M$, receives the values $H(i,j-1)$, $E(i,j-1)$, and b_j from its left neighbour $i-1$, while the values $H(i-1,j-1)$, $H(i-1,j)$, $F(i-1,j)$, a_i, α, β, and $Sbt(a_i,b_j)$ are stored locally. PE 0 receives b_j in steps j with $1 \leq j \leq K$. Computation for linear gap penalties is similar.

Thus, it takes $M+K-1$ steps to compute the alignment score of the two sequences with the SW algorithm. However, notice that after the last character of B enters the array, the first character of a new subject sequence can be input for the next iteration step. Thus, all subject sequences of the database can be pipelined with only one step delay between two different sequences.

Because of the very limited memory of each PE, only the highest score of matrix H is computed on the FPGA for each pairwise comparison. Ranking the compared sequences and reconstructing the alignments are carried out by the front end PC. Because this last operation is only performed for very few subject sequences, its computation time is negligible.

Our PE design incorporates the maximum computation of the matrix H with only a constant time penalty as follows: After each iteration step all PEs compute a new value *max* by taking the maximum of the newly computed H-value and the old value of *max* from its left neighbor. After the last character of a subject sequence has been processed in PE M, the maximum of matrix H is stored in PE M, which is then written into the off-chip memory.

So far we have assumed a processor array equal in size of the query sequence length. In practice, this rarely happens. Since the length of the sequences may vary (several thousands in some cases, however commonly the length is only in hundreds), the computation must be partitioned on the fixed size processor array. The query sequence is usually larger than the processor array. For sake of clarity we firstly assume a query sequence of length M and a processor array of size N where M is a multiple of N, i.e. $M=k \cdot N$ where $k \geq 1$ is an integer. A possible solution is to split the computation into k passes:

The first N characters of the query sequence are loaded into the processor array together with the corresponding substitution table columns. The entire database then crosses the array; the H-value and E-value computed in PE N in each iteration step are output. In the next pass the following N characters of the query sequence are loaded into the array. The data stored previously is loaded together with the corresponding subject sequences and sent again through the processor array. The process is iterated until the end of the query sequence is reached.

Unfortunately, this solution requires a large amount of off-chip memory (assuming 16-bit accuracy for intermediate results, four times database size bytes per pass are needed). The memory requirement can be reduced by factor p by splitting the database into p equal-sized pieces and computing the alignment scores of all subject sequences within each piece. However, this approach also increases the loading time of substitution table columns by factor p.

In order to eliminate this loading time we have slightly extended our PE design. Each PE now stores k columns of the substitution table instead of only one. Although this increases the area per PE a bit (see Section 5 for details), it allows for alignment of each database sequence with the complete query sequence without additional delays. It also reduced the required off-chip memory for storing intermediate results to four times longest database sequence size (again assuming 16-bit accuracy). Figure 4 illustrates our solution. We can again take advantage of reconfiguration and design different configurations for different values of k. This allows us to load a particular configuration that is suited for a range of query sequence lengths.

So far we have assumed that the query sequence length M is a multiple of the processor array size N, i.e. $M = k \cdot N$ where k is an integer. If this is not the case, we can still use our design by filling substitution table columns in the remaining PEs with zeros.

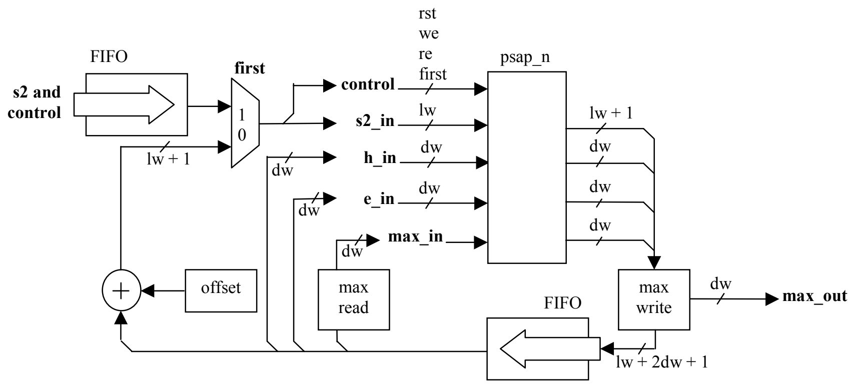


Figure 4. System Implementation: The linear array of PEs is encapsulated in *psap_n.* The database sequences are passed in from the host one by one through a FIFO to the *S*2 interface. The database sequences have been pre-converted to LUT addresses. For query lengths longer than the PE array the intermediate results are stored in a FIFO of width 2×*dw* + *lw* + 1 for affine gap penalty. For linear gap penalty the FIFO width is *dw* + *lw* + 1. The FIFO depth is sized to hold the longest sequence in the database. The database sequence is also stored in the FIFO. On each consecutive pass an LUT offset is added to address the next column of the substitution table stored within the PEs. The maximum score on each pass is compared with those from all other passes and the absolute maximum is returned to the host.

Table 1. Achieved number of PEs and clock frequencies of our different designs on a Virtex II XC2V6000. The maximal query sequence lengths and performance (in Giga CUPS) for each design is also reported.

Design	Number of PEs	Clock frequency	Max. query length	Performance
Linear, *k*=3	252	55 MHz	756	13.9 GCUPS
Linear, *k*=12	168	55 MHz	2016	9.2 GCUPS
Affine, *k*=3	168	45 MHz	504	7.6 GCUPS
Affine, *k*=12	126	45 MHz	1512	5.7 GCUPS

5. Performance Evaluation

We have described the PE design in Verilog and targeted it to the Xilinx Virtex II architecture. The size of a linear gap penalty PE is 3×10 CLBs and the size of an affine gap penalty PE is 6×8 CLBs. Figure 5 shows the layout plans. We have implemented a linear array of these PEs. Using a Virtex II XC2V6000 we are able to accommodate 252 linear PEs or 168 affine PEs using *k*=3. This allows handling of query sequence lengths up to 756 and 504 respectively, which is sufficient in most cases (74% of sequences in Swiss-Prot are ≤ 500 [2]). For longer queries we have implemented a design with *k* = 12, which can accommodate 168 linear PEs or 126 affine PEs. The corresponding clock frequencies are 55 MHz for linear and 45 MHz for affine.

A performance measure commonly used in computational biology is *cell updates per second* (CUPS). A CUPS represents the time for a complete computation of one entry of the matrix *H*, including all comparisons, additions and maxima computations. The CUPS performance of our implementations can be measured by multiplying number of PEs times clock frequency. Table 1 summarizes our results.

Since CUPS does not consider data transfer time, query length and initialization time, it is often a weak measure that does not reflect the behavior of the complete system. Therefore, we will use database scans for different query lengths in our evaluation. Table 2 reports the performance for scanning the Swiss-Prot protein databank (release 42.5, which contains 138’922 sequences comprising 51’131’444 amino acids [2]) for query sequences of various lengths using our design on an RC2000 FPGA Mezzanine PCI-board with a Virtex II XC2V6000 from Celoxica [4].

(a) (b)

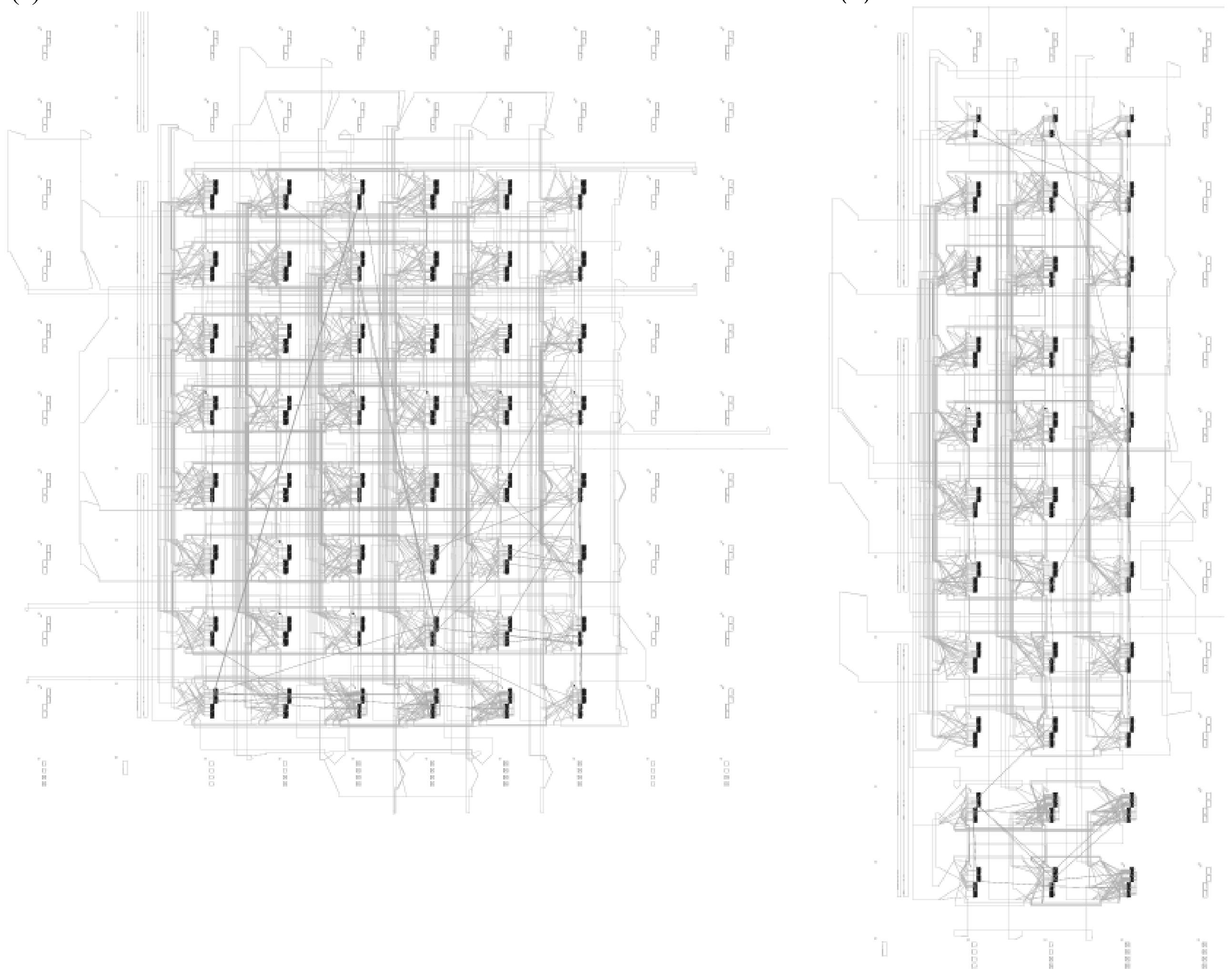


Figure 5. Layout plans for a single affine gap penalty PE (a) and for a single linear gap penalty PE (b) on the Virtex II architecture using *k*=3.

Table 2. Performance evaluation for various query sequence length ranges of our implementation (for both linear and affine gap penalties) on a Virtex II XC2V6000 FPGA. Mean performance indicates the performance for the mean value of the corresponding query length range.

Query length range	Mean performance (linear)	Query length range	Mean performance (affine)
1 – 252	5.6 GCUPS	1 – 168	3.2 GCUPS
253 – 504	9.2 GCUPS	169 – 336	5.1 GCUPS
505 – 756	10.6 GCUPS	337 – 504	5.8 GCUPS
757 – 840	8.3 GCUPS	505 – 630	4.8 GCUPS
841 – 1004	8.0 GCUPS	631 – 756	5.0 GCUPS

For the same application an optimized C-program on a Pentium IV 1.6 GHz has a performance of 52 MCUPS for linear gap penalties and 40 MCUPS for affine gap penalties. Hence, our FPGA implementation achieves a speedup of approximately 170 for linear gap penalties and 125 for affine gap penalties.

For the comparison of different massively parallel machines, we have taken data from [6,12,19,22] for a database search with the SW algorithm for different query lengths. The Virtex II XC2V6000 is around ten times faster than the much larger 16K-PE MasPar. Kestrel, Fuzion and Systola 1024 are one-board SIMD solutions. Kestrel is 12 times slower [6], Fuzion is two to three times slower [19], and Systola is around 50 times slower [19] than our solution. All these boards reach a lower performance, because they have been built with older CMOS technology (Kestrel: 0.5-µm, Fuzion: 0.25-µm, Systola 1024: 1.0-µm) than the Virtex II XC2V6000 (0.15-µm). Extrapolating to this technology both SIMD and reconfigurable FPGA platforms have approximately equal performance. However, the difference between both approaches is that FPGAs allow easy upgrading, e.g. targeting our design to a Virtex II XC2V8000 would improve the performance by around 30%.

Our implementation is slower than the FPGA implementations described in [9,13,23]. However, all these designs only implement edit distance. This greatly simplifies the PE design and therefore achieves a higher PE density as well as a higher clock frequency. Although of theoretical interest, edit distance is not used in practice because it does not allow for different gap penalties and substitution tables. The FPGA implementation presented in [22] on a Virtex XCV2000E is around three times slower than our solution. Unfortunately, the design only implements global alignment.

6. Conclusions

In this paper we have demonstrated that reconfigurable hardware platforms provide a cost-effective solution to high performance biosequence database scanning. PE designs for linear gap penalties and for affine gap penalties have been presented. We have described a partitioning strategy to implement database scans with a fixed-size processor array and varying query sequence lengths. Using our PE design and our partitioning strategy we can achieve supercomputer performance at low cost on an off-the-shelf FPGA.

The exponential growth of genomic databases demands even more powerful parallel solutions in the future. Because comparison and alignment algorithms that are favoured by biologists are not fixed, programmable parallel solutions are required to speed up these tasks. As an alternative to inflexible special-purpose systems, hard-to-upgrade SIMD systems, and expensive supercomputers, we advocate the use of reconfigurable hardware platforms based on FPGAs.

Our future work includes extending our design to database scanning with hidden Markov Models using the Viterbi algorithm and making our implementation available as a special resource in a computational grid. We will be making the design more flexible at run-time. This requires the processors to be described using a language like Xilinx's RTPCore [10] specification which, in turn, uses the JBits API [11].

References

[1] Altschul, S.F., Gish, W., Miller, W., Myers, E.W, Lipman, D.J.: Basic local alignment search tool, *Journal of Molecular Biology* 215 (1990) 403-410.

[2] Boeckmann, B., Bairoch, A., Apweiler, R., Blatter, M.-C., Estreicher, A., Gasteiger, E., Martin, M.J., Michoud, K., O'Donovan, C., Phan, I., Pilbout, S., Schneider, M.: The SWISS-PROT protein knowledgebase and its supplement TrEMBL in 2003 *Nucleic Acids Research* 31(2003) 365-370.

[3] Borah, M., Bajwa, R.S., Hannenhalli, S., Irwin, M.J.: A SIMD solution to the sequence comparison problem on the MGAP, *in Proc. ASAP'94*, IEEE CS (1994) 144-160.

[4] Celoxica Corporation, www.celoxica.com

[5] Chow, E., Hunkapiller, T., Peterson, J., Waterman, M.S.: Biological Information Signal Processor, *Proc. ASAP'91*, IEEE CS (1991) 144-160.

[6] Dahle, D., Grate L., Rice, E., Hughey, R.: The UCSC Kestrel general purpose parallel processor, *Proc. Int. Conf. Parallel and Distributed Processing Techniques and Appilcations* (1999) 1243-1249.

[7] Glemet, E., Codani, J.J.: LASSAP, a Large Scale Sequence compArison Package, *CABIOS* 13 (2) (1997) 145-150.

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

[9] Guccione, S.A., Keller, E.: Gene Matching using JBits, *Proc. 12th Int. Workshop on Field-Programmable Logic and Applications* (FPL'02), Springer, LNCS 2438 (2002) 1168-1171.

[10] Guccione, S.A., Levi, D.: Run-Time Parameterizable Cores, *Proc.* 9th *Int. Workshop on Field Programmable Logic and Applications* (FPL'99), Springer, LNCS 1673, (1999) 215-222.

[11] Guccione, S.A., Levi, D., Sundararajan, P.: JBits: A Java-based Interface for Reconfigurable Computing, *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD Con'99)*

[12] Guerdoux-Jamet, P., Lavenier, D.: SAMBA: hardware accelerator for biological sequence comparison, *CABIOS* 12 (6) (1997) 609-615.

[13] Hoang, D.T.: Searching genetic databases on Splash 2, in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, IEEE CS, (1993) 185-191.

[14] Hughey, R.: Parallel Hardware for Sequence Comparison and Alignment, *CABIOS* 12 (6) (1996) 473-479.

[15] Lavenier, D., Pacherie, J.-L.: Parallel Processing for Scanning Genomic Data-Bases, *Proc. PARCO'97*, Elseiver (1998) 81-88.

[16] Lopresti, D.P.: P-NAC: A systolic array for comparing nucleic acid sequences, *Computer* 20 (7) (1987) 98-99.

[17] Pearson, W.R.: Comparison of methods for searching protein sequence databases, *Protein Science* 4 (6) (1995) 1145-1160.

[18] Singh, R.K. et al.: BIOSCAN: a network sharable computational resource for searching biosequence databases, *CABIOS*, 12 (3) (1996) 191-196.

[19] Schmidt, B., Schröder, H., Schimmler, M: Massively Parallel Solutions for Molecular Sequence Analysis, *Proc. 1st IEEE Int. Workshop on High Performance Computational Biology,* Ft. Lauderdale, Florida, 2002.

[20] Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences*, Journal of Molecular Biology* 147 (1981) 195-197.

[21] TimeLogic Corporation, http://www.timelogic.com.

[22] Yamaguchi, Y., Maruyama, T., Konagaya, A.: High Speed Homology Search with FPGAs, *Proc. Pacific Symposium on Biocomputing'02*, pp.271-282, (2002).

[23] Yu, C.W., Kwong, K.H., Lee, K.H., Leong, P.H.W.: A Smith-Waterman Systolic Cell, *Proc. 13th Int. Workshop on Field Programmable Logic and Applications (FPL'03)*, Springer, LNCS 2778, (2003) 375-384.