A General Smith-Waterman Algorithm Implementation Using the CREC Reconfigurable Computer

Balint Szente
Department of Automation
"Petru Maior" University of Targu-Mures
Nicolae Iorga 1, Targu-Mures, RO-540088
Romania
bszente@rdslink.ro

Octavian Cret
Computer Science Department
Technical University of Cluj-Napoca
Gheorghe Baritiu 26, Cluj-Napoca, RO-400027
Romania
cret@cs.utcluj.ro

Zsolt Mathe
Computer Science Department
Technical University of Cluj-Napoca
Gheorghe Baritiu 26, Cluj-Napoca, RO-400027
Romania
mazsolt@fastmail.fm

Cristian Vancea
Computer Science Department
Technical University of Cluj-Napoca
Gheorghe Baritiu 26, Cluj-Napoca, RO-400027
Romania
vcristian14@yahoo.com

Florin Rusu
Computer Science Department
Technical University of Cluj-Napoca
Gheorghe Baritiu 26, Cluj-Napoca, RO-400027
Romania
usur_nirolf@yahoo.com

Abstract – **The Smith-Waterman is one of the fundamental algorithms in the field of Bioinformatics. Several hardware implementations of this algorithm were reported, but only for some very particular cases of it. This paper presents a *general case implementation* of this algorithm using the *CREC* low-cost, General-Purpose Reconfigurable Computer. The main idea of the CREC system is to generate the best-suited hardware architecture for each software application through a Hardware/Software CoDesign process during which the aim is to exploit the high intrinsic parallelism of the application. The hardware architecture is described in VHDL code that is automatically generated by a program, written in ANSI C. Finally, CREC system is implemented in a FPGA device. The overall hardware architecture, the software code and the performance estimation formulas are presented and used to demonstrate the system's efficiency, which is close to the dedicated systolic implementations. The obtained results prove the efficiency of this CREC-based implementation and the significant gain of speed over a PC-based implementation.**

I. INTRODUCTION

The Smith-Waterman (SW) algorithm [1] is a very widely used algorithm in Bioinformatics. It represents an optimal method for sequence alignment and homology searches in genetic databases and makes all pair wise comparisons between the two strings. It achieves high sensitivity as all the matched and near-matched pairs are detected. In this field, the most commonly used algorithms are FASTA and BLASTA [2]. These are fast algorithms that prune the search involved in a sequence alignment using heuristic methods, but the level of errors yielded by these algorithms is about two times higher than the one provided by the SW algorithm [2].

However, the computation time required strongly limits the use of the SW algorithm. This is why, several hardware (ASIC and FPGA-based) implementations of this algorithm were proposed, but these implementations handle only a particular case of the algorithm, where the three parameters are assumed to have the following values: *ins* = *del* = 1, *sub* = 2 (see Section 2 for details), and there are only four characters in the alphabet (corresponding to the four nucleotides in the DNA: *A*, *C*, *G* and *T*) – as it is typical for many applications. For this particular case, Lipton and Lopresti [3] made an observation that considerably reduced the complexity of the algorithm, thus allowing high-performance implementations.

The goal of this paper is to present an application of the CREC General-Purpose Reconfigurable Computer consisting in the implementation of a general case of the SW algorithm, which seems to become a common need. The three parameters *ins*, *del* and *sub* can take any value between 0 and 15, and there are 32 characters in the alphabet. Here, the main limitation is given by the capacity of the FPGA chip. The physical support of the implementation was chosen to be a Xilinx Virtex FPGA device. Because of the generality of the considered case, the architecture occupies significantly more space in the FPGA chip than in the previous implementations. Starting from a systolic cell proposed by Yu et. al. [4], an original solution that lifts up the scalability issues is presented.

The main idea of the CREC design is to build a low-cost GPRC able to exploit the intrinsic parallelism present in many low-level applications by generating the best-suited hardware for implementing a specific application. The CREC design was introduced for the first time in [5]; the main novelty introduced by CREC consists of the combination, in a very effective way, of several design styles and architectural concepts. The result is a new computational model based on reconfigurable architecture concepts and whose main features are: *Instruction Level Parallelism*, *parallel RISC style architecture*, *dynamically generated fully scalable architecture*. The results presented in this paper prove the applicability of the CREC system for specialized computing intensive applications, approaching the performances obtained by dedicated systolic solutions, even if it is a general-purpose computer system.

II. SMITH-WATERMAN ALGORITHM

Pattern-matching problems appear in many different disciplines. Algorithms designed to solve the pattern-matching problem have evolved over time. This section discusses the Smith-Waterman algorithm (SW).

Smith and Waterman devised an algorithm for matching similar patterns. This algorithm compares a pattern *S* to a text *T* and calculates the penalty required to change *S* into *T*. Due to the fact that *S* and *T* do not match exactly, the penalty will take into account the number of insertions, deletions, and substitutions needed to convert the strings to match each other. This penalty is referred to as the edit distance.

The SW algorithm solves the matching problem using a dynamic programming approach. The algorithm uses the solutions from smaller problems to create the larger solution and in the process, creates a matrix of edit distances. If H is the SW matrix, the value in cell $H_{i,j}$ represents the degree of similarity between the sequences up to T_i and S_j. Lipton and Lopresti [3] presented a simplification of the SW algorithm. In the modified SW algorithm, each element of the matrix uses three other elements to compute its value:

$$d = \min \begin{cases} \begin{cases} a, & \text{if } T_i = S_j \\ a + sub, & \text{if } T_i \neq S_j \end{cases} \\ b + ins \\ c + del \end{cases}$$

	S_1		S_j
	0	1 ... j-1	j
T_1	1		⋮
	⋮		
	i-1	... a	b
T_i	i	c	d

Fig. 1. The SW formula and the internal table

If S and T have the lengths m and n respectively, then the time complexity of a serial implementation of the Smith-Waterman algorithm is $O(m \times n)$.

In a parallel implementation, the positive slope diagonal entries of Fig. 2 can be computed simultaneously. The final edit distance between the two strings appears in the bottom right table entry. Data dependencies mean that entries d in the table can only be calculated if the corresponding a, b and c values are already known and so the computation of the table spreads out.

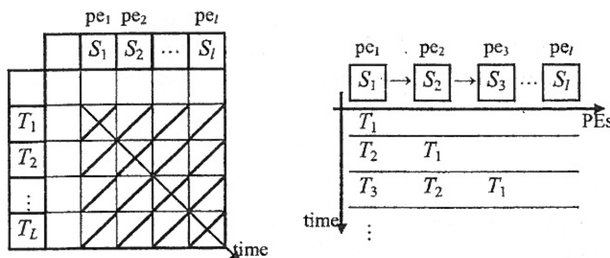

Fig. 2. Data dependencies in the SW table and the systolic operating mode

This systolic approach was used for implementing the SW algorithm using the CREC general-purpose reconfigurable computing system.

III. THE CREC DESIGN FLOW

CREC is the final product of a Hardware/Software CoDesign process, where the hardware part is dynamically and automatically generated during compilation. The resulting architecture is optimal because it exploits the intrinsic application parallelism. The steps in the application development are (Fig. 3):

1. The application's source code is written in *CREC assembly language*.
2. The source code is compiled using a *parallel compiler*, which allows the implementation of ILP (instruction-level parallelism). The compiler detects and analyses *data dependencies*, then it determines which instructions can be executed in parallel. A collection of instructions that can be executed in parallel constitutes a *program slice*. Thus, the whole program is divided into slices.
3. According to the slice's size, the hardware structure will be generated. The generic architecture already exists in a VHDL source file, so at this moment it is only adjusted. It will be materialized in an FPGA device.
4. The VHDL file is compiled and the FPGA device is configured.

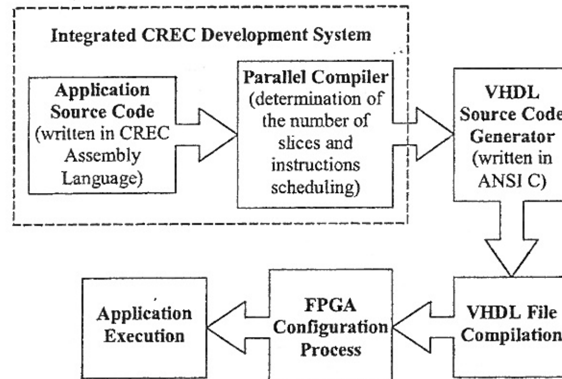

Fig. 3. The CREC Design Flow

IV. THE CREC LOW-LEVEL PARALLEL COMPILER

What particularizes the CREC compiler is the role of the last stage, *code generation.* Besides assigning the instructions to EUs, the *Low-Level Parallel Compiler* also determines the sets of instructions that can be executed in parallel, according to the hardware architecture. The algorithm that realizes this feature using the intrinsic parallelism of instructions has a linear execution time, proportional to the number of program's instructions, $O(N)$.

The compiler's task is to divide a CREC assembly language program written in a sequential manner into pieces to be executed in parallel. The compiler generates a file in a specific format that describes the tailored CREC architecture using the expanded form of program's instructions resulted from the previous phase. This file will include the size of the various functional parts, the subset of instructions involved, the number of EUs, etc., together with the sequence of instructions that makes up the program.

Another aspect that makes CREC particular with respect to classic processors, besides its reconfigurable nature, is the parallel nature of the computations: each EU has its own accumulator register. Thus, at each moment during execution, there are N distinct EUs, each one executing the instruction that was assigned to it by the compiler, taking into account its nature and the rules associated with instruction scheduling. Some instructions specify the precise EU to execute them, while other instructions can be executed by any EU. For instance, "*MOV R1,7*" will be executed by EU1, since it works with the register R1, while "*JMP 3*" has no specific EU associated and will be assigned one by the compiler, depending on the availability.

The accumulator register of all the EUs have equal capacities, *but the internal structure of each EU will be*

different, according to the subset of instructions (from the CREC Instruction Set) that the EU will actually execute.

The *scheduling algorithm* groups instructions so that they can be executed in parallel. A group of instructions that are executed at the same time is called a *slice*.

V. CREC HARDWARE ARCHITECTURE

The hardware structure is described using VHDL code, which is generated and optimized by a package of programs. The optimization is done by the VHDL Source Code Generator and consists of eliminating each unnecessary element, signal or bus.

The architecture's main components are: the N EUs; the N local configuration memories for the N EUs (in DPGA style), called *Instructions Memories*; a *Data Stack Memory*, used in instructions like PUSH or POP; a *Slice Stack Memory*, used to store the current slice address (CALL, RETURN); a *Slice Program Counter*; an associative memory that maps instructions to the slices that must be executed by each EU, called *Slice Memory*; a *Store Buffer* and a *Load Buffer* (temporary data buffers for the *Data Memory*); a *Data Memory*; *Operand Memories*, which contain the direct operands for the EUs.

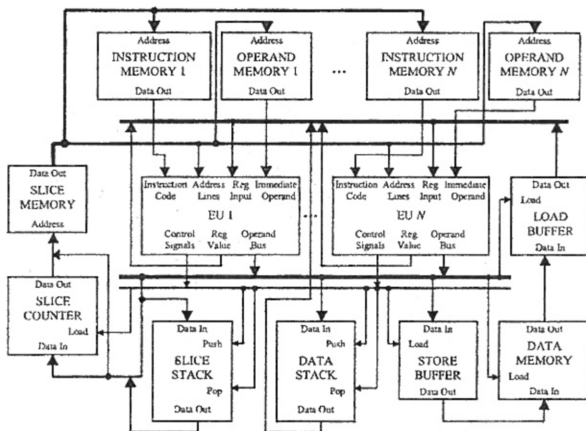

Fig. 4. The general CREC architecture

The linkage between the basic hardware elements is shown on Fig. 4. The *links between EUs are point-to-point*, but the Data Memory, the Slice Counter and the Slice Stack Memory are accessed via Address, Data and Control busses. The Direct Operands Memory can be accessed only by its corresponding EU.

The *Slice Memory* is an associative memory that stores the general slice word, which contains the pointers in the each Instruction Memory and operand Memory. According to the number of instructions that each EU performs in the current application, the widths of these two fields are variable and different for each EU. This way, the word width is variable with each application and will always be kept to a minimum. This memory is implemented in the Virtex BlockRAMs.

The *Instructions Memory* and the *Direct Operands Memory* are *distributed* memory blocks, implemented in the Virtex Look-up Tables configured as ROMs. Their size depends on the number of instructions and on the number of direct operands that each EU works with during the application execution.

The general *Data Stack Memory* is implemented as RAM. Its size is variable and will be estimated according to the number of PUSH / POP instructions present in the application source code. The *Slice Stack Memory* is used by the general instructions CALL and RETURN. Here is stored the slice number for a procedure call and the return slice address from a procedure call.

The *Data Memory* is normally implemented outside the FPGA chip, but for simulation purposes we created this block inside the Virtex chip. This memory contains the usual data used in the application.

VI. CREC INSTRUCTION SET

Each instruction is encoded on the same number of bits, like in RISC architectures. Although CREC is a RISC processor, its EU has a relatively large instruction set, making it attractive for a wide range of applications. The instruction set is divided in the *Data Manipulation* and the *Program Control Groups*. The Data Manipulation Group contains the specific instructions for manipulating the value of the EU's accumulator. Each instruction performs operations on unsigned numbers. The Program Control Group contains the instructions for altering the program execution.

The EU can perform the following *Data* Manipulation instructions: *Addition* with/without carry and *Subtraction* with/without borrow and *Compare*; Logical functions: *And, Or, Xor, Not* and *Bit test*; *Shift arithmetic* and *logic* left/right; *Rotate* and *rotate through carry* left/right; *Increment/decrement* the accumulator and negation.

The Program Control instructions are: Slice counter manipulation: *Jump*, *Call* and *Return*; *Data movement*; Data-Stack manipulation: *Push* and *Pop*; *Input from* and *Output to port*; *Load* from and *Store* in the Data memory. Each program control instruction is conditioned, thus offering a great flexibility. This way, the source code can be optimized, because most of the Compare and Jump statements can be replaced by conditional instructions (ex. conditioned Move).

VII. THE CREC MAIN EXECUTION UNIT

The main part of the CREC processor is the scalable EU. The word length of the EU is $n \times 4$ bits. At the current state of the implementation, the parameter n is limited to 4, so the word length can be up to 16 bits. The complete structure of the EU is presented on Fig. 5.

There are two variants of the CREC EU implementation, but from the functional point of view they are absolutely the same. In the first one, each subunit is strongly optimized for the Xilinx VirtexE FPGA family, occupying the same number of Virtex Slices ($2 \times n$ Slices) and using the dedicated Fast Carry Logic. This leads to a platform-dependent solution, but there was the need to increase the performance of the EU and to obtain almost equal propagation times. The second variant uses a general VHDL code, not optimized for any FPGA devices family.

This increases CREC's portability, but the architectural optimizations become the VHDL compiler's task.

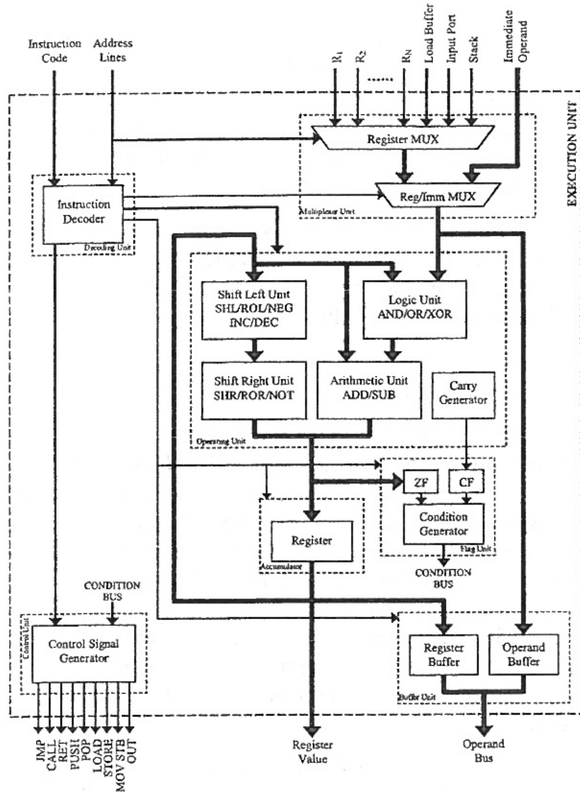

Fig. 5. The basic CREC Execution Unit

The EU has six major parts: *Decoding Unit* (decodes the instruction code); *Control Unit* (generates the control signals for the Program Control Group); *Multiplexer Unit* (selects the second operand of binary instructions); *Operating Unit* (implements the data manipulating operations); *Accumulator Unit*; *Flag Unit* (contains the two flags: Carry and Zero) and the *Buffer Unit* (interfaces the EU with the processor's internal data buses).

The *Operating Unit* is organized in a symmetrical way. At the right side are the binary operation blocks, and at the left side are the unary operation blocks. Its four blocks are: the *Logic Unit*, the *Arithmetic Unit*, the *Shift Left Unit* and the *Shift Right Unit*.

The output of the *Flag Unit* is a 6-bits wide Condition Bus for the six possible condition cases: *Zero*, *Not Zero*, *Carry*, *Not Carry*, *Above* and *Below or Equal*. This bus validates the conditioned Program Control instructions.

The *Multiplexer Unit* is built on two levels for optimal instruction encoding. At the first level, the Register and the Data MUXs have the same selector. The second 2:1 multiplexer selects the input operand for the instruction. This Unit is also customizable: only those input lines would be implemented, which are actually used by the EU. In Virtex FPGAs, only 8-to-1 multiplexers can be implemented on a single level of CLBs. For this reason, the multiplexer is optimized for up to 8 inputs. For CREC architecture with more than 8 EUs, the multiplexers are implemented on two levels of logic. This disadvantage can be overcome by using FPGAs containing wider multiplexers. The most important aspect is that this unit's size increases linearly with the increase of the word length and the number of EUs.

The *Accumulator Unit* stores the primary operand also the result of each Data Manipulation instruction.

The *Decoding Unit* generates the appropriate signals for the four functional parts of the Operating Unit and for the Carry Generator. The *Control Unit* generates the validation signals for the Program Control instructions taking into account the Condition Bus. The condition code is compared against the value of the flags generated by the previous non-program control instruction.

The *Buffer Unit* consists of two simple tri-state buffers, to select the operands for the Program Control instructions.

The Execution Unit is customizable. For example, if an EU will not execute any Logical Instruction, then this part is simply cut out, resulting in a gain of space. All four units use the same number of Virtex slices. For this reason, the size of the Operating Unit is growing *linearly* with the word length. But the operating time will not decrease significantly, because the number of CLB levels is constant. The EU can be easily pipelined for higher frequency operation.

VIII. THE IMPLEMENTATION OF THE SMITH-WATERMAN ALGORITHM

Because of the initial conditions (the parameters *ins*, *del* and *sub* can take any value between 0 and 15), the size of the generated results strongly increases and it is necessary to store them on 16 bits, in order to cover the worst case.

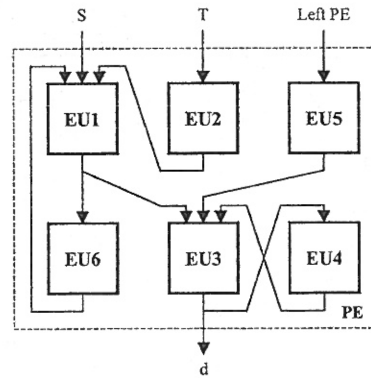

Fig. 6. The structure of a Processing Element

In the implementation of the Smith-Waterman algorithm, 6 EUs of 16 bits were considered to form one *Smith-Waterman Processing Element* (PE) as shown on Fig. 6. The register associations are as follows: $S \rightarrow$ R1, $T \rightarrow$ R2, $a \rightarrow$ R3, $b \rightarrow$ R4, $c \rightarrow$ R5 and an additional R6 register for temporarily storing the value of S. The portion of code for one PE is presented in Fig. 7. The output value of the left PE is symbolized with R?, because the index of this register will be known only in the case of the fully written source code. For example the 1st PE has the registers from R1 to R6 as shown on the Fig. 6. The 2nd PE will have the next six registers, namely the R7 through R12. The 3rd PE will have R13 to R18, and so on. This way the Left PE entry for the 2nd PE will be derived from R3 and for the 3rd PE from R9.

The first 4 instructions present the updating phase of each cycle, when the values/parameters are shifted.

```
[1]  MOV R2,P2     ; Read T from Port 2
[2]  MOV R4,R3     ; b ← d
[3]  MOV R3,R5     ; a ← c
[4]  MOV R5,R?     ; c ← d_Left PE
[5]  MOV R6,R1
[6]  CMP R1,R2     ; T_i = S_j ?
[7]  ME  R1,0      ; Move if Equal
[8]  MNE R1,SUB    ; Move if Not Equal
[9]  ADD R3,R1     ; a+SUB
[10] MOV R1,R6
[11] ADD R4,INS    ; b+INS
[12] ADD R5,DEL    ; c+DEL
[13] CMP R3,R4     ; Calculation of d
[14] MA  R3,R4     ; Move if Above
[15] CMP R3,R5
[16] MA  R3,R5     ; The minimum of 3 numbers
[17] SUB R4,INS    ; Restoring the b and c
[18] SUB R5,DEL    ; values
```

Fig. 7. The task of one Processing Element

The slice mapping of the above-presented algorithm is shown on the Table 1. A 2:1 compression ratio was obtained. The number of program slices remains constant and does not depend on the number of PEs, due to the independent and parallel work of the EUs.

TABLE 1. SLICE MAPPING OF THE PE'S TASK

#	EU1	EU2	EU3	EU4	EU5	EU6
1		mov r2,p2	mov r3,r5	mov r4,r3	mov r5,r?	mov r6,r1
2	cmp r1,r2			add r4,ins	add r5,del	
3	me r1,0					
4	mne r1,sub					
5	mov r1,r6		add r3,r1			
6			cmp r3,r4			
7			ma r3,r4	sub r4,ins		
8			cmp r3,r5			
9			ma r3,r5		sub r5,del	

The whole structure is presented on the Fig. 8. The 3 FIFO memories are implemented in the internal BlockRAMs of the FPGA devices taking advantage of the dual porting possibility. The *S* and the *F* strings are initialized by the host computer and the results are stored in the *d* parameters array. This array must be filled initially with the *c* parameters.

The FIFO memories are connected to the Processing Elements by means of the Input and Output Ports of the appropriate Execution Units.

The *ins*, *del* and *sub* parameters are implemented as simple registers and only one copy exists in the whole system. From the occupied space point of view in the FPGA chip this is a better approach than to store locally the value of these parameters for each *Processing Element*. This way the three parameters are very simple custom Execution Units, containing only the accumulator register and a few control parts.

Initially they are loaded with the corresponding parameter values, after they can deliver the numbers to the corresponding Execution Units of the Processing Elements. In the case of the 1st PE the customized *ins* Execution Unit is connected to the *EU4*, the *del* is linked with *EU5* and *sub* with *EU1*, as shown on Table 1. For the 2nd PE the connections are as follows: *ins* → *EU10*, *del* → *EU11* and *sub* → *EU7*, and so on for the further Processing Elements.

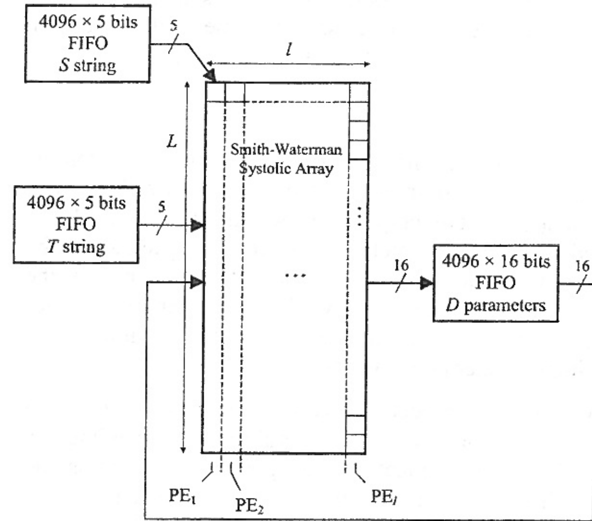

Fig. 8. The structure of the proposed solutions

IX. EXPERIMENTAL RESULTS

The algorithm was first implemented in software using Borland Delphi compiler on a PC station featuring an AMD Athlon XP 1700+ based system with 128 MB DDR SDRAM. The average execution time was 210 milliseconds.

Then, the algorithm was implemented using the methodology exposed above, in a CREC system. The physical support for this implementation was a VirtexE600 FPGA on a Nallatech Strathnuey© + Ballyderl© board.

The implementation was realized and simulated for several Xilinx FPGA chips. In a Spartan3 5000 chip, the maximal possible implementation occupies 8250 CLBs, which means a SW array composed of 33 PEs can be implemented. In a Virtex II 8000 chip, the maximal possible implementation occupies 11,500 CLBs, which means a SW array composed of 46 PEs can be implemented. Table 2 contains different statistics for several Xilinx FPGA devices:

TABLE 2. SPACE OCCUPATION AND PERFORMANCE

FPGA Device	Used CLBs	Number of PEs	Frequency	Execution time	Speedup ratio
SpartanII 600E	3,430	7	100 MHz	216 ms	0.97
Spartan3 1500	3,250	13	150 MHz	77 ms	2.73
Spartan3 4000	6,750	27	150 MHz	37 ms	5.68
Spartan3 5000	8,250	33	150 MHz	31 ms	6.77
Virtex 600E	3,430	7	100 MHz	216 ms	0.97
Virtex 3200E	15,680	32	100 MHz	47 ms	4.47
VirtexII 1000	1,250	5	150 MHz	201 ms	1.04
VirtexII 3000	3,500	14	150 MHz	72 ms	2.92
VirtexII 6000	8,250	33	150 MHz	31 ms	6.77
VirtexII 8000	11,500	46	150 MHz	22 ms	9.55

The CREC-based implementation runs at 100 MHz in a VirtexE 600 chip. The execution time estimations were performed by taking into consideration the computationally-intensive part of the algorithm, when all

the PEs are operational. In this case 7 PEs can work in parallel, and for a sequence of 4096 characters in the *T* stream 4096 × 9 = 36,864 clock cycles are necessary to perform the computations. But for 4096 characters in the *S* stream 4096 / 7 = 586 computation cycles are needed. This means that the total execution time will result in 21,570k × 10 ns = 215,700 μs = 216 ms.

X. CONCLUSIONS

In conclusion, the CREC-based implementation is approximately 2÷6 times faster than the PC-based one with an adequate FPGA chip. With the largest VirtexII chip a speedup of approximately one order of magnitude can be achieved. Further work will consist in optimizing the CREC implementation to work on higher frequencies (~200MHz) on Spartan3 and VirtexII families thus obtaining greater performances.

An important aspect of this implementation against the dedicated systolic one is the possibility of extending the algorithm. The application may pre or post process the data, different software modules may be added and implemented in the design thus achieving great flexibility with a few efforts.

This result is a new proof of the efficiency of the CREC general-purpose reconfigurable computer. It can be easily implemented also as an embedded system in different applications, where great performance is needed at lower price.

XI. REFERENCES

[1] Smith, T. F. and Waterman, M. S. "Identification of common molecular subsequence." *Journal of Molecular Biology*, 147, 1981, 196-197.

[2] National Center for Biotechnology Information. BLAST home page, 2003. http://www.ncbi.nlm.nih.gov/blast.

[3] Lipton, R. and Lopresti, D. "A systolic array for rapid string comparison", in *Proceedings of the Chapel Hill Conference on VLSI*, 1985, 363-376.

[4] Yu, C. W., Kwong, K. H., Lee, K. H., Leong, P. H. W. "A Smith-Waterman Systolic Cell", in *Proceedings of the 13th Conference on Field-Programmable Logic and Applications (FPL2003)*, Springer-Verlag Publishing House, Lisbon, September 2003, 375-384.

[5] Cret, O., Pusztai, K., Vancea, C. Szente, B., "CREC: A Novel reconfigurable Computing Design Methodology", in *Proceedings of the 17th IPDPS*, Nice, France, April 2003, 175.

[6] DeHon, A. "Reconfigurable Architectures for General-Purpose Computing". *PhD. Thesis*, MIT, 1996.

[7] Szente, B., Vancea, C., Uiorean, L., Rusu, F., Cret, O., Pusztai, K. "The CREC General Purpose Reconfigurable Computer", in *Proceedings of the IP Based SoC Design Conference*, Grenoble, France, November 2003, 217-222.

[8] Singh, H., Lee, M. H., Lu, G., Kurdahi, F., Baghrzadeh, N., Chaves Filho, E. "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications". *IEEE Transactions on Computers*, Vol. 49, 5, May 2000.

[9] Green, C. and Franklin, P. "RaPiD - reconfigurable pipelined datapath", in R. W. Hartenstein and M. Glesner, editors, *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers. 6th International Workshop on Field-Programmable Logic and Applications*, Darmstadt, Germany, September 1996, 126-135.