Implementation of the Smith-Waterman Algorithm on A Reconfigurable Supercomputing Platform

Peiheng ZHANG, Guangming TAN, Guang R. GAO
peiheng.zhang@gmail.com, {guangmin, ggao}@capsl.udel.edu
CAPSL, University of Delaware, April 16th 2007

Abstract

An innovative reconfigurable supercomputing platform – XD1000 is being developed by XtremeData to exploit the rapid progress of FPGA technology and the high-performance of Hyper-Transport interconnection. In this paper, we present implementations of the Smith-Waterman algorithm for both DNA and protein sequences on the platform. The main features include: (1) we bring forward a multistage PE (processing element) design which significantly reduces the FPGA resource usage and hence allows more parallelism to be exploited; (2) our design features a pipelined control mechanism with uneven stage latencies – a key to minimize the overall PE pipeline cycle time; (3) we also present a compressed substitution matrix storage structure, resulting in substantial decrease of the on-chip SRAM usage. Finally, we implement a 384-PE systolic array running at 66.7MHz, which can achieve 25.6GCUPS peak performance. Compared with the 2.2GHz AMD Opteron host processor, the FPGA coprocessor results in speedup of 185 and 250 respectively.

1. Introduction

The XD1000 is an innovative reconfigurable supercomputing platform developed by XtremeData Inc. [20]. Taking advantage of the rapid progress of FPGA technology and the high-performance of Hyper-Transport interconnection that provide efficient link between a main processor with a FPGA coprocessor, the XD1000 provides an ideal and cost effective acceleration platform for many algorithms. As shown in Figure 1, the XD1000 integrated a leading edge Altera StratixII FPGA into a dual Opteron based system. The FPGA coprocessor module can be inserted directly into an Opteron 940 socket and uses the motherboard's existing CPU infrastructure to create a full featured environment for FPGA based reconfigurable computing coprocessor functions. The FPGA coprocessor connects directly to the CPU's HyperTransport bus and the DIMM slots on the motherboard while utilizing the existing power supply and heat sink solution for the CPU. The high-bandwidth, low-latency HyperTransport link between the XD1000 coprocessor and the Opteron CPU enables tightly-coupled FPGA acceleration of X86 applications previously impossible with legacy PCI-bus based solutions [9].

Based on the XD1000 platform, we present implementations of the Smith-Waterman algorithm for both DNA and

protein sequences. To squeeze the maximum performance from the FPGA, (1) we bring forward a multistage PE (processing element) design which significantly reduces the FPGA resource usage and hence allows more parallelism to be exploited; (2) our design features a pipelined control mechanism with uneven stage latencies – a key to minimize the overall PE pipeline cycle time; (3) we also present a compressed substitution matrix storage structure, resulting in substantial decrease of the on-chip SRAM usage. Using these methods, we implement a 384-PE systolic PE array operating at 66.7MHz, which can achieve a peak performance of 25.6GCUPS. Compared with the 2.2GHz AMD Opteron host processor, the FPGA could gain 185 and 250 times speedup respectively.

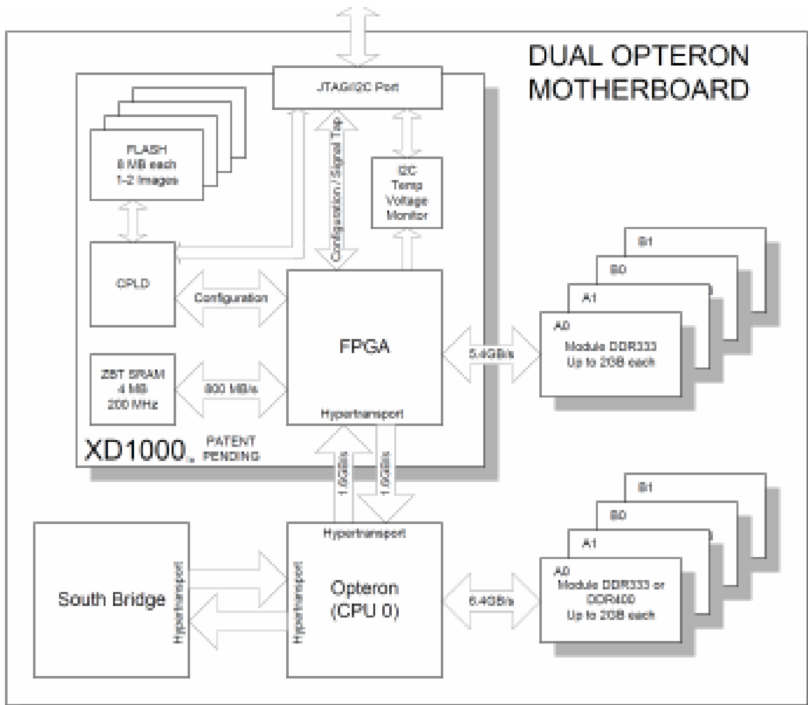


Figure 1. The XD1000 platform block diagram

The rest of this paper is organized as follows. The next section provides an overview of the Smith-Waterman algorithm and how to map the algorithm to a systolic array. A detailed discussion of how to design the processing element is presented in Section 3. Section 4 introduces the compressed substitution matrix storage method. A general structure of the computing core logic is given in section 5, while performance results are shown in Section 6. Relevant conclusions are presented in Section 7.

2. Smith-Waterman algorithm and systolic processing element array

The Smith-Waterman algorithm is a well-known dynamic programming algorithm for performing local sequence alignment for determining similar regions between two DNA or protein sequences. The algorithm was first proposed by T. Smith and M. Waterman in 1981. Nowadays it is still a core algorithm of many applications [18].

The algorithm mainly consists of two steps: (1) Calculate the similarity matrix score; (2) According to the dynamic programming method, trace back the similarity matrix to search for the optimal alignment. In the algorithm, the first step will consume the largest part of the total calculation time. The definition of Smith-Waterman algorithm is shown as below:

For two sequences *S* and *T*, the length of *S* is *n*, $|S| = n$; the length of *T* is *m*, $|T| = m$; *V(i,j)* is the optimal alignment score of two sub-sequence *S[1]…S[i]* and *T[1]…T[j]*, the calculation of *V(i,j)* is defined as formula (2.1) and (2.2):

Initialization:

$$\begin{cases} V(i,0) = 0, & 0 \le i \le n; \\ V(0,j) = 0, & 0 \le j \le m; \end{cases} \tag{2.1}$$

Recursion relation:

$$V(i,j) = \max \begin{cases} 0 \\ V(i-1,j-1) + \sigma(S[i],T[j]) \\ V(i-1,j) + \sigma(S[i],-) \\ V(i,j-1) + \sigma(-,T[j]) \end{cases}, \qquad 1 \le i \le n, 1 \le j \le m \tag{2.2}$$

In the formulas, a "–" stands for a null character or gap; *V(i,0)* stands for the result of comparing each character in *S* with a gap in *T*, the definition of *V(0,j)* is the counterpart of comparing each character in *T* with a gap in *S*; *σ(S[i],T[j])* is the value of substitution matrix.

	-	S1	S2	S3	S4	S5	…
-	0	0	0	0	0	0	
T1	0	L	L	L	L	L	L
T2	0	L	L	L	L	L	L
T3	0	L	L	L	L	L	L
T4	0	L	L	L	L	L	L
T5	0	L	L	L	L	L	L
…		L	L	L	L	L	

Figure 2. Similarity matrix calculating sequence and data dependency

While calculating the similarity matrix, the score of any matrix element *V(i,j)* always depends on the score of other three elements: (1) The up-left neighbor element *V(i-1,j-1)*; (2) The left neighbor *V(i,j-1)*; (3) The up neighbor *V(i-1,j)*. Therefore the calculation sequence of the similarity matrix will be shown as Figure 2. It begins from the top-left element to bottom-right element according to the direction as shown as the arrow. Through observation of the similarity matrix calculation process, we can find that for each clock cycle, every element on an anti-diagonal line marked with same number

could be calculated simultaneously, the ①②③④⑤⑥⑦⑧⑨⑩ stand for the elements that could be calculated at same time. For example, in the first cycle, only one element marked as ① could be calculated; In the second cycle, two elements marked as ② could be calculated; In the third cycle, three elements marked as ③ could be calculated, etc., and this feature implies that the algorithm has a very good potential parallelity.

To further describe the level of similarity between two real bioinformatics sequences, an affine gap model is introduced to the Smith-Waterman algorithm by O. Gotoh in 1982 [4]. In the affine gap model, gap is used to compensate for the insertion or deletion, to make the alignment more condensed in satisfying an expecting model. Gap is usually a consecutive null character string in a sequence and should be as long as possible. In the affine gap model, the penalty score (or cost) for the first gap is called *gap-open*, the cost for the following gaps is called *gap-extension*. According to the affine gap model, the formulas to calculate the similarity matrix are described as below:

Initialization:

$$\begin{cases} V(i,\ 0) = E(i,\ 0) = 0, & 0 \le i \le n; \\ V(0,\ j) = F(0,\ j) = 0, & 0 \le j \le m; \end{cases} \tag{2.3}$$

Recursion relation:

$$V(i,j) = \max \begin{cases} 0 \\ E(i,j) \\ F(i,j) \\ V(i-1,j-1) + \sigma(S[i],T[j]) \end{cases} ,1 \le i \le n, 1 \le j \le m; \tag{2.4}$$

$$E(i,j) = \max \begin{cases} V(i,j-1) - \alpha, \\ E(i,j-1) - \beta, \end{cases} \quad 1 \le i \le n, 1 \le j \le m; \tag{2.5}$$

$$F(i,j) = \max \begin{cases} V(i-1,j) - \alpha, \\ F(i-1,j) - \beta, \end{cases} \quad 1 \le i \le n, 1 \le j \le m; \tag{2.6}$$

In the formulas, α stands for the *gap_open*, β stands for the *gap_extension*. $E(i,j)$ and $F(i,j)$ are the maxima of the following two items: open a new gap or keep extending an existing gap.

Systolic array was firstly introduced by H. T. Kung and C. E. Leiserson in their papers [10] [11], and it was proved very efficient in the computing of matrix multiplication or LU-decomposition. Further research revealed that dynamic programming algorithm also mapped very well on to a systolic array due to its potential parallelity [12].

In 1985, R. J. Lipton and D. P. Lopresti mapped the edit distance algorithm, a typical dynamic programming algorithm for the global alignment of DNA sequences, to a systolic array by implementing an nMOS prototype chip [13]. The preliminary results of the prototype showed hundreds to thousands times faster than a contemporary minicomputer. In 1987, D.P. Lopresti built the first systolic array system for comparing nucleic acid sequences–*P-NAC* [14]. Based on these works, many systolic array systems were developed, including *BISP* of CalTech [3], *BioSCAN* of UNC [21], *B-SYS*, *Splash*

/*Splash-2* of Brown Univ. [8] [6] [7], *Kestrel* of UCSC [5], etc.

In recent years, along with the rapid progress of bioinformatics and FPGA technology, some new systems were developed for both commercial and research purpose, including *TimeLogic DeCypher* [19], *CLC Bioinformatics Cube* [2], and the *Hyper Customized Processors for Bio-Sequence Database Scanning* of NTU [16] [17], etc.

Based on these works, we present our implementations of the Smith-Waterman algorithm for both DNA and protein sequences on an innovative reconfigurable supercomputing platform – XD1000. Compared with these works, from the perspective of application, our design extends the sequence length limit to 64KBp, which will satisfy the requirement of various applications; In the Smith-Waterman algorithm design for DNA sequence, there are four software programmable parameters, which allow the hardware implementation compatible with the existing software programs, including both linear and affine gap model algorithms; In the Smith-Waterman algorithm design for protein sequence, the substitution matrix is also reconfigurable, which allow users to choose from the different evolution models or develop their own evolution models. From the perspective of hardware architecture, we present a new multistage PE design and a compressed substitution matrix storage method, which result significant decrease of the FPGA resources usage and hence allows more parallelism to be exploited from the FPGA.

In our design, we map a systolic PE array to an anti-diagonal line of the score matrix as shown in Figure 3. For instance, in the 5th clock cycle, the PE array is mapped to calculate the elements marked with ⑤, and in the next cycle, the PE array will be mapped to calculate the elements marked with ⑥. In the following sections, we will discuss in detail how to implement a PE.

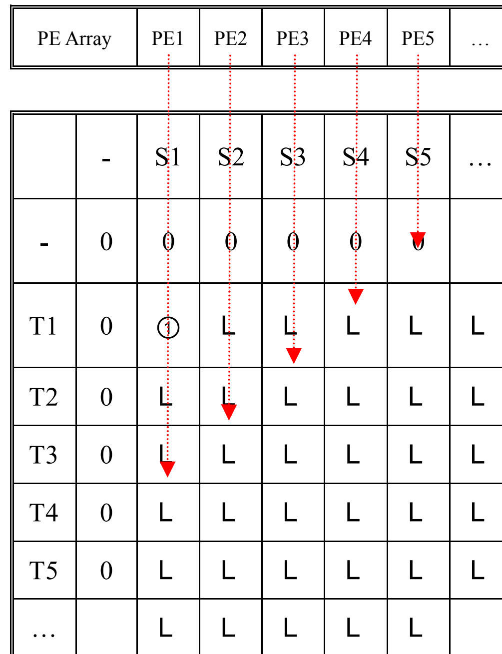


Figure 3. Mapping the Smith-Waterman algorithm to a systolic PE array

Due to the hardware resource limitation, we can only implement a limited number of PE on the FPGA. Thus, in the calculation of a similarity matrix, we need to divide the matrix into sub-matrices, in each iteration, the PE array will calculate one sub-matrix, and store the intermediate results in the memory for the next iteration to use.

As shown in Figure 4, a systolic PE array consists of many identical cascading PE. Before the start of the calculation, sequence *S* should be shifted into the array under the control of the *Move_in_S* signal; The *init_in* signal to each PE decide whether or not this PE will join in the calculation; Sequence *T* is synchronous to *init_in* when entering into the PE array; The *mid_in* is used to feed back the temporary intermediate data to the PE array when multi-iteration calculation is needed.

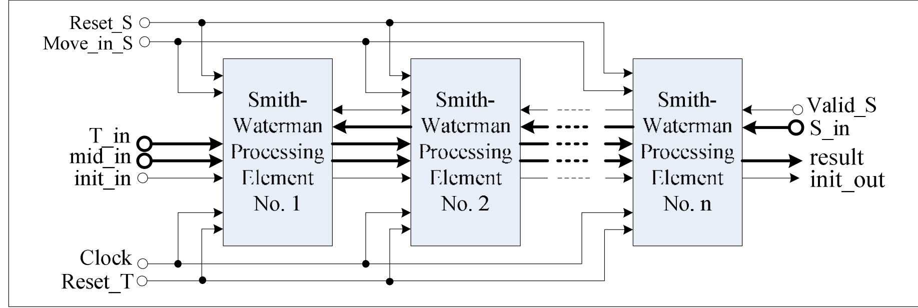


Figure 4. Systolic PE array of the Smith-Waterman algorithm

A feature of our design laid in that the shift in direction of sequence *S* is opposite to that of sequence *T*, this guarantees that sequence *S* will be stored in the PE array as the original sequence, which means the tail of the sequence will always be stored in the last PE. This method not only facilitates the software process of preparing data, but also guarantees the computing of the score matrix is continuous when multi-iterations are needed.

3. The Smith-Waterman algorithm processing element design

From the formulas of (2.3) through (2.6), a straightforward PE schematic was proposed as shown in Figure 5 [16] [17], the functions of each DFF (D type flip-flop) are detailed as below:

- ✧ *S-Out* and *T-out* DFF are used to store *S[i]* and *T[j]*;
- ✧ *E-out* DFF is used to store *E(i,j)*, and it will be used by the same PE in the next clock cycle. Its inputs come from the same PE which was generated in previous clock cycle, representing the values in its upper neighbor element;
- ✧ *F-out* DFF is used to store *F(i,j)*, and it will be used by the next PE. Its inputs come from the previous PE, representing the values in its left neighbor element;
- ✧ The input of *V-diag* DFF comes from the previous PE, and it is registered for one cycle before it is used by the PE, so it represents the value of its up-left neighbor element;
- ✧ *V-out* DFF is used to store *V(i,j)*;
- ✧ *Max_out* DFF is used to store the maximum value of the similarity matrix. It has three inputs: (1) The maximum value coming from previous PE; (2) *V(i,j)* coming from current PE; (3) The maximum value stored in itself.

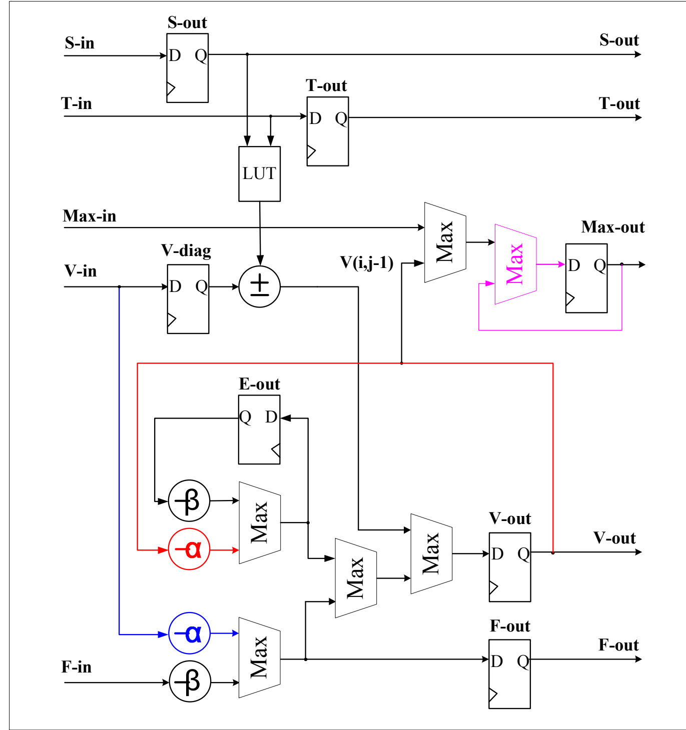


Figure 5. A straightforward Smith-Waterman algorithm PE design

Before the hardware implementation, we need to estimate the quantity of FPGA resource used by the PE design. The PE data width should be decided by the maximum sequence length and the maximum value in the substitution matrix. For example, if the length is 64KBp and the maximum value is 11, then the PE data width should be at least 20 bits, which means $2^{20} > 64K*11$. In the straightforward PE design, there are five add/sub operations and six max operations. Because each max operation consists of a subtraction and a 2:1 Mux operation, there are eleven add/sub and six 2:1 Mux operations in total for a PE. If the data width is set to 20 bits, a PE will cost about 340 ALUT (Adaptive Look-Up Table).

Altera EP2S180 FPGA has 143520 ALUT in total, the Hyper-Transport interface and some other logic will cost 8% resource, normally the maximum FPGA resource utilization is less than 90%, so we can put at most 340 PE. Considering the cost of other necessary control logic modules, the number of PE we can implement will less than 270 if we simply adopt the straightforward PE design. Therefore we need to endeavor to reduce the resource cost of the PE, so that we can put as many PE on the FPGA as possible.

3.1 Simplify the Max-out operation of *V(i,j)*

In the PE design, the *Max-out* DFF is used to store the maximum value of the similarity matrix. To get the maximum value, for the first step, it needs to compare the maximum output from previous PE with *V(i,j)* of the current PE. It stands for the comparison between the horizontal neighbor matrix elements; For the second step, it needs to compare the output from the first step with the value stored in the *Max-out* DFF itself, it stands for the comparison between the vertical

neighbor matrix elements. The result will be stored back to the *Max-out* DFF itself again after the two steps.

Considering that at the end of the calculation, all the maximum values stored in each *Max-out* DFF will shift out of the array, we can move the second comparison step outside of the array, while the first comparison step is kept doing by each PE. This way, we can delete a max operation from the PE (The purple Max block).

3.2 Simplify the operation of "*V(i,j) – α*"

When we cascade multiple PEs to form a PE array as shown in Figure 6, we will find that the output of *"V-out – α"* in the left PE (red sub block) is identical to the *"V-in – α"* of the right PE (blue sub block). Therefore, we can delete the *"V-in – α"* from the PE design. What we need to do is add a new output of the "*V-out – α*" called "*V-out-Alpha*" and an input called *"V-in-Alpha"* instead of the *"V-in – α"*, when cascading the PE to form the PE array, we need to connect the *"V-out-Alpha"* signal of the left PE to the *"V-in-Alpha"* signal of the right PE.

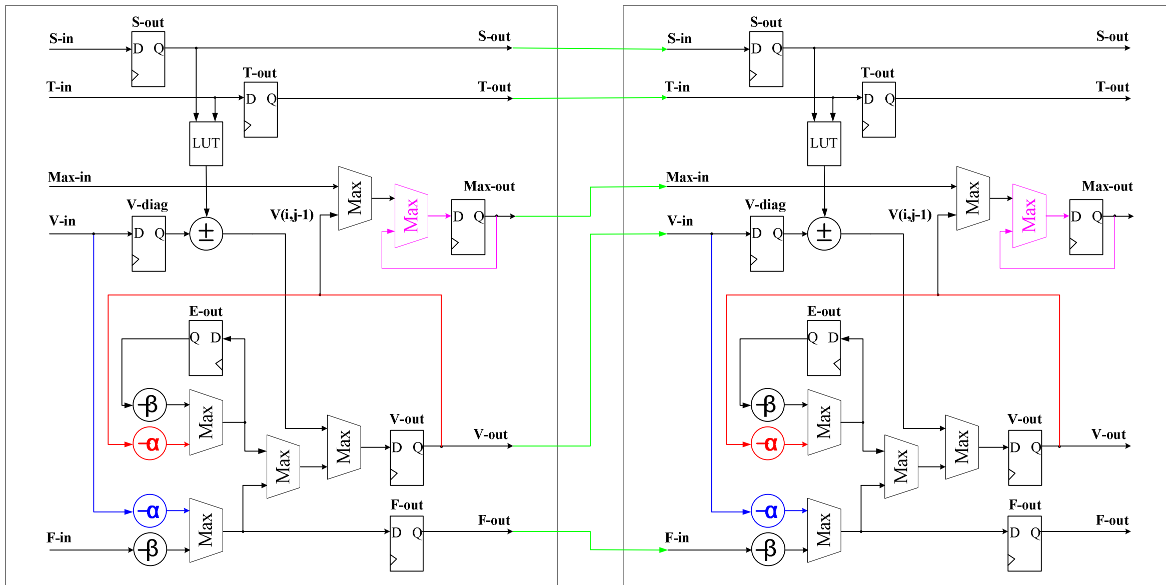


Figure 6. Cascading Smith-Waterman algorithm PE in an array

3.3 Compact the max operation

A max operation consists of a subtraction and a 2:1 Mux operation. For example, "$R = (X < Y)?\ Y : X$" will translate to two equations – "$Temp = X - Y;\quad R = Sign_Bit_Temp?\ Y : X$" when it is synthesized into a FPGA. Therefore, the max operation of two 20 bit data will cost 40 ALUT. 20 ALUT are used for the subtraction, another 20 ALUT are used for the 2:1 Mux operation, and both of the operations will be implemented by the ALUT. But indeed, only the sign bit of the result will be used as the select control signal for the 2:1 Mux operation. The difference itself will be discarded. In hardware implementation, the sign bit is identical to the *carry_out* of the MSB (most magnificent bit) of the subtraction.

Compared o its previous generation FPGA, StratixII made significant improvement in its ALM (Adaptive Logic Module) design [1]. While operating in arithmetic mode, the ALM can support simultaneous use of the adder's carry output along with combinational logic outputs. In this mode, the output of the adder is ignored. This usage of the adder with the combinational logic output provides resource savings of up to 50% for functions that can use this ability. A conditional operation, such as the max operation – “$R = (X < Y)?\ Y : X$” can fully use this feature of the ALM, as shown in Figure 7.

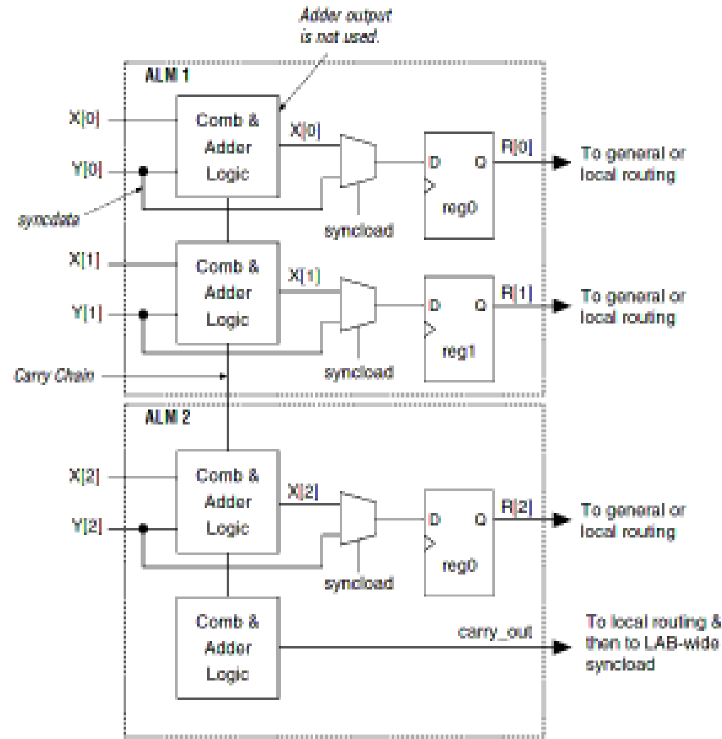


Figure 7. Max operation example

To implement this function, the adder is used to subtract *Y* from *X*. The *carry-out* signal is then fed to the LAB-wide *sync-load* signal. If *X* is less than *Y*, the carry-out signal is 1. The *sync-load* is asserted and selects the *sync-data* as input. In this case, the data *Y* drives the *sync-data* inputs to the registers. If *X* is greater than or equal to *Y*, the *sync-load* signal is de-asserted and *X* drives the data port of the registers.

A necessary prerequisite to compact the comparison operation and 2:1 Mux operation to an ALM is that the output of the 2:1 Mux only feeds to a DFF. In other words, we need a DFF immediately after the Max operation. By doing this, it appears that we will introduce additional logic resource usage, but in fact, because it improves the usage percentage in each ALM, it helps to save the general FPGA resources usage.

A problem arising from these introduced DFF is that the PE would require more than one clock cycle to finish the calculation of one matrix element, which will hinder the performance greatly. We will discuss this in the following sections.

3.4 Handling the negative number

When calculating the "*V(i-1,j-1)+σ(S[j],T[j])*", the result is probably negative due to the *σ(S[j],T[j])* is negative when *S[j]* is not equal to *T[j]*. Because all the max operations are based on unsigned numbers, we need to process the negative number before they come to the max function.

According to formula (2.4), all negative number will be reset to zero. That means that whenever a negative number is generated from the add/sub operation, we can reset it to zero unconditionally. Therefore, in the PE design, we introduce a DFF to store the result from the add/sub operation, and we use the MSB (the sign bit) of the add/sub result as a synchronous clear signal to the DFF. Thus, when a positive number is derived from the add/sub function, it will be stored into the DFF as it is; when a negative number is derived from the add/sub function, a zero will be stored into the DFF instead.

3.5 Multistage in the PE design

In Section 3.3 and 3.4, we introduced some DFFs to reduce the area cost and to process the negative numbers, but in the meantime, it also incurred more clock cycles to finish the calculation of a matrix element. To solve this problem, our design features a pipelined control mechanism with uneven stage latencies – a key to minimize the overall PE pipeline cycle time.

With the FPGA internal PLL (Phase Lock Loop), we generate 4 clocks with the same clock frequency but with different phase relationship, as shown in Figure 8. These clocks are connected to the DFFs as the requirement of the multistage PE design as shown in Figure 9.

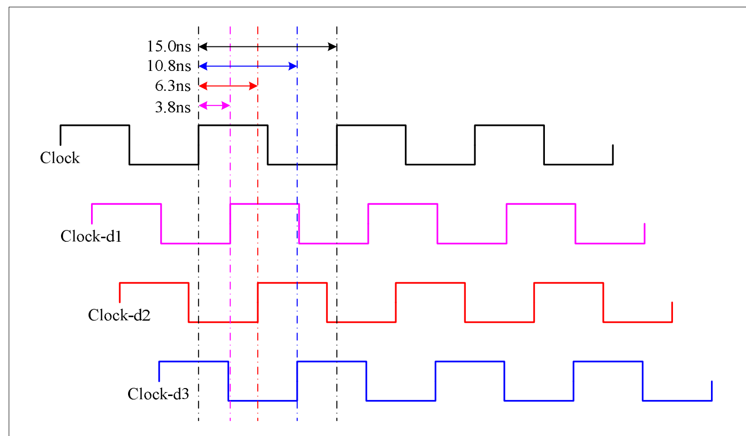


Figure 8. Clocks for the multistage PE design

The phase delays are decided by the timing simulation of the PE design. For example, the delay from *clock* to *clock-d2* is set to 6.3ns, because there is a subtraction operation and a max operation that need to finish during this period, and the longest data path is about 6ns; the delay from *clock-d2* to *clock-d3* is set to 4.5ns, because there is only a maximum operation during this period, and the longest data path is about 4ns.

Figure 9 is the block diagram of a PE design with multistage (The LUT logic will be discussed in next section). In the design, α and β are software programmable parameters, and the values of $\sigma(S[i],T[j])$ are also two software programmable parameters. By setting the parameters properly, the hardware implementation could be compatible with the existing software programs, including both linear and affine gap model algorithms.

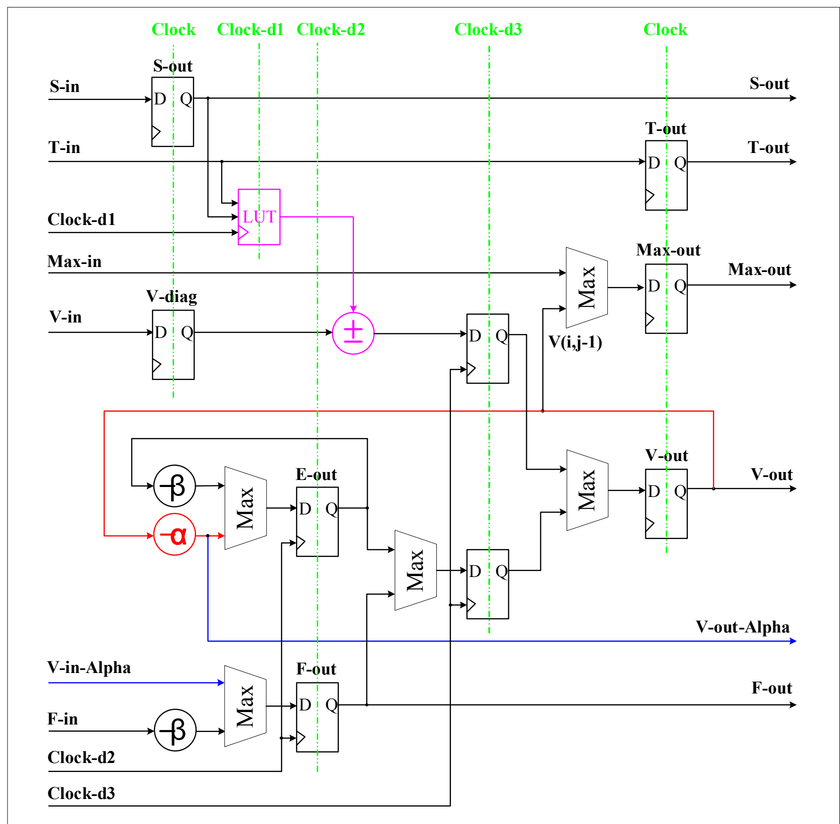


Figure 9. Multistage processing element design

When implemented in the Altera StratixII FPGA, it will cost about 180 ALUT. In contrast, for the un-optimized straightforward PE design, even when considering the techniques of packing the max operation with the output register(for *V-out* and *Max-out*), it will still need 300 ALUT. Therefore through the optimizations discussed in section 4.1 through 4.5, we reduced about 40% ALUT usage of a PE, this means that we can implement more PEs in the FPGA, so that more parallelism could be exploited from it. In the meanwhile, the uneven stage latencies control mechanism guarantees that the PE can work at a reasonable high frequency. In our final implementation, the main clock frequency of the PE is 66.7MHz.

4. Look up table design in Smith-Waterman PE for protein sequence

The difference between a DNA sequence and a protein sequence is that there are only 4 types of nucleotides (A, G, C and T) in a DNA sequence, but there are 20 types of amino acids in a protein sequence. When encoding the DNA sequence, we need only 2 bits to represent the 4 letters, but for the amino acid sequence, we need at least 5 bits. That will cost a little

bit more DFFs to store the sequences in the PE design, but will not affect the structure of the PE design.

The key point is that the penalty score substitution matrices for the nucleotide and amino acid are totally different. For the DNA sequence, there are only two values for the substitution matrix. These are the values when *S[i]* and *T[j]* are equal and not equal. But for the protein sequence, the penalty score of substituting a letter with another letter is different for each letter because of the biology meaning. Therefore the substitution matrix is normally organized as a 20*20 penalty score matrix. A widely used score matrix – Blosum62 is shown in Table 1 as below [15].

Table 1. Blosum62 substitution matrix

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

Obviously we need to store the substitution matrix in a RAM block as a look up table in the PE. In Blosum62, the maximum score is 11, and the minimum score is -4, but in some other types of substitution matrices, the data may vary in a bigger range. Considering the generality, we set the data width to 9 bit, the MSB is the sign bit, therefore the data range is "-255 ~ +255", it should be big enough for most cases.

As we mentioned in the beginning of this section, we need 10 bits to store *S[i]* and *T[j]* in the PE, 5 bits for each, if we simply implement the look up table without any optimization, the depth of the RAM should be 2^{10}=1024, therefore we need two M4k RAM blocks to store a look up table.

In fact, there are only 400 entries in the table used by the substitution matrix, the others are blank, which are occupied by the non-existent encoding positions. In the meanwhile, the substitution matrix is symmetric to the diagonal line, only 210

elements are necessary, therefore if we simply implement the look up table without any optimization, nearly 80% memory space will be wasted in storing the unnecessary information. To store the substitution matrix more efficiently in the RAM block, we introduced a new storage method which divide the matrix into four small matrices and store only 3 of them respectively. The sub-matrix partition is shown in Table 2.

Table 2. Sub-Matrix partition

S/T	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	10	11	12	13
00																				
01																				
02																				
03																				
04																				
05																				
06																				
07																				
08																				
09																				
0a																				
0b																				
0c																				
0d																				
0e																				
0f																				
10																				
11																				
12																				
13																				

We can divide the whole 20*20 matrix into four sub-matrices according to the four possible combinations of the MSB of *S[i]* and *T[j]*, as shown in Figure 11. The yellow partition is a 16*16 matrix, the blue partition is a 4*4 matrix, and the pink partitions are two symmetric 4*16 matrices. Based on the partition, we can store these sub-matrices into a RAM block according the rules in the following Table 3 and Table 4.

Table 3. Substitution matrix element address encoding method

MSB of S & T		MSB			Memory Address[8:0]					LSB
Ds[4]	Dt[4]	Bit[8]	Bit[7]	Bit[6]	Bit[5]	Bit[4]	Bit[3]	Bit[2]	Bit[1]	Bit[0]
0	0	0	Ds[3:0]				Dt[3:0]			
0	1	1	0	0	Dt[1:0]		Ds[3:0]			
1	0	1	0	0	Ds[1:0]		Dt[3:0]			
1	1	1	0	1	0	0	Ds[1:0]		Dt[1:0]	

Table 3 shows how to map these sub-matrices into the different positions of the RAM block, or how to assign addresses to the matrix elements. We encode *S[i]* and *T[j]* to *Ds[4:0]* and *Dt[4:0]* respectively according the alphabet

sequential.

- When *Ds[4]=0* and *Dt[4]=0*, let *Addr[8]=0, Addr[7:4]=Ds[4:0], Addr[3:0]=Dt[4:0],* there are 256 entries in this address range, and the yellow partition will be stored in it;
- When *Ds[4]=0* and *Dt[4]=1*, let *Addr[8:6]=3’b100, Addr[5:4]=Dt[1:0], Addr[3:0]=Ds[4:0],* there are 64 entries in this address range, and the pink partition will be stored in this area;
- When *Ds[4]=1* and *Dt[4]=0*, we can swap Ds and Dt, so the second pink partition will be stored in the same address range. In other words, because the two pink partitions are symmetric, we only store one of them to the RAM block.
- When *Ds[4]=1* and *Dt[4]=1*, let *Addr[8:4]=5’b10100, Addr[3:2]=Ds[1:0], Addr[1:0]=Dt[1:0],* there are 16 entries in this address range, and the blue partition will be stored in this area.

According this method, the three different partitions could be stored in a continuous address range from *9’b0_0000_0000* to *9’b1_0100_1111*, only 336 entries are needed to store the whole substitution matrix. In a real FPGA, we can use an M4k block to carry out this implementation.

Table 4. Substitution matrix storage structure in memory

MSB			Address					LSB	MSB(=Signed bit)			Data					LSB
0	0	0	0	0	0	0	0	0	S								
									S								
									S								
									S								
0	1	1	1	1	1	1	1	1	S								
1	0	0	0	0	0	0	0	0	S								
									S								
									S								
1	0	0	1	1	1	1	1	1	S								
1	0	1	0	0	0	0	0	0	S								
									S								
1	0	1	0	0	1	1	1	1	S								

Table 4 is the memory storage structure of the substitution matrix, the left 9 columns are the address of the memory, and the right 9 columns are the data stored in the memory. The MSB of the data is signed bit, when it is 0, the value is a non-negative number; when it is 1, the value is a negative number.

5. Core logic design of the Smith-Waterman algorithm

In order to implement the whole calculation process, we need to design some supplemental modules outside the systolic PE array, including counters, FIFOs, shifters, calculator and some registers. The block diagram of core logic is shown as in Figure 10.

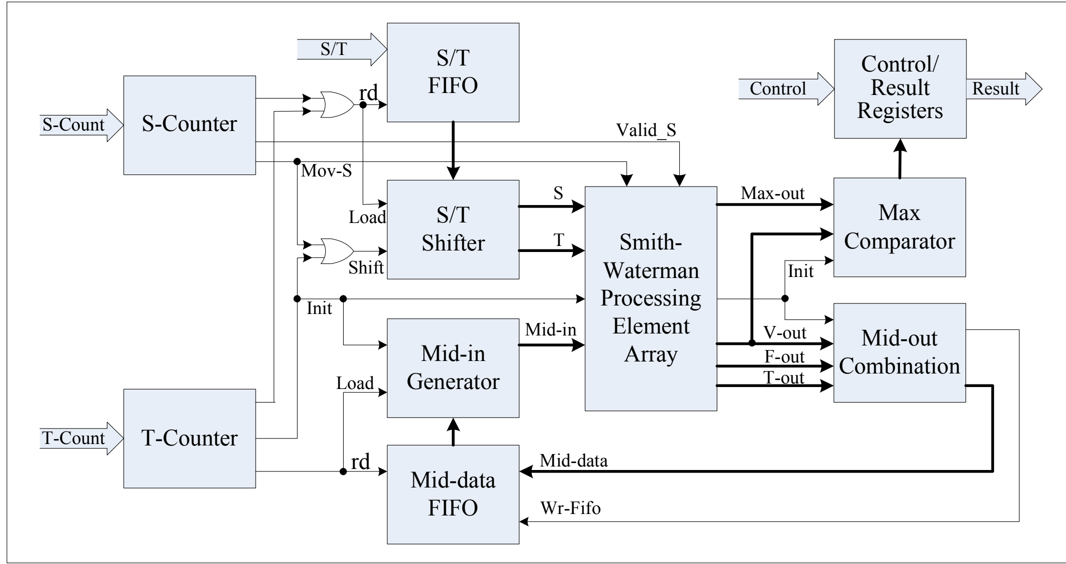


Figure 10. Core logic design of the Smith-Waterman algorithm

In Figure 10, the functions of each module are described as below:

- The *S* and *T Counter* are used to control the sequences shifting into the PE array process according to the length of *S*/*T*;
- The *S/T FIFO* is used to buffer the input data coming from HyperTransport bus interface;
- The *S/T Shifter* is used to shift the 64 bit input data to an encoded sequence letter in each clock cycle, and the encoded sequence letter will be transferred into the PE array;
- The *Mid-data FIFO* is used to stored the temporary intermediate data come out from the PE array when multi-iteration calculation is needed;
- The *Mid-in Generator* module is used to calculate *"V-in – α"* for the first PE, and it is also used to buffer the mid data for the PE array according to the requirement of multistage clocks;
- The *Mid-out Combination* module is used to combine the output from the PE array and write to *Mid-data FIFO*;
- The *Max Comparator* module is used to compare the *Max-out* and *V-out* from the PE array and store the maximum value in it. This part is equivalent to the second comparison step in Figure 5 (the purple Max block), it stands for the comparison between the vertical neighbor matrix elements.
- The *Control/Result Registers* module is used by the host to write parameters and control registers to, and read the status as well as the result from.

6. Performance Evaluation

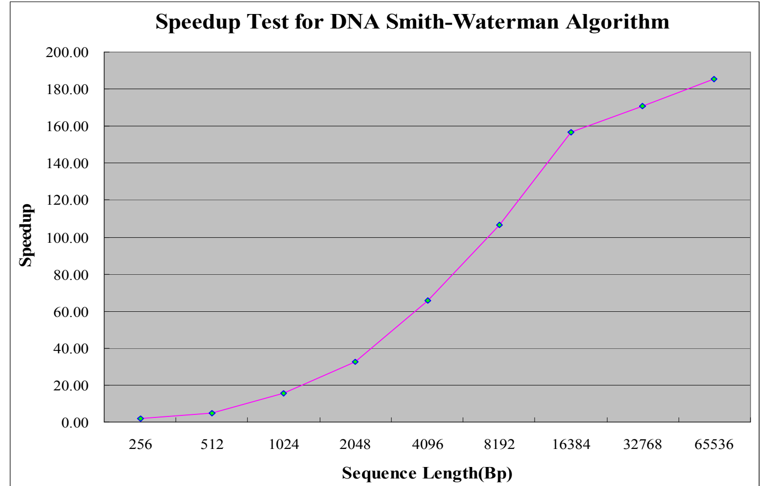
To evaluate the performance of the FPGA, we compare it with the host system of the XD1000 platform, which has a 2.2GHz AMD64 Opteron processor and 8GB DDR2 SDRAM memory in it. The operating system of the XD1000 is Linux 2.6.16.14, all software was compiled by GCC 4.1.1 with the "–O3" optimization option.

6.1 Speedup test for Smith-Waterman algorithm of DNA

For the Smith-Waterman algorithm of DNA, we implement 384 PE in the FPGA, which cost 121,836 ALUT (85% of 143,520 ALUT in total) and 3,587,296 memory bits (38% of 9,383,040 bits in total). The PE array working frequency is 66.7MHz, and the peak performance of the PE array is 25.6 GCUPS. The testing results are shown in Table 5.

Table 5. Speedup test result of Smith-Waterman algorithm of DNA

Length of S/T (Bp)	Software Time (s)	FPGA Time (s)	Speedup
256	0.000461	0.000226	2.04
512	0.001837	0.000374	4.91
1024	0.007307	0.000472	15.48
2048	0.029225	0.000898	32.54
4096	0.116680	0.001781	65.51
8192	0.497743	0.004661	106.79
16384	2.208849	0.014080	156.88
32768	8.351658	0.048909	170.76
65536	33.524406	0.180816	185.41



We can find that when the sequences are short, the speedup is very low. For example, when the length of *S/T* sequence both are 256Bp, the speedup over the software is only 2.04, that's because:

(1) The PE array is not fully joining in the calculation. At any clock cycle, only a fraction of the PEs are effectively running. In fact, amongst the 384 PE, only 256 of them are valid with a query letter in it (from sequence *S*), and the beginning 128 PE are invalid. According to working principle of the systolic PE array, all the letter of sequence *T* will firstly transfer through the beginning 128 PE before a valid calculation would occur. Therefore, the beginning 128 cycles are used to pipe the sequence T to the first valid PE, in the 129th cycle afterwards, there is only 1 PE participate the calculation, then for each following clock cycles, another PE will join in the calculation until in the 384th cycle, there will be 256 PE running, after that point, for each following clock cycles, a PE will quit from the calculation, until in the 640th cycle, only the last PE is running. Therefore the calculating will cost 640 cycles in total, and only there are average 102 PEs are running effectively in each cycle.

(2) Another important reason is that the FPGA initialization time is a constant when the sequences are very short. Before transferring the sequences to the PE array, each nucleotide letter is encoded to a 2-bit data, for example, the 256Bp will be decoded to 512 bit or 64 byte data. However according to the HT DMA transfer requirement, the minimum transfer block is 4Kbyte. Therefore, it will cost the same time to transfer a 256Bp sequence as to transfer a 16KBp sequence from the host CPU to the PE array. This also implies that it is needed to improve the performance of transferring small data block for the XD1000 platform.

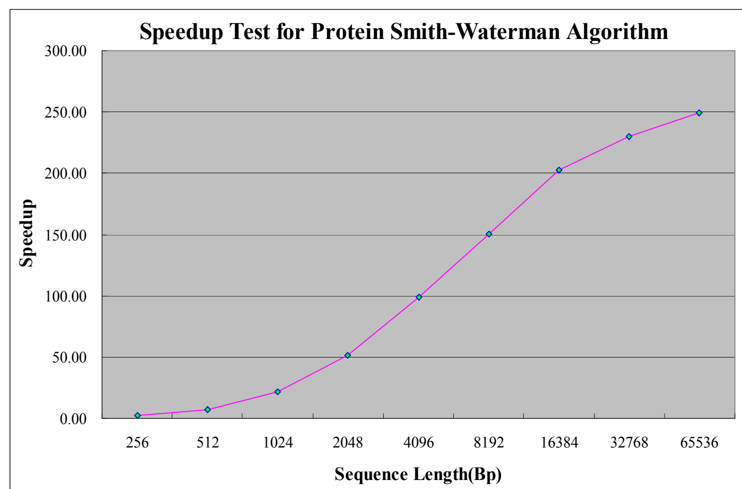
(3) During the calculation, except the parallel part that could be run by the PE array, still there is a serial part that needs to be run by the host CPU, for example, preparing data, initializing the FPGA, etc. According to Amdahl's law, when the task scale is small, this serial part will dominate the time cost, no matter how many times you can speedup the parallel part, the overall improvement of the task will be limited by the parallel part to serial part ratio.

6.2 Speedup test for Smith-Waterman algorithm of protein

For the Smith-Waterman algorithm of protein, we also implement 384 PE in the FPGA, which cost 111,574 ALUT (78% of 143,520 ALUT in total) and 5,348,320 memory bits (57% of 9,383,040 bits in total). The PE array working frequency is 66.7MHz, and the peak performance of the PE array is 25.6 GCUPS. The testing results are shown in Table 6.

Table 6. Speedup test result of Smith-Waterman algorithm of protein

Length of S/T (Bp)	Software Time (s)	FPGA Time (s)	Speedup
256	0.000675	0.000303	2.23
512	0.002693	0.000376	7.16
1024	0.010747	0.000490	21.93
2048	0.042928	0.000828	51.85
4096	0.172775	0.001753	98.56
8192	0.704018	0.004675	150.59
16384	2.887660	0.014221	203.06
32768	11.364635	0.049333	230.37
65536	45.297020	0.181534	249.52



Comparing with the DNA Smith-Waterman algorithm, it gets much higher speedup with the same hardware peak performance. That is because when using software to calculate, it needs to look up a big table to get the substitution matrix value, and it will use more time than just comparing two letters to see if they are equal.

A Blosum62 substitution matrix is shown as in Table 1. In the software implementation, the table is store in a two dimensional array, before access an element of the array, software need to calculate the index of the element. Unfortunately, the amino acid letters are not continuous as the alphabet, five letters are missing before the last letter –*"Y"*, therefore, the software program have to use the *"case"* statement to map a letter to an index. There are 20 types letters in the protein sequences, so the software program needs very long *"case"* statement. Without optimization, the software program would cost even as twice long time as in Table 6.

As an improvement, we expand the substitution matrix from 20*20 to 25*25, put these five missing letters into the matrix, and re-order the matrix according alphabet sequential. The substituting score from a valid letter to an invalid letter is set to zero. By this way, we can calculate the index by only one sentence: just use the ASCII value of a letter minus 0x41 (0x41 is the ASCII of "A"). This way helps the software runs 100% faster than the un-optimized software version.

7. Conclusion and future work

In this paper, we present implementations of the Smith-Waterman algorithm for both DNA and protein sequences based on the XD1000 innovative reconfigurable supercomputing platform. To exploit more parallelism from the FPGA, we propose a multistage processing element design with uneven stage latencies control mechanism and a compressed substitution matrix storage method, which greatly decrease the FPGA resources usage. By these methods, we implement a 384-PE systolic array working on 66.7 MHz, which can achieve 25.6 GCUPS peak performance. Comparing with the 2.2GHz AMD64 Opteron host processor of XD1000 platform, the FPGA could gain 185 and 250 times speedup respectively.

In the meanwhile, our design extends the sequence length limit to 64KBp, which will satisfy the requirement of various applications; In the Smith-Waterman algorithm design for DNA sequence, there are four software programmable parameters, which allow the hardware implementation compatible with the existing software programs, including both linear and affine gap model algorithms; In the Smith-Waterman algorithm design for protein sequence, the substitution matrix is also reconfigurable, which allow users to choose from the different evolution models or develop their own evolution models. These features also make our implementation much more practicable in the real application.

Our future work includes extending our design to accelerate the network content processing, such as approximate character string matching for multi-patterns and multi-rules, network intruding detection, etc. An initial experiment shows that it will probably achieve more than 100 times speedup over the general purpose CPU. We will also try to harness the reconfigurable computing platform in the scientific computing area of floating point, such as Monte Carlo algorithm, BLAS, FFT, FIR, etc., and hopefully overcome the performance of Opteron processor by an order of magnitude.

Acknowledgements

We would like to than XtremeData Inc., they granted us to join their university plan and donated our laboratory the XD1000 platform coming with well prepared documents. We would also like to thank Mr. Gary Finley of XtremeData Inc., he give us a lot of technical support on the platform. Thanks to Mr. Dimitrij Krepis, he helped a lot in setting up the XD1000 environment. I would specially thank to Mr. Tom St. John, he reviewed the paper very carefully and revised many errors in the writing.

References

[1] Altera Corporation, StratixII Device Handbook, http://www.altera.com/, 2006

[2] CLC Bio Inc., http://www.clcbio.com/, 2006

[3] E. Chow, T. Hunkapiller, J. Peterson, M. S. Waterman, "Biological Information Signal Processor," in Proc. Int. Conf. ASAP (M. Valero et al., eds.), Los Alamitos, CA, pp: 144~160, IEEE CS, September 1991

[4] O. Gotoh, "An Improved Algorithm for Matching Biological Sequences", Journal of Molecular Biology, 162, pp:

705~708, 1982

[5] J. D. Hirschberg, R. Hughey, K. Karplus, Kestrel: “A programmable array for sequence analysis”, In: Proc. Int. Conf. ASAP ’96, IEEE CS, pp: 25~34, Chicago, IL, 1996

[6] D. T. Hoang, “A systolic array for the sequence alignment problem”, Brown University, Providence, RI, Technical Report, pages CS-92-22, 1992

[7] D. T. Hoang, “Searching genetic databases on splash 2”, Proc. IEEE Workshop on FPGAs for Custom Computing Machines, pp: 185~192, CS Press, Los Alamitos, CA, 1993

[8] R. Hughey, D. P. Lopresti, “B-SYS: A 470-processor programmable systolic array”, Proc. Int. Conf. Parallel Processing (C.Wu, ed.), vol. 1, (Boca Raton, FL), pp: 580~583, CRC Press, August 1991

[9] HyperTransport Consortium, http://www.hypertransport.org/, 2006

[10] H. T. Kung, C. E. Leiserson, “Systolic Arrays for VLSI”, Interim report, Department of Computer Science, Carnegie Mellon University, December 1978

[11] H. T. Kung, C. E. Leiserson, “Algorithms for VLSI Processor Arrays”, Introduction to VLSI Systems (C. A. Mead and L. A. Conway, eds.), chapter 8.3, pp: 271~292, Addison-Wesley, 1980

[12] H. T. Kung. “Why systolic architectures?”, IEEE Computer, 15(1), pp: 37~46, January 1982

[13] Richard J. Lipton and Daniel Lopresti, “A Systolic Array for Rapid String Comparison”, Proceedings of the Chapel Hill Conference on Very Large Scale Integration, pp: 363~376, 1985

[14] D.P. Lopresti, “P-NAC: A Systolic Array for Comparing Nucleic Acid Sequences”, IEEE Computer, 20 (7), pp: 98~99, 1987

[15] National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov/, 2006

[16] Timothy Oliver, Bertil Schmidt, “High Performance Biosequence Database Scanning on Reconfigurable Platforms”, Proceedings of the 18th IPDPS, 2004

[17] Timothy Oliver, Bertil Schmidt, Douglas Maskell, “Hyper Customized Processors for Bio-Sequence Database Scanning on FPGAs”, FPGA’05, Monterey, CA, 2005

[18] T. F. Smith, M. S. Waterman, “Identification of Common Molecular Subsequences”, Journal of Molecular Biology, 147(1): pp: 195~197, 1981

[19] TimeLogic Corp., http://www.timelogic.com/, 2005

[20] XtremeData, Inc., XD1000 Development System, http://www.xtremedatainc.com/, 2006

[21] C. T. White, R. K. Singh, et al. “BioSCAN: A VLSI-based System for Biosequence Analysis”, IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp: 504~509, IEEE Computer Society Press, Washington, DC, 1991