White Paper

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

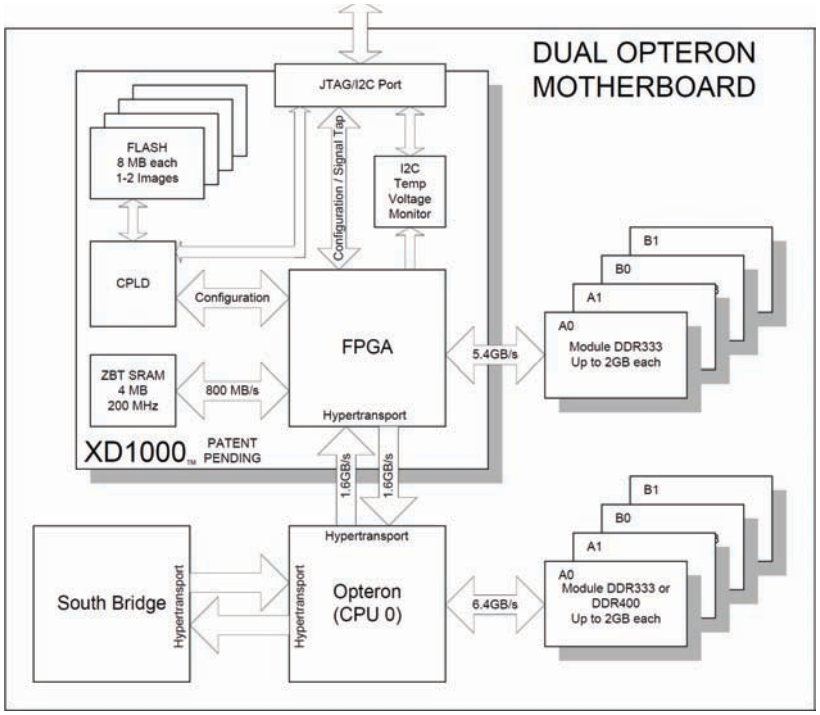
Abstract

An innovative reconfigurable supercomputing platform—XD1000—is being developed by XtremeData to exploit the rapid progress of FPGA technology and the high performance of HyperTransport™ 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: a multistage processing element (PE) design which significantly reduces the FPGA resource usage and allows more parallelism to be exploited; a pipelined control mechanism with uneven stage latencies—a key to minimize the overall PE pipeline cycle time; and 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 66.7 MHz, which can achieve 25.6 GCUPS peak performance. Compared with the 2.2-GHz AMD Opteron host processor, the FPGA coprocessor results in acceleration of 185 and 250, respectively.

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 the HyperTransport interconnection that provides an 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 integrates a leading-edge Altera® Stratix® II 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].

Figure 1. The XD1000 Platform Block Diagram



Utilizing 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, we bring forward a multistage PE design which significantly reduces the FPGA resource usage and, hence, allows more parallelism to be exploited. In addition, our design features a pipelined control mechanism with uneven stage latencies—a key to minimizing the overall PE pipeline cycle time. 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 66.7 MHz, which can achieve a peak performance of 25.6 GCUPS. Compared with the 2.20-GHz AMD Opteron host processor, the FPGA could gain 185 and 250 times acceleration, respectively.

Smith-Waterman Algorithm and Systolic PE 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 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 the 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 1 and Formula 2:

Initialization:

$$\begin{cases} V(i,0) = 0, 0 \le i \le n \\ V(0,j) = 0, 0 \le j \le m \end{cases} \quad \text{(Formula 1)}$$

Recursion relation:

$$V(i,j) = \max \begin{cases} 0 \\ V(i-1,j-1) + \sigma(S[i],T[j]) \\ V(i-1,j) + \sigma(S[i],-) \\ V(i,j-1) + \sigma(-,T[j]) \end{cases}, 1 \le i \le n, 1 \le j \le n \quad \text{(Formula 2)}$$

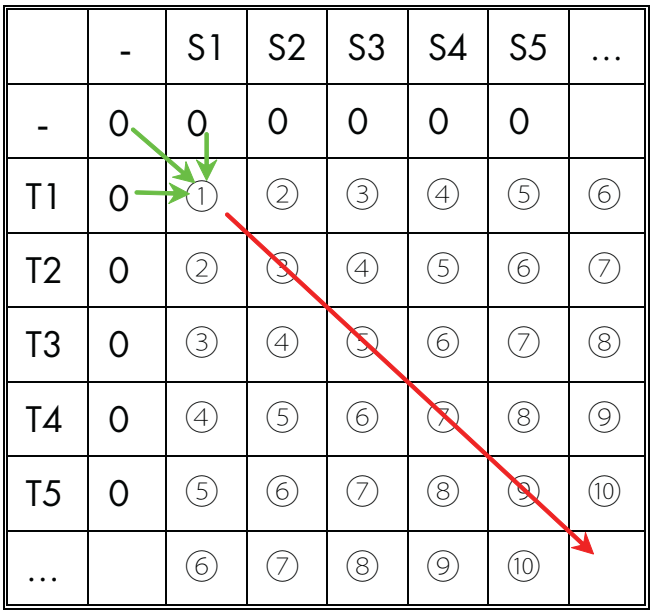
In these formulas, a "-" stands for a null character or gap; $V(i,0)$ stands for the result of comparing each character in S with a gap in T; the definition of $V(0,j)$ is the counterpart of comparing each character in T with a gap in S; and $(S[i],T[j])$ is the value of substitution matrix.

While calculating the similarity matrix, the score of any matrix element $V(i,j)$ always depends on the score of three other elements:

- The up-left neighbor element $V(i-1,j-1)$
- The left neighbor $V(i,j-1)$
- The up neighbor $V(i-1,j)$

Therefore, the calculation sequence of the similarity matrix will be as shown as Figure 2. It begins from the top-left element to bottom-right element according to the direction as shown by the arrow. Through observation of the similarity matrix calculation process, we found that for each clock cycle, every element on an anti-diagonal line marked with the same number could be calculated simultaneously, with the standing for the elements that could be calculated at the 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.

Figure 2. Similarity Matrix Calculating Sequence and Data Dependency



To further describe the level of similarity between two real bioinformatics sequences, an affine gap model was introduced to the Smith-Waterman algorithm by O. Gotoh in 1982 [4]. In the affine gap model, the gap is used to compensate for the insertion or deletion, to make the alignment more condensed in satisfying an expecting model. The 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*, and 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 below:

Initialization:

$$\begin{cases} V(i,0) = E(i,0) = 0, 0 \le i \le n \\ V(0,j) = F(0,j) = 0, 0 \le j \le m \end{cases} \quad \text{(Formula 3)}$$

Recursion relation:

$$V(i,j) = \max\begin{cases} 0 \\ E(i,j) \\ F(i,j) \\ V(i-1,j-1) + \sigma(S[i],T[j]) \end{cases}, 1 \le i \le n, 1 \le j \le m \quad \text{(Formula 4)}$$

$$E(i,j) = \max\begin{cases} V(i,j-1) - \alpha \\ E(i,j-1) - \beta \end{cases}, 1 \le i \le n, 1 \le j \le m \quad \text{(Formula 5)}$$

$$F(i,j) = \max\begin{cases} V(i-1,j) - \alpha \\ F(i-1,j) - \beta \end{cases}, 1 \le i \le n, 1 \le j \le m \quad \text{(Formula 6)}$$

In these formulas, α stands for the gap_open, and 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 initially introduced by H. T. Kung and C. E. Leiserson in their papers [10] [11], and it has proven to be very efficient in computing matrix multiplication or LU-decomposition. Further research revealed that dynamic programming algorithm also mapped very well to a systolic array due to its potential parallelity [12].

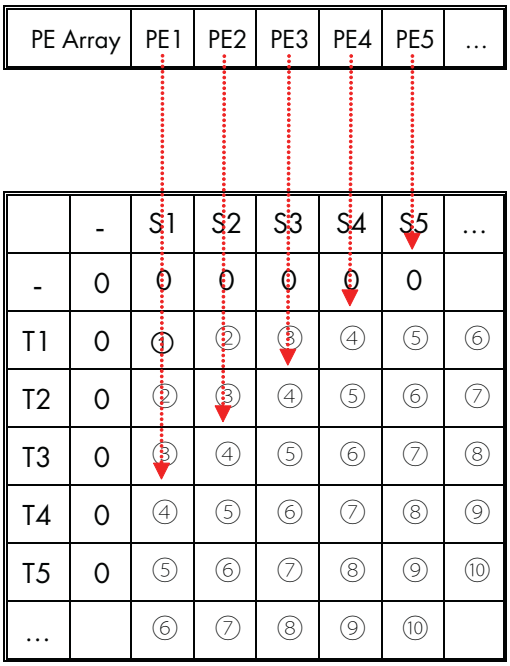
In 1985, R. J. Lipton and D. P. Lopresti mapped the edit distance algorithm, a typical dynamic programming algorithm for the global alignment of DNA sequences, to a systolic array by implementing an nMOS prototype chip [13]. The preliminary results of the prototype performed hundreds to thousands times 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 California Institute of Technology [3], *BioSCAN* of the University of North Carolina [21], *B-SYS*, *Splash /Splash-2* of Brown University, [8] [6] [7], *Kestrel* of the University of California at Santa Cruz [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, the XD1000. Compared with these works, from the perspective of application, our design extends the sequence length limit to 64 KBp, 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 allows 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 in a significant decrease of FPGA resource 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 fifth 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.

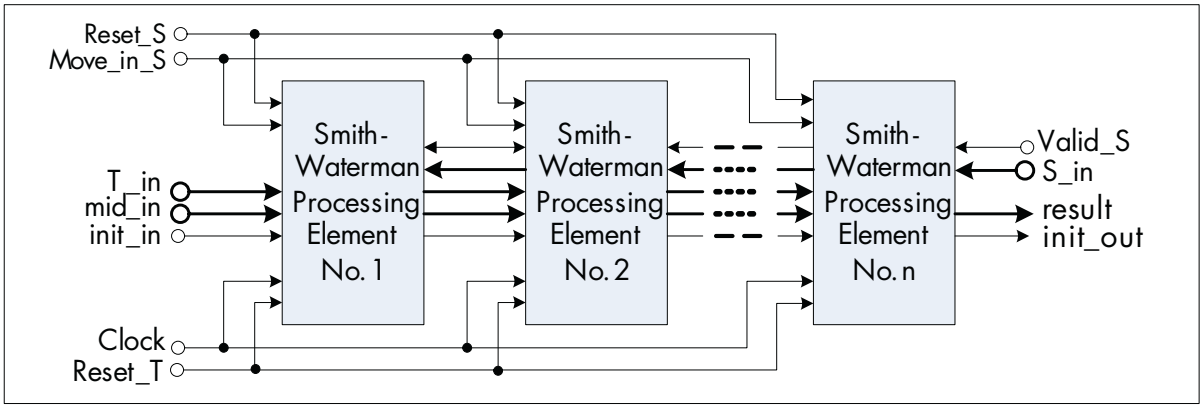
Figure 3. Mapping the Smith-Waterman Algorithm to a Systolic PE Array



Due to the hardware resource limitation, we can only implement a limited number of PEs 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 memory for the next iteration to use.

As shown in Figure 4, a systolic PE array consists of many identical cascading PEs. 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 decides 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



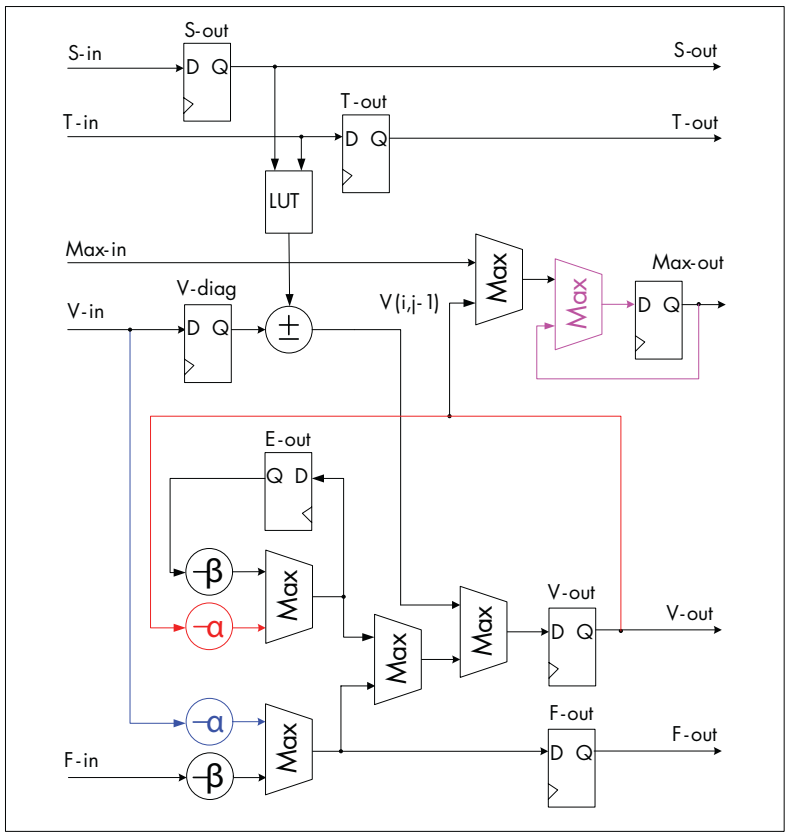
Our design was created such that the shift in direction of sequence *S* is opposite to that of sequence *T*. This configuration guarantees that sequence *S* will be stored in the PE array as the original sequence, which means the tail of the sequence will always be stored in the last PE. This method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

The Smith-Waterman Algorithm-PE Design

In Formula 3, Formula 4, Formula 5, and Formula 6, a straightforward PE schematic was proposed (shown in Figure 5) [16] [17], and the functions of each DFF (D-type flip-flop) are detailed below:

- *S-Out* and *T-out* DFFs 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 the 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 upper-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:
  - The maximum value coming from the previous PE
  - *V(i,j)* coming from the current PE
  - The maximum value stored in itself

Figure 5. A Straightforward Smith-Waterman Algorithm PE Design



Before the hardware implementation, we need to estimate the FPGA resources 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 64 KBp 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 multiplexing operation, there are 11 add/sub and 6 2:1

multiplexing operations in total for a PE. If the data width is set to 20 bits, a PE will require about 340 adaptive look-up tables (ALUTs).

The Altera Stratix II EP2S180 FPGA used as the FPGA coprocessor in this study has 14,3520s ALUTs in total. The HyperTransport interface and some other logic will require 8 percent of resources (normally, maximum FPGA resource utilization is less than 90 percent), so we can accommodate at most 340 PEs. Considering the cost of other necessary control logic modules, the number of PEs we can implement will be 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 PEs on the FPGA as possible.

Simplify the Max-Out Operation of V(i,j)

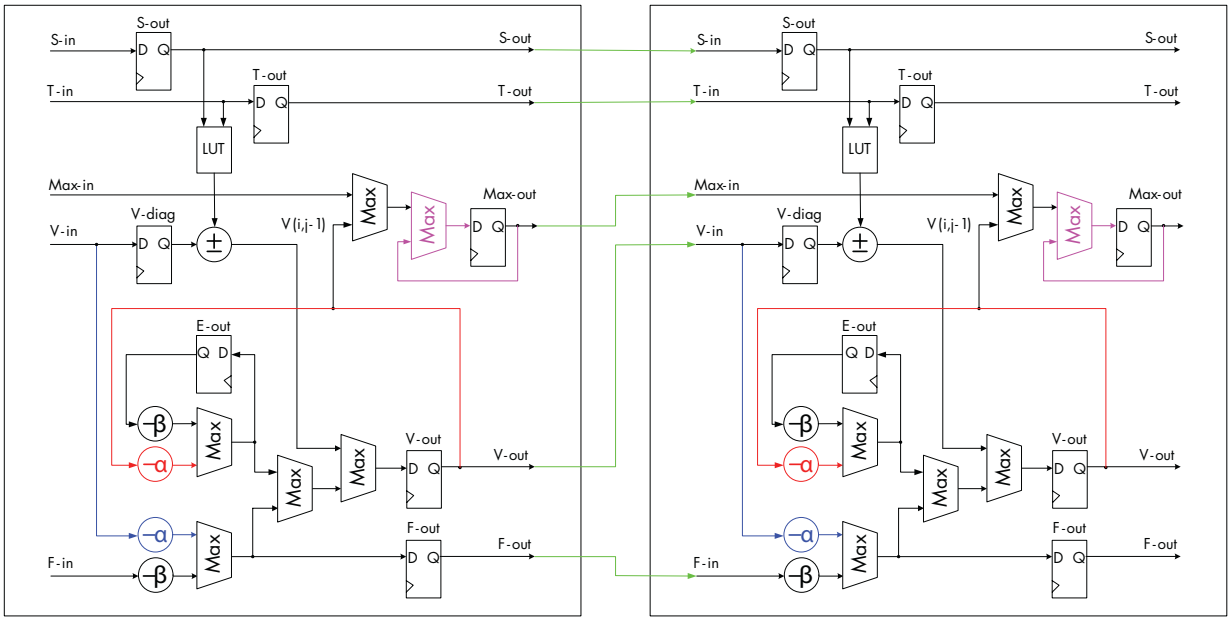
In the PE design, the *Max-out* DFF is used to store the maximum value of the similarity matrix. To get the maximum value, for the first step, it must compare the maximum output from the previous PE with *V(i,j)* of the current PE, which stands for the comparison between the horizontal neighbor matrix elements. In the second step, it must compare the output from the first step with the value stored in the *Max-out* DFF itself. This 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 maintained by each PE. This way, we can delete a maximum operation from the PE (the purple "max" block in Figure 5.).

Simplify the Operation of V(i,j) - α

When we cascade multiple PEs to form a PE array as shown in Figure 6, we 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 – 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

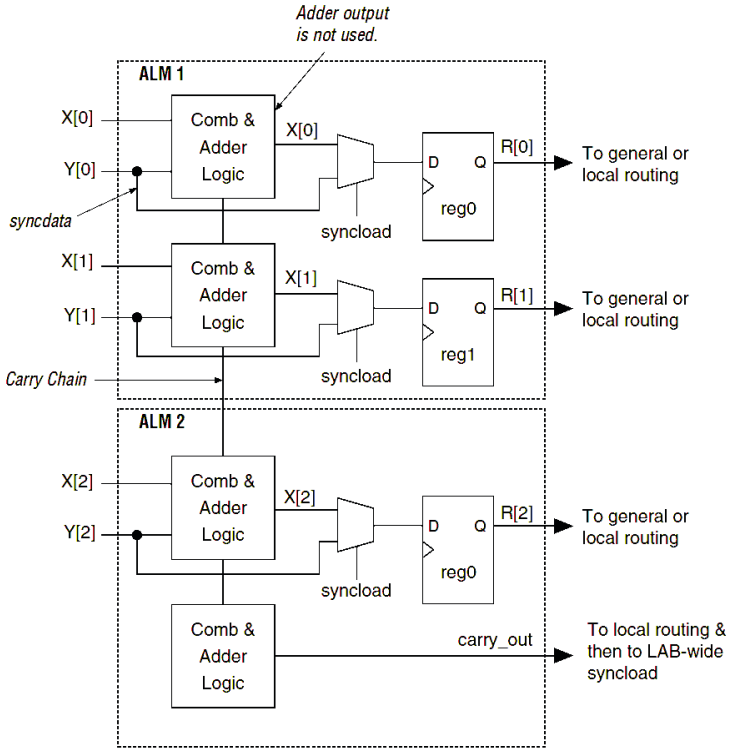


Compact the Max Operation

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

Compared to its previous-generation FPGA, the Stratix II FPGA represents a significant improvement in its adaptive logic module (ALM) 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 percent 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 logic array block (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 prerequisite to compacting the comparison operation and 2:1 multiplexing operation to an ALM is that the output of the 2:1 multiplexing 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. Because it improves the usage percentage in each ALM, however, it helps to conserve general FPGA resources.

A problem arising from these introduced DFFs is that the PE would require more than one clock cycle to finish the calculation of one matrix element, which would hinder the performance greatly.

Handling the Negative Number

When calculating the $V(i-1,j-1)+\sigma(S[j],T[j])$, the result is probably negative due to the $\sigma(S[j],T[j])$ being 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 it comes to the max function.

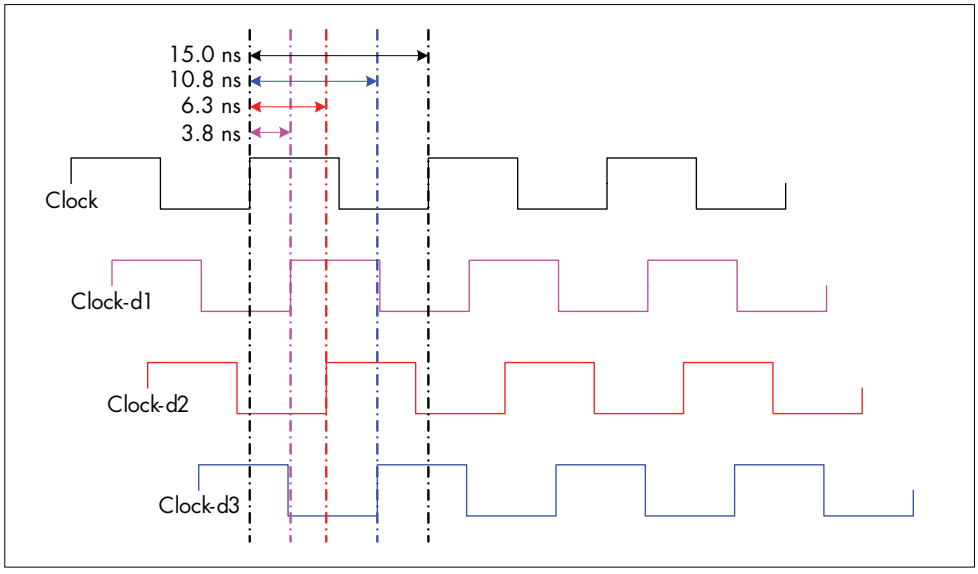
According to Formula 4, all negative numbers will be reset to zero. This means that whenever a negative number is generated from the add/sub operation, we can reset it to zero unconditionally. Therefore, in the PE design, we introduce a DFF to store the result from the add/sub operation, and we use the 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.

Multistage in the PE Design

In the previous sections, we introduced some DFFs to reduce the area cost and to process the negative numbers, but this also generated 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 minimizing the overall PE pipeline cycle time.

With the FPGA internal phase-locked loop (PLL), we generate four clocks with the same clock frequency but with a 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

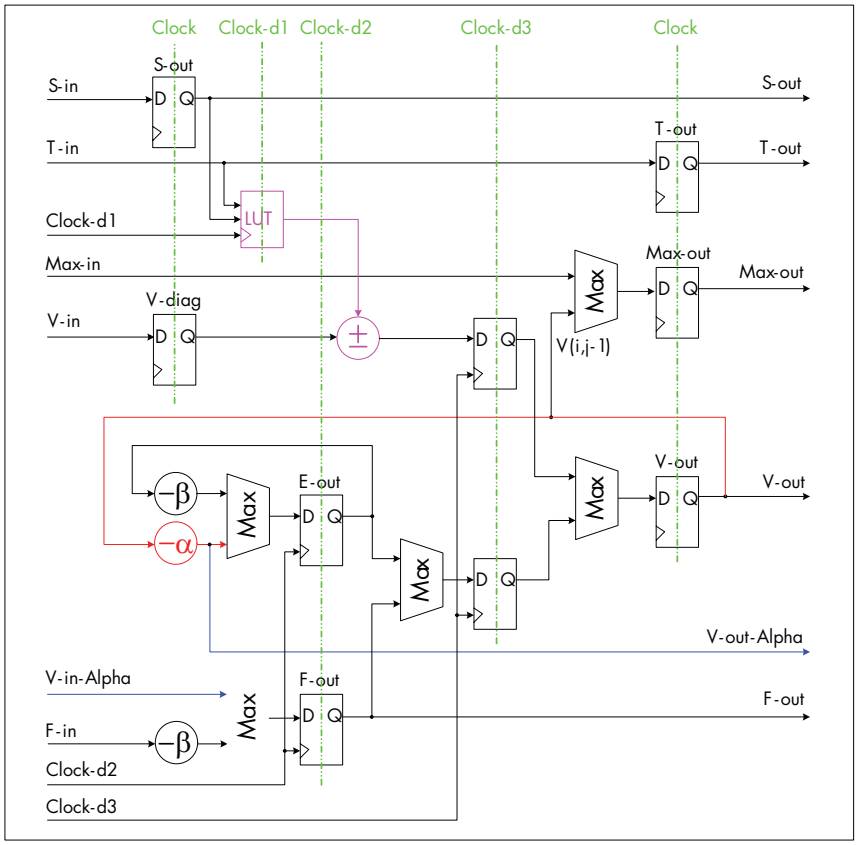


The phase delays are decided by the timing simulation of the PE design. For example, the delay from *clock* to *clock-d2*, is set to 6.3 ns, because there is a subtraction operation and a max operation that need to finish during this period, and the longest data path is about 6 ns; the delay from *clock-d2*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

Figure 9 is the block diagram of a PE design with multistage (the LUT logic will be discussed in the 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 PE Design



When implemented in the Altera Stratix II FPGA, the design will require about 180 ALUTs. In contrast, for the unoptimized 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 ALUTs. Therefore, through the optimizations discussed in the next section, we reduced about 40 percent of ALUT usage of a PE. This means that we can implement more PEs in the FPGA and, as a result, exploit more parallelism from it. Meanwhile, the uneven stage latencies control mechanism guarantees that the PE can work at a reasonably high frequency. In our final implementation, the main clock frequency of the PE is 66.7 MHz.

LUT Design in Smith-Waterman PE for Protein Sequence

The difference between a DNA sequence and a protein sequence is that there are only four 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 two bits to represent the four letters, but for the amino acid sequence, we need at least five bits. That will require 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 biological meaning. Therefore, the substitution matrix is normally organized as a 20*20 penalty score matrix. A widely used score matrix, Blosum62, is shown in Figure 10 [15].

Figure 10. Blosum62 Substitution Matrix

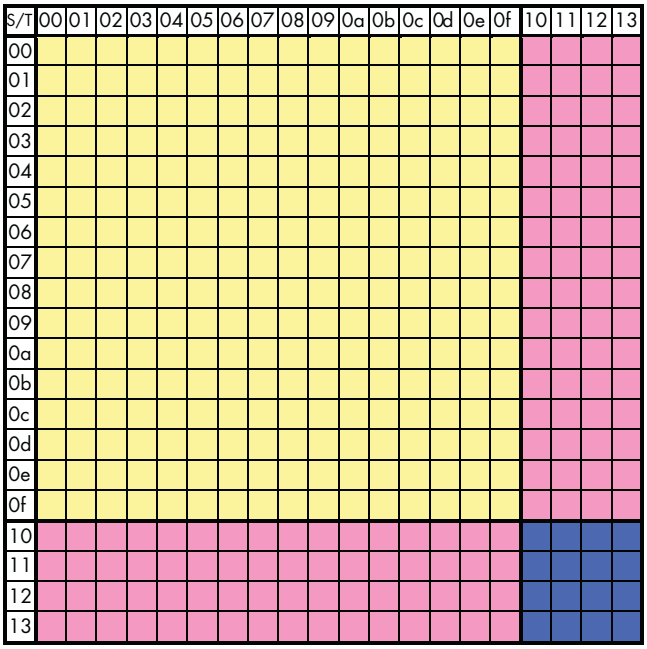
	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
C	9	-1	-1	-3	0	-3	-3	-3	-4	-3	-3	-3	-3	-1	-1	-1	-1	-2	-2	-2
S	-1	4	1	-1	1	0	1	0	0	0	-1	-1	0	-1	-2	-2	-2	-2	-2	-3
T	-1	1	4	1	-1	1	0	1	0	0	0	-1	0	-1	-2	-2	-2	-2	-2	-3
P	-3	-1	1	7	-1	-2	-1	-1	-1	-1	-2	-2	-1	-2	-3	-3	-2	-4	-3	-4
A	0	1	-1	-1	4	0	-1	-2	-1	-1	-2	-1	-1	-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 LUT 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 bits, and since the MSB is the sign bit, the data range is -255 ~ +255, which should be large 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 LUT without any optimization, the depth of the RAM should be 210=1024; therefore, we need two M4k RAM blocks to store a LUT.

In fact, there are only 400 entries in the table used by the substitution matrix. The others are blank, and occupied by the non-existent encoding positions. Meantime, the substitution matrix is symmetric to the diagonal line—only 210 elements are necessary, so if we simply implement the LUT without any optimization, nearly 80 percent of the 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 divides the matrix into four small matrices and stores only three of them respectively. The sub-matrix partition is shown in Figure 11.

Figure 11. Sub-Matrix Partition



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 Figure 12 and Figure 13.

Figure 12. Substitution Matrix Element Address Encoding Method

MSB of S & T		MSB			Memory Addess[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]	

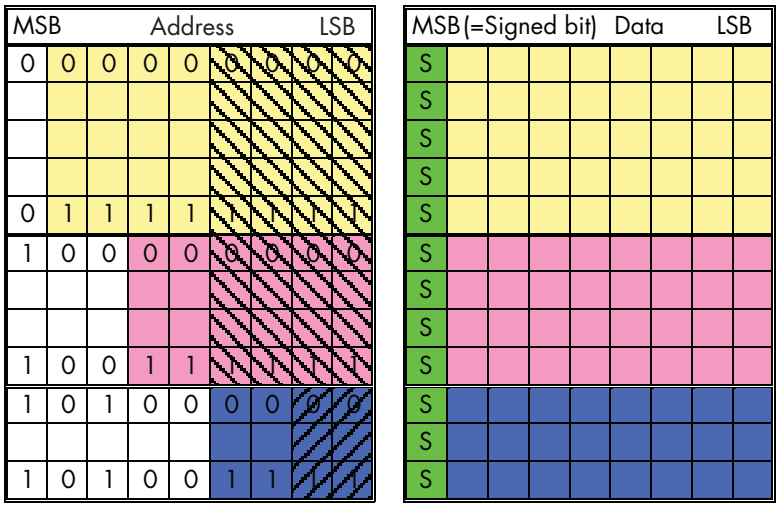
Figure 12 shows how to map these sub-matrices into the different positions of the RAM block, or how to assign addresses to the matrix elements. We encode *S[i]* and *T[j]* to *Ds[4:0]* and *Dt[4:0]* respectively, according to the alphabet sequential.

- 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 to this method, the three different partitions could be stored in a continuous address range from *9'b0_0000_0000* to *9'b1_0100_1111*, and 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.

Figure 13 shows the memory storage structure of the substitution matrix; the left nine columns are the addresses of the memory, and the right nine 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.

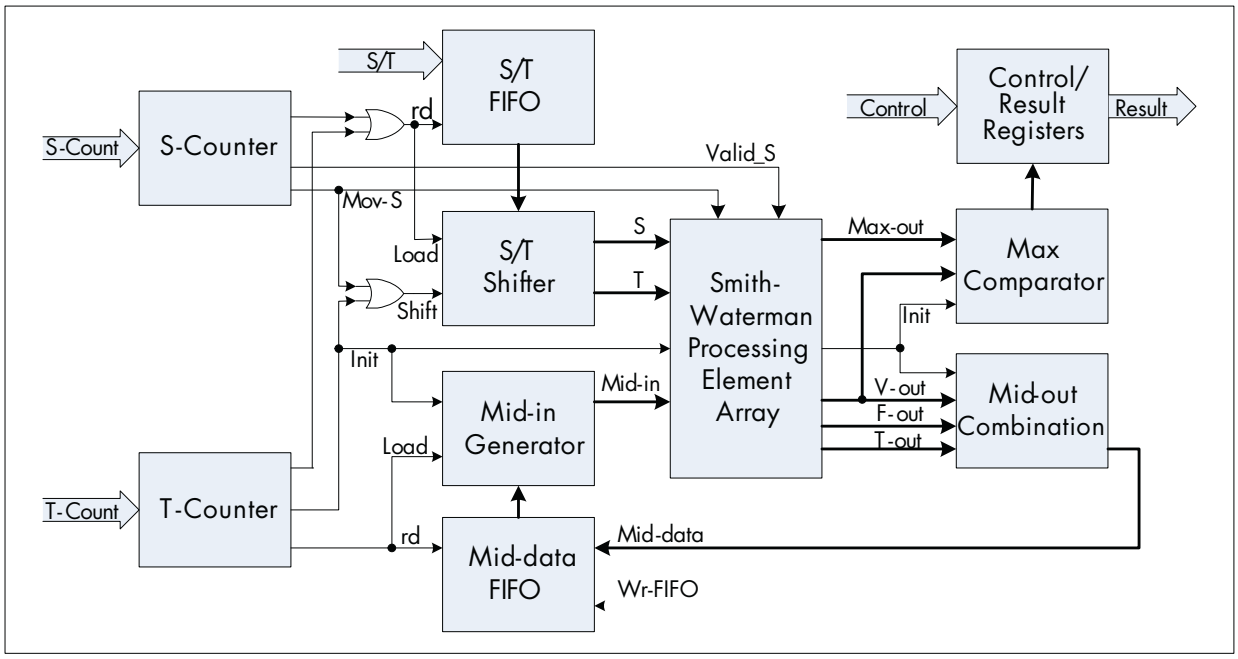
Figure 13. Substitution Matrix Storage Structure in Memory



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 in Figure 14.

Figure 14. Core Logic Design of the Smith-Waterman Algorithm



In Figure 14, the functions of each module are described 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 the 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 store the temporary intermediate data 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) and represents 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

Performance Evaluation

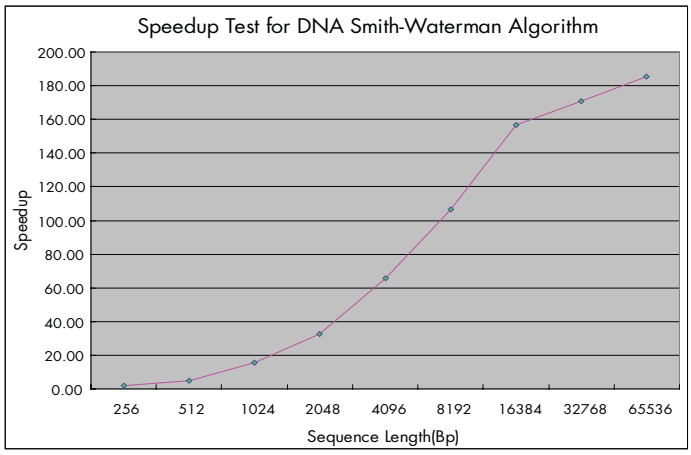
To evaluate the performance of the FPGA, we compared it with the host system of the XD1000 platform, which has a 2.2-GHz AMD64 Opteron processor and 8-Gbyte DDR2 SDRAM memory. The operating system of the XD1000 is Linux 2.6.16.14, and all software was compiled by GCC 4.1.1 with the "-O3" optimization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

Speedup Test for Smith-Waterman Algorithm of DNA

For the Smith-Waterman algorithm of DNA, we implemented 384 PEs in the FPGA, which cost 121,836 ALUTs (85 percent of 143,520 ALUTs in total) and 3,587,296 memory bits (38 percent of 9,383,040 bits in total). In this study, the PE array working frequency was 66.7MHz, and the peak performance of the PE array was 25.6 GCUPS. The testing results are shown in Figure 15.

Figure 15. 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 256 Bp, the speedup over the software is only 2.04 for the following reasons.

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 PEs, only 256 of them are valid with a query letter in it (from sequence *S*), and the beginning 128 PEs are invalid. According to the working principle of the systolic PE array, all the letters of sequence *T* will initially transfer through the beginning 128 PEs before a valid calculation can 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 participating in the calculation. For each of the following clock cycles, another PE will join in the calculation until the 384th cycle, where there will be 256 PEs running. After that point, for each following clock cycles, a PE will quit from the calculation, until in the 640th cycle, when only the last PE is running. Therefore, the calculating will cost 640 cycles in total, and there are only an average of 102 PEs running effectively in each cycle.

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 256 Bp will be decoded to 512-bit or 64-byte data. However, according to the HyperTransport direct memory access (DMA) transfer requirement, the minimum transfer block is 4 Kbytes. Therefore, it will cost the same time to transfer a 256 Bp sequence as to transfer a 16 KBp sequence from the host CPU to the PE array. This also implies that it is needed to improve the performance of transferring a small data block for the XD1000 platform.

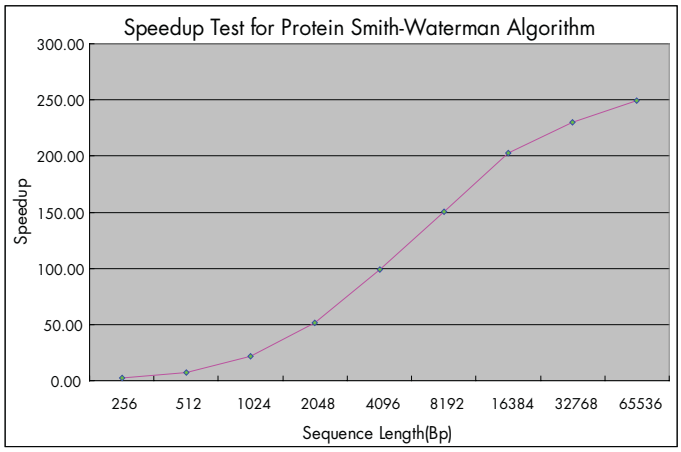
During the calculation, except for the parallel part that could be run by the PE array, there still is a serial part that needs to be run by the host CPU; for example, for such functions as preparing data, initializing the FPGA, etc. According to Amdahl's law, when the task scale is small, this serial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

Speedup Test for Smith-Waterman Algorithm of Protein

For the Smith-Waterman algorithm of protein, we also implemented 384 PEs in the FPGA, which cost 111,574 ALUTs (78 percent of 143,520 ALUTs in total) and 5,348,320 memory bits (57 percent of 9,383,040 bits in total). In this study, the PE array working frequency was 66.7 MHz, and the peak performance of the PE array was 25.6 GCUPS. The testing results are shown in Figure 16.

Figure 16. 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



Compared to the DNA Smith-Waterman algorithm, the protein test reveals much higher speedup with the same hardware peak performance. That is because when the calculations are driven by software, the software needs to look up a big table to get the substitution matrix value, and it will use more time than it would if it were just comparing two letters to see if they are equal.

In the software implementation for the Blosum62 substitution matrix (Figure 10), the matrix is stored in a two-dimensional array. Before accessing an element of the array, the software needs to calculate the index of the element. Unfortunately, the amino acid letters are not continuous as the alphabet is. Five letters are missing before the last letter "Y;" therefore, the software program has to use the "case" statement to map a letter to an index. There are 20 types of letters in the protein sequences, so the software program needs a very long case statement. Without optimization, the software program would require as much as twice the time, as shown in Figure 16.

As an improvement, we expanded the substitution matrix from 20*20 to 25*25, put these five missing letters into the matrix, and re-ordered the matrix according to the alphabet sequential. The substituting score from a valid letter to an invalid letter was set to zero. With this approach, we could calculate the index by only one sentence: just use the ASCII value of a letter minus 0x41 (0x41 is the ASCII of "A"). This approach enables the software to run 100 percent faster than the unoptimized software version.

Conclusion and Future Work

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

In the meantime, our design extended the sequence length limit to 64KBp, which satisfies 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 allows users to choose from the different evolution models or develop their own evolution models. These features also make our implementation much more practical in a 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 applications, such as processing Monte Carlo algorithms, BLAS, FFT, FIR, etc., hopefully extending the performance of the Opteron processor by an order of magnitude.

References

1. Altera Corporation, Stratix II Device Handbook, 2006:
www.altera.com

2. CLC Bio Inc., 2006:
www.clcbio.com

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, 2006:
www.hypertransport.org

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, 198.0

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, 2006:
www.ncbi.nlm.nih.gov

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., 2005:
www.timelogic.com

20. XtremeData, Inc., XD1000 Development System, 2006:
www.xtremedatainc.com

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.

Acknowledgements

We would like to thank XtremeData Inc. for inviting us to join their university program and donating to our laboratory the XD1000 platform with well prepared documents. We would also like to thank Mr. Gary Finley of XtremeData Inc. for technical support on the platform. Thanks to Mr. Dimitrij Krepis for help in setting up the XD1000 environment. I would specially like to thank Mr. Tom St. John for his careful edits on this paper.

- Peiheng Zhang, CAPSL, University of Delaware
- Guangming Tan, CAPSL, University of Delaware
- Guang R. Gao, CAPSL, University of Delaware