IMPLEMENTATION OF A DYNAMIC PROGRAMMING ALGORITHM FOR DNA SEQUENCE ALIGNMENT ON THE CELL MATRIX™ ARCHITECTURE

by

Bin Wang

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Donald H. Cooley Major Professor	Lisa J. K. Durbeck Committee Member
Nicholas J. Macias Committee Member	Daniel Watson Committee Member

Dr. Thomas Kent
Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2002

ABSTRACT

Implementation of a Dynamic Programming Algorithm for DNA

Sequence Alignment on the Cell Matrix™ Architecture

by

Bin Wang, Master of Science

Utah State University, 2002

Major Professor: Dr. Donald H. Cooley
Department: Computer Science

DNA sequence alignment is an important tool for modern molecular biology. The algorithm used by most sequence alignment tools is the dynamic programming algorithm introduced by Needleman and Wunsch in 1970. This algorithm has a time complexity of $O(n^2)$, where n is the length of the input sequence. This dynamic programming algorithm has been implemented on a new parallel computing architecture called Cell Matrix™. The approach taken in this work was to configure a block of Cell Matrix cells as a custom processor to perform the comparison and scoring function in the dynamic programming algorithm. The custom processors were then configured into a 2D array that closely matched the dynamic programming algorithm and allowed them to function in parallel. For an appropriately large Cell Matrix, the implementation in this thesis achieves a time complexity of O(n) in finding the optimal alignment score for two DNA sequences.

(37 pages)

To my wife.

ACKNOWLEDGMENTS

I would like to thank Dr. Donald H. Cooley for introducing me to the exciting nanocomputing architecture Cell Matrix™. I would especially like to thank Nicholas J. Macias and Lisa J. K. Durbeck at the Cell Matrix Corporation for making available to me the designing and testing tools used in this thesis. I would also like to thank my thesis committee members, Donald H. Cooley, Lisa J. K. Durbeck, Nicholas J. Macias, and Daniel Watson, for their guidance and support.

I give special thanks to my family, especially my wife, Fang Zhang. I could not have done it without their continuous encouragement and support.

Bin Wang

CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Sequence Alignment and Molecular Biology

An organism's genetic and functional information is stored as DNA, RNA, and proteins. These biological macromolecules are very complicated, composed of thousands or even millions of atoms chemically bonded together with complex 3-dimensional atomic structures. However, they are all polymer chains assembled from a fixed alphabet of well-understood chemicals. DNA is made up of four deoxyribonucleotides (adenine, thymine, cytosine, and guanine, or A T C G). RNA is made up of four ribonucleotides, and protein is made up of 20 amino acids. Because they are linear chains of defined components, they can be represented as sequences of symbols. The sequences of these macromolecules contain considerable information about their biological functions. In a DNA chain, the four deoxyribonucleotides can occur in any order, and the order they occur determines the DNA's biological function. In a protein, the amino acid sequence determines its 3D atomic structure and function. Sequence analysis of these macromolecules now plays an important role in modern molecular biology. For example, feature detection and pattern recognition are very useful tools to understand biological functions of new gene sequences that have not been characterized by traditional molecular biological experiments. With the development of modern methods for DNA sequencing, a huge amount of data has been and is being generated, especially for the human genome project. It is impossible to do any useful sequence analysis on this scale without the help of a computer. Many sequence analysis tools have been developed to assist molecular biologists. Because of the huge amount of data now available and the development of computer tools, sequence analysis is becoming increasingly powerful.

Sequence alignment is one of the most important operations in computational biology, facilitating everything from identification of gene function to structure prediction of proteins. Alignment of two sequences shows how similar the two sequences are, where there are differences between them, and the correspondence between similar subsequences. All of this represents important information for biologists. For example, sequences which are similar may also share a common origin, such as a common ancestor sequence, and thus may have similar or related 3D atomic structure and biological functions. A similar subsequence between two genes with similar function might represent a sequence that is critical to the common function of these two genes. As a result, this sequence remains largely unchanged through evolution, such as the protein sequence that forms the catalytic center of an enzyme. In 1983 sequence alignment assisted in the discovery of the link between cancer-causing genes (oncogenes) and genes involved in normal growth and development of the cell [2, 10]. Oncogene v-sys in the simian sarcoma virus causes uncontrolled cell growth and leads to cancer in monkeys. Growth factor PDGF is a protein that exists transiently in normal cells and stimulates normal cell growth. When the sequences of these two proteins are aligned, they show over 85% similarity (Figure 1). The sequence similarity between these two proteins suggests that the simian sarcoma virus causes cancer by using its v-sys to stimulate cell growth through the same pathway as PDGF. This observation led to the conclusion that cancer may be caused by a normal cell growth pathway being switched on at the wrong time.

```
LALIGN finds the best local alignments between two sequences
 version 2.2u00 November 2001
Please cite:
 X. Huang and W. Miller (1991) Adv. Appl. Math. 12:373-381

alignments <  E(  0.05):score: 54 (50 max)
 Comparison of:
(A) @          v-sys - 245 aa
(B) @          PDGF  - 295 aa
 using matrix file: BL50, gap open/ext: -12/-2 E(limit)   0.05

  85.5% identity in 249 aa overlap (2-245:47-295); score: 1407 E(10000): 1.8e-108

              10        20             30        40        50        60        70
v-sys  ISSIMIANSARCMAVIRSMT-----LTWQGDPIPEELYKMLSGHSIRSFDDLQRLLQGDSGKEDGAELDLNMTRS
       : .  .::  :: :.. :.      .. .:::::::::.::: :::::::::::::.:: :.:::::::::::::
PDGF   IENSHMANMNRCWALFLSLCCYLRLVSAEGDPIPEELYEMLSDHSIRSFDDLQRLLHGDPGEEDGAELDLNMTRS
         50        60        70        80        90       100       110       120

              80        90       100       110       120       130       140
v-sys  HSGGELESLARGKRSLGSLSVAEPAMIAECKTRTEVFEISRRLIDRTNANFLVWPPCVEVQRCSGCCNNRNVQCR
       ::::::::::::.::::::..::::::::::::::::::::::::::::::::::::::::::::::::::::::
PDGF   HSGGELESLARGRRSLGSLTIAEPAMIAECKTRTEVFEISRRLIDRTNANFLVWPPCVEVQRCSGCCNNRNVQCR
             130       140       150       160       170       180       190

        150       160       170       180       190       200       210       220
v-sys  PTQVQLRPVQVRKIEIVRKKPIFKKATVTLEDHLACKCEIVAAARAVTRSPGTSQEQRAKTTQSRVTIRTVRVRR
       ::::::::::::::::::::::::::::::::::::::: ::::: :::::: :::::::: :.:::::::::::
PDGF   PTQVQLRPVQVRKIEIVRKKPIFKKATVTLEDHLACKCETVAAARPVTRSPGGSQEQRAKTPQTRVTIRTVRVRR
        200       210       220       230       240       250       260       270

             230       240
v-sys  PPKGKHRKCKHTHDKTALKETLGA
       :::::::: :::::::::::::::
PDGF   PPKGKHRKFKHTHDKTALKETLGA
             280       290
```

Figure 1. Sequence alignment of v-sys and PDGF protein sequence. Identical sequences are highlighted. The PDGF and v-sys sequences were obtained from the SWISS-PROT protein knowledgebase (http://www.expasy.ch/sprot/). Sequence alignment was performed using the LALIGN tool in the FASTA sequence analysis software package (http://fasta.bioch.virginia.edu/).

Dynamic Programming Algorithm for Sequence Alignment

To form an alignment between two sequences, spaces may be inserted in arbitrary positions in the sequences so that they end up with same length, and then each character or space in one sequence will have a corresponding character or space in the other sequence. An alignment score can then be assigned to such an alignment: if a character in sequence A matches its corresponding character in sequence B, it will receive a score of 1 (match); otherwise it will receive a score of –1(mismatch), and if one of the two characters is a space, it will receive a score of –2 (gap), and the total score over the whole sequence is the score of this alignment. The optimal alignment problem is to find the maximal score of all possible alignments between two sequences. This maximal score can be used to measure the similarity between the two sequences. This scoring scheme is closely related to the concept of edit distance between two sequences introduced by Levenshtein in 1966 [5]. The Levenshtein edit distance from sequence A to sequence B is defined as the minimal number of edit actions to change A into B, where the edit actions are substitution, insertion, and deletion. Here mismatch corresponds to substitution, and gap corresponds to insertion or deletion. Both of them have negative score values. In order to maximize the total score of alignment, the number of mismatches and gaps in the alignment must be minimized.

Clearly, to solve this alignment score problem, an efficient algorithm other than simply generating all possible alignments between two sequences must be used. The number of possible alignments between two sequences increases exponentially as the sequences get longer. Needleman and Wunsch first introduced a dynamic programming algorithm for comparing two sequences in 1970 [8]. It is the basic algorithm for most pair-wise sequence alignment tools used today by molecular biologists. Because of various applications of sequence alignment, this algorithm has been discovered and re-discovered many times, including speech processing and text editing. The algorithm solves an instance of the problem by taking advantage of computed solutions for smaller instances of the same problem. To find optimal alignment score F[i, j] of two sequences s[1...i] and t[1...j], we can break it down into three smaller problems:

- Find optimal alignment score F[i, j-1] of s[1...i] and t[1...j-1], and
- Find optimal alignment score F[i-1, j-1] of s[1...i-1] and t[1...j-1], and
- Find optimal alignment score F[i-1, j] of s[1...i-1] and t[1...j].

If we know the solutions of these three smaller problems: F[i, j-1], F[i-1, j-1], and F[i-1, j], we can find optimal alignment score F[i, j] of s[1...i] and t[1...j] because optimal alignment of s[1...i] and t[1...j] can only be one of the following possibilities:

- Align s[1...i] with t[1...j-1] and match a space with t[j], or
- Align s[1...i-1] with t[1...j-1] and match s[i] with t[j], or
- Align s[1...i-1] with t[1...j] and match s[i] with a space.

The optimal alignment scores for each are:

• F[i, j-1] – 2	(-2 for a gap between a space and t[j])
• F[i-1, j-1] ± 1	(+1 for match if s[i] = t[j], -1 for mismatch if s[i] ≠ t[j])
• F[i-1, j] – 2	(-2 for a gap between s[i] and a space)

Then the optimal alignment score for s[1...i] and t[1...j] is the largest value of the three:

$$F[i,\ j] = MAX \begin{cases} F[i,\ j-1] - 2 \\ F[i-1,\ j-1] \pm 1 \\ F[i-1,\ j] - 2 \end{cases}$$

This algorithm uses a (i+1)*(j+1) size score matrix F. F[m, n] is the optimal alignment score for s[1...m] and t[1...n]. The values for F[0, 0], F[0, n], and F[m, 0] are known:

- F[0, 0] = 0 because it is the score of the alignment between two empty sequences.
- F[0, n] = -2*n, F[m, 0] = -2*m because they are the scores for aligning an empty sequence to another sequence of length n or m, resulting in a gap of length n or m.

Then the score matrix can be filled from F[1, 1], row by row, left to right in each row, or any other order that makes sure F[m, n-1], F[m-1, n-1], and F[m-1, n] are available when computing F[m, n]. After the whole score matrix is filled, F[i, j] will be the score for the optimal alignment of s[1...i] and t[1...j], and the optimal alignment can be recovered from the score matrix F by tracing which of the three choices was chosen to compute F[m, n] from F[i, j] all the way back to F[1, 1].

An example of this dynamic programming algorithm is shown in Figure 2. The two sequences in this example are GACGGATTAG and GATCGGAATAG. The score of the optimal alignment is shown in the right bottom corner of the score matrix F: F[10, 11]=6. The optimal alignment is also shown in the figure.

This algorithm has a space complexity proportional to the size of the score matrix F, which is $O(i \times j)$. Since the cells in the score matrix F must be filled one by one, this algorithm also has a time complexity of $O(i \times j)$, or $O(i^2)$ if the two sequences have nearly the same length, i.

Score matrix F:

		G	A	C	G	G	A	T	T	A	G
	0	-2	-4	-6	-8	-10	-12	-14	-16	-18	-20
G	-2	**1**	-1	-3	-5	-7	-9	-11	-13	-15	-17
A	-4	-1	**2**	0	-2	-4	-6	-8	-10	-12	-14
T	-6	-3	**0**	1	-1	-3	-5	-5	-7	-9	-11
C	-8	-5	-2	**1**	0	-2	-4	-6	6	-8	-10
G	-10	-7	-4	-1	**2**	1	-2	-4	-6	-7	-7
G	-12	-9	-6	-3	0	**3**	1	-1	-3	-5	-6
A	-14	-11	-8	-5	-2	1	**4**	2	0	-2	-4
A	-16	-13	-10	-7	-4	-1	2	**3**	1	1	-1
T	-18	-15	-12	-9	-6	-3	0	3	**4**	2	0
A	-20	-17	-14	-11	-8	-5	-2	1	2	**5**	3
G	-22	-19	-16	-13	-10	-7	-4	-1	0	3	**6**

Optimal alignment:

```
GA-CGGATTAG
|| |||| |||
GATCGGAATAG
```

Score for optimal alignment:

F[10,11]=6

Figure 2. Example of the Needleman-Wunsch algorithm for DNA sequence alignment.

If this algorithm is implemented on a parallel computing architecture, the cells in the score matrix can be filled in parallel instead of one by one, and thus the time complexity can be greatly reduced. For example, after F[1, 1] is computed, both F[1, 2] and F[2, 1] are ready to be computed because all the elements needed are there: F[1, 1] F[0, 1] F[0, 2] for F[1, 2] and F[2, 0] F[1, 0] F[1, 1] for F[2, 1]. After F[1, 2] F[2, 1] are computed, F[2, 3] F[2, 2] F[3, 2] are ready to be computed. This can be done from the top left corner to the bottom right corner in a wave front style and computation time can be cut to $O\left(\sqrt{i^2 + j^2}\right)$, which is the length of the diagonal of the score matrix F, or $O(i)$ if the two sequences have similar lengths.

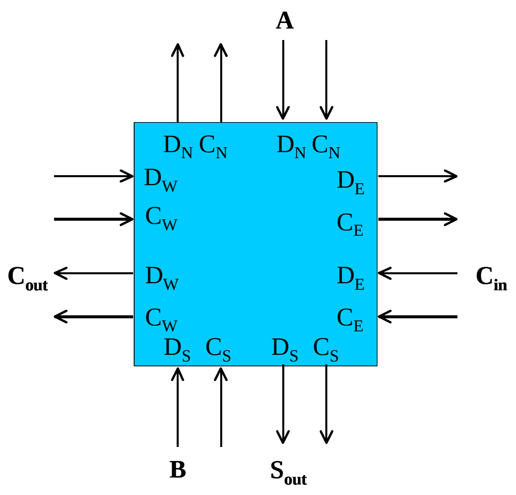
To implement this dynamic algorithm on a parallel computing architecture, a large number of processors would have to be used to compute the values in score matrix cells in parallel. The Cell Matrix™ architecture provides a good parallel computing platform to implement this dynamic algorithm.

Cell MatrixÔ Architecture

The Cell Matrix™ architecture is a nanocomputing platform developed by the Cell Matrix Corporation in Salt Lake City, Utah [3]. Unlike traditional CPU/memory architectures, the hardware of the Cell Matrix™ architecture is homogeneous. The basic element of a Cell Matrix is called a cell. Cells are repeated and interconnected to form the basic architecture of a Cell Matrix. Each cell accepts inputs from its neighbor cells, processes the inputs based on its internal reconfigurable memory, and then outputs the result to its neighbor cells. Cells in the architecture are physically identical, which simplifies the manufacture process. After manufacture, cells can be configured to perform different functions, and the collection of different configured cells can form various digital circuits to perform complex computations.

An example of a Cell Matrix cell is shown in Figure 3. This is a four-sided cell that forms a two dimensional Cell Matrix with each cell connected to four neighbors. Other topologies are also possible: six-sided cells can form a Cell Matrix with a honeycomb-like structure. Each cell has a data input, a data output, a control input, and a control output on each side. It also has an internal memory of 16 rows by eight columns. A cell can operate in one of two modes: data mode and control mode. In data mode, all of the control inputs are 0 and the cell is a pure combinatorial device. It uses its four data inputs and the internal memory to determine its outputs. In control mode, one of the four control inputs is 1. In this mode, the cell's internal memory can be reconfigured. The data input is serially shifted into the cell's internal memory according to a systemwide clock. After this cell returns to data mode, the reconfigured memory will control the cell's behavior. A cell can be configured to perform many different functions, such as a piece of wire, a logic gate, and more complex functions. The cell shown in Figure 3 is configured as 1-bit full adder. It adds data inputs from north, south, carry input from west, and outputs the result to south and carry to west. Several such configured cells can be linked together to form an n-bit adder. More complex circuits can be implemented on this Cell Matrix™ architecture, such as arithmetic logic units (ALU) and memory in a traditional CPU/memory architecture.

Because the cells can be individually configured, this architecture supports a one-problem, one-machine model of computing. The circuit can be specifically designed for the particular problem to be solved, and thus could solve the problem faster or better than



Cell's internal memory configuration:

Inputs				Outputs							
D_N	D_S	D_W	D_E	C_N	C_S	C_W	C_E	D_N	D_S	D_W	D_E
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	1	0	0
0	1	0	0	0	0	0	0	0	1	0	0
0	1	0	1	0	0	0	0	0	0	1	0
0	1	1	0	0	0	0	0	0	1	0	0
0	1	1	1	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0	1	0	0
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	0	0	1	0
1	1	0	0	0	0	0	0	0	0	1	0
1	1	0	1	0	0	0	0	0	1	1	0
1	1	1	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	1	1	0

Cell's memory expressed using Boolean equation:

```
DS = DN&DS&DE + DN&~DS&~DE + ~DN&DS&~DE + ~DN&~DS&DE
DW = DN&DS + DS&DE + DE&DN
```

Figure 3. A Cell Matrix cell configured as a 1-bit full adder.

an implementation on a general-purpose machine. This architecture also provides a good platform for distributed and parallel computing. Different parts of the hardware can be configured to perform different tasks, or similar tasks with different data. Thus, this problem can be distributed spatially rather than temporally achieving massively parallel computing. Because of its support for distributed and parallel computation, an extremely large number of switches can be efficiently implemented on a Cell Matrix™ architecture. An example is a DES cracker constructed using a Cell Matrix [3]. This DES cracker uses an array of custom processors and divides the search space efficiently among these processors. This design can achieve $O(1)$ run time in finding an encryption key. Another important feature of the Cell Matrix™ architecture is that a cell's internal memory can be reconfigured by other cells in the architecture. This leads to dynamic, self-modifying circuits. Some interesting examples of this feature include a ringed GA [6, 7], evolvable hardware, and run-time, fault-tolerant hardware.

The dynamic algorithm for DNA sequence alignment can also be efficiently implemented on a Cell Matrix™ architecture due to its support for distributed and parallel computing. The algorithm has a space complexity (number of processors) of $O(i^2)$ and a time complexity lower bound of $O(i)$ when implemented on a parallel computing architecture. The goal of this work is to present a sequence alignment system that achieves this lower bound time complexity and falls within the space complexity. This lower bound time complexity of $O(i)$ has not been achieved yet and no research has demonstrated such a sequence alignment machine. The approach taken in this work is to configure a block of cells in a Cell Matrix as a custom processor to perform the function of computing the values in the score matrix F. Parallelism is archived by linking these custom processors into a 2D processor array. Each of the processors constantly computes its value based on its inputs and outputs its result to the next processor. Sequence data are input from the top and left edges of the 2D array. When the processor array is stabilized, the right bottom processor unit contains the value of the optimal alignment score.

A different approach to this problem is to code the algorithm for a multiprocessor machine or a cluster of machines. However, to achieve the lower bound time complexity, an extremely large number of processors are needed. Processor latency and inter-processor communication delay will become a serious problem. The approach taken in this work allows a system design that closely resembles the original algorithm. And, by carefully designing the custom processor, the processor latency and interprocessor communication can be greatly reduced, resulting in a better constant factor for the time complexity. Also, because the custom processor is much simpler than a general purpose processor, the hardware requirement is reduced.

CHAPTER II

DESIGN AND IMPLEMENTATION

Processor Design

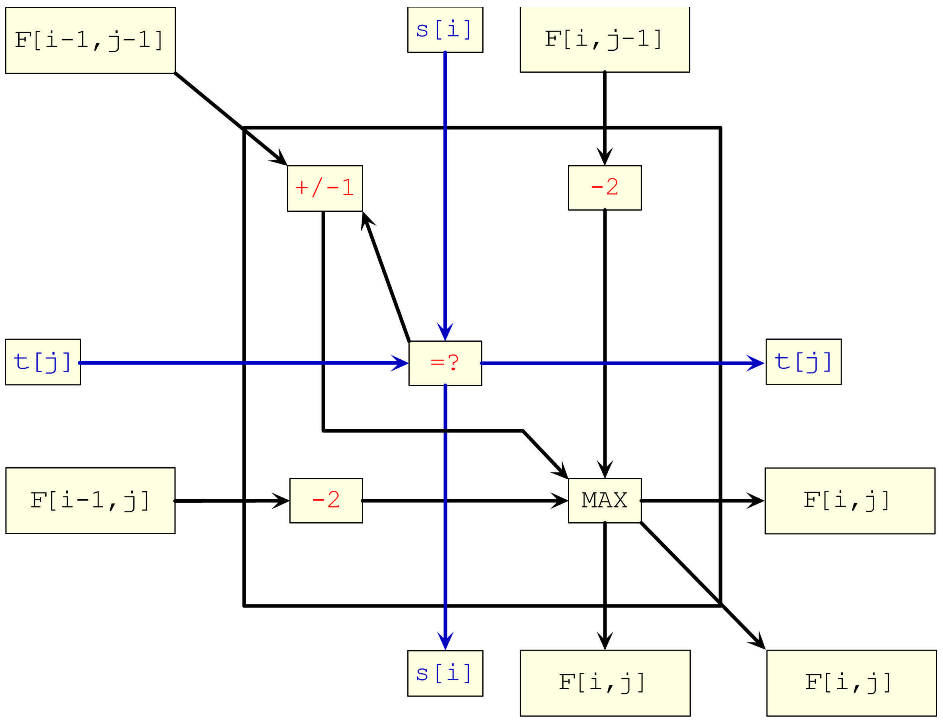
Implementation of this dynamic algorithm on a parallel computing architecture, such as a Cell Matrix, requires a 2D array of processors. Each processor is responsible for the value of one cell in the score matrix, and they work in parallel to compute the values in the score matrix. In the Cell Matrix™ architecture, a block of Cell Matrix cells can be configured as such a custom processor. Figure 4A shows the basic design of the custom processor unit. The processor unit has five inputs, two of which contain the sequence data s[i] and t[j]; the other three inputs are computed as score matrix cell values from its neighboring processors. The processor performs the following function:

$$F[i,\ j] = MAX\ \begin{cases} F[i,\ j-1]-2 \\ F[i-1,\ j-1]\pm 1 \\ F[i-1,\ j]-2 \end{cases}$$

The processor compares the sequence data inputs s[i] and t[j], and adds or subtracts one from F[i-1, j-1] according to the comparison result. It also subtracts two from F[i, j-1] and F[i-1, j]. These three results are then compared. The largest one is the value of F[i, j] and this value is passed to its neighboring processors, which use this value to compute their values.

There are several basic circuits in the processor. Each performs a simple function, and they work together to perform the function of computing a value for the score matrix. These basic circuits are shown as boxes inside the processor in Figure 4A. First, a sequence comparison unit is required to compare the sequence data s[i] and t[j].

A



B

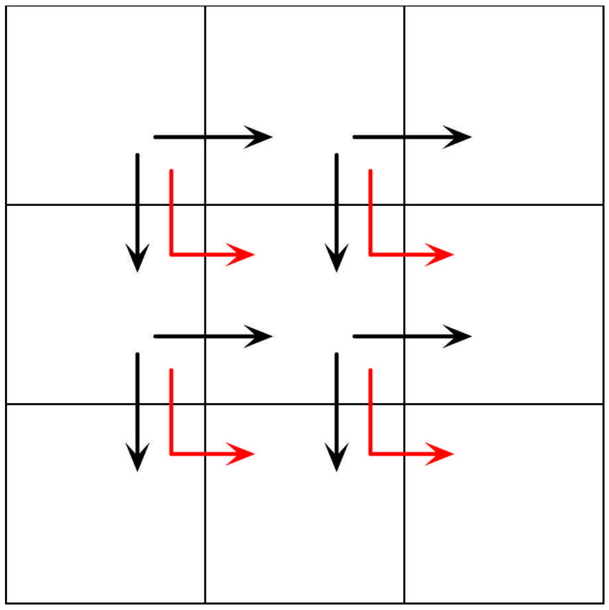


Figure 4. Custom processor design.

The result of this comparison is used to control the +/-1 unit, which adds or subtracts one from F[i-1, j-1] according to the result of the comparison. Two –2 units are used to subtract 2 from F[i, j-1] and F[i-1, j]. And finally a MAX unit is used to select the largest value and pass the value to its neighboring processors.

This design requires the processor exchanging information with its six neighbors, two of which are on its diagonal, F[i-1, j-1] and F[i+1, j+1]. However, four-sided cells in a Cell Matrix cannot directly pass information diagonally, because data inputs/outputs are only on the side of the cell and diagonally linked cells do not share sides or data inputs/outputs. So this kind of information exchange must be routed through their neighbors, and the scheme shown in Figure 4B is a method to do so. Here, each processor unit will have an extra input and an extra output connected directly to each other in its upper right corner. This connection can pass information from the neighbor above it to the neighbor on its right. Otherwise these two diagonally linked processors cannot exchange information.

In this design, different values in the sequence data are represented using 3 bits: 000 for A, 001 for G, 010 for C, and 011 for T. The first bit is reserved for other possible sequence data values, such as U (uracil) in the RNA sequence, or N for bad sequence read out. Values in the score matrix F are represented using 9 bits. The first bit is the sign bit. Negative numbers are represented in 2's complement format. This gives a range of -256 to +255, which is enough to align two DNA sequences up to 127 base pairs. This architecture can be easily scaled up for longer sequence alignment.

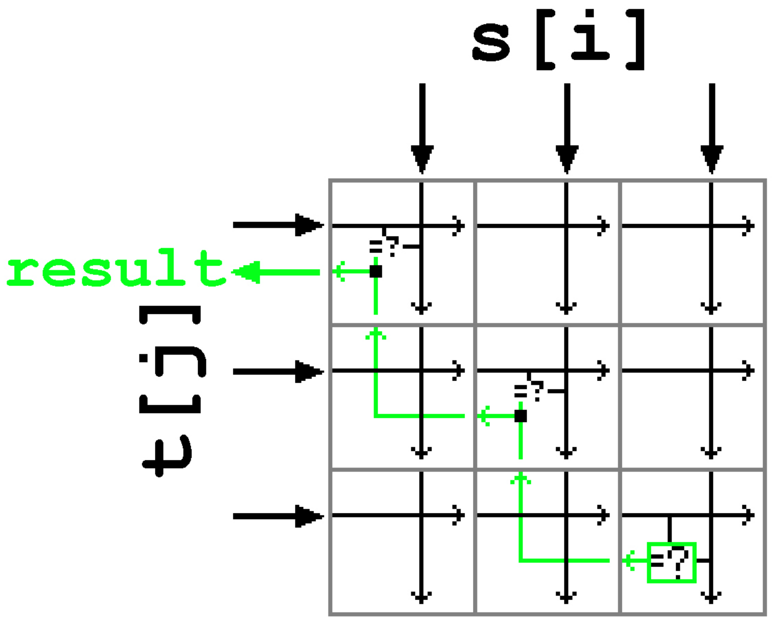
Implementation on Cell MatrixÔ Architecture

The custom processor was designed and implemented using the Cell Matrix Layout Editor™ development tool from the Cell Matrix Corporation. This Layout Editor provides a graphical user interface for designing Cell Matrix circuits. It allows the developer to lay out four-sided Cell Matrix cells in a 2D grid, edit the content/memory of individual cells, and create circuits using collections of differently configured Cell Matrix cells. It provides several other useful functions, such as copy and paste cells, rotate cells within the grid, and create icons to represent different cells. And finally, the grid containing the circuit can be written out as a .bin file, which can be read and tested using the Cell Matrix Simulator™.

The sequence comparison unit requires a 3x3 Cell Matrix block shown in Figure 5A. Sequence data inputs are on the top and left edges of the unit. Corresponding bits of the sequence data are compared, and the output of the unit is the logical AND of the three individual bit comparison results. This sequence comparison unit's output is 1 when two input sequence data are the same, and 0 otherwise. The truth table configurations for each of the nine cells are shown in Figure 5B. This sequence comparison unit also allows the sequence data to pass through, so that the custom processor can pass the sequence data to its neighboring processors.

Since all negative numbers are represented in 2's complement format, all addition and subtraction can be performed using adders. As shown in Chapter I, a Cell Matrix cell can be configured as a 1-bit full adder, and several such configured cells can be linked to form an n-bit adder. Figure 6A shows another configuration of a 1-bit full adder. Its icon

A



B

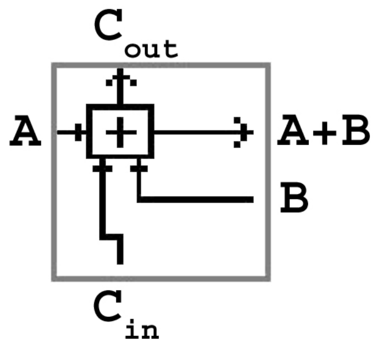
DS=N
DE=W
DW=(N&W + ~N&~W) & S
	DS=N
	DE=W
		DS=N
		DE=W

DS=N
DE=W
DN=E
	DS=N
	DE=W
	DW=(N&W + ~N&~W) & S
		DS=N
		DE=W

DS=N
DE=W
	DS=N
	DE=W
	DN=E
		DS=N
		DE=W
		DW=N&W + ~N&~W

Figure 5. Sequence comparison unit.

A



```
Truth table:
DE= W&E&S + W&~E&~S + E&~S&~W
    + S&~W&~E
DN= W&E + E&S + S&W
```

B

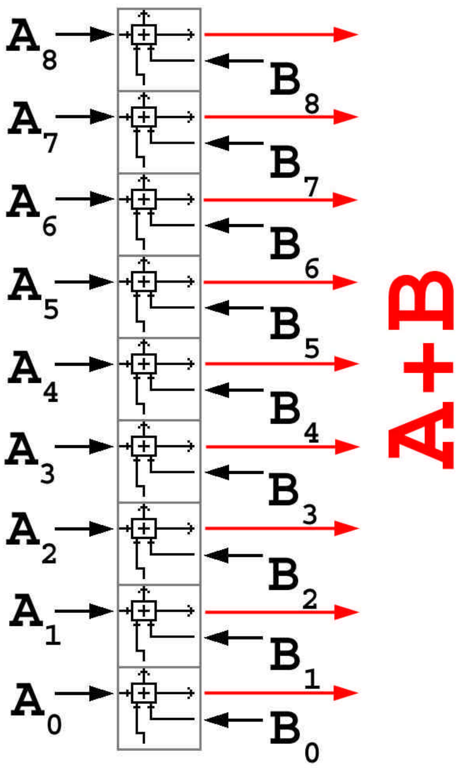


Figure 6. A Cell Matrix cell configured as a 1-bit full adder and a 9-bite adder.

is shown on the left and its truth table is shown on the right. It adds data inputs from west and east, carry input from the south, and outputs its sum to the east and carry to the west. The custom processor requires a 9-bit adder since all values are represented using nine bits. A 9-bit adder can be formed by stacking nine 1-bit full adders vertically, as shown in Figure 6B. Data inputs are on its left and right, with high order bit on the top, low order bit on the bottom. Carries are passed from bottom to top. The adder sends the results to its right. The +/-1 unit and –2 unit are all based on this 9-bit adder.

The +/-1 unit adds 1 or –1 to an input value according to the output of the sequence comparison unit. The sequence comparison unit outputs 1 when s[i]=t[j], and the processor adds 1 to F[i-1, j-1]. Otherwise, sequence comparison unit outputs 0, and the processor adds –1 to F[i-1, j-1]. In the +/-1 unit, F[i-1, j-1] is one of the two inputs of the 9-bit adder. The other input of the adder is either 1 or –1, and this is controlled by the sequence comparison unit. This input should be 1 when the sequence comparison unit's output is 1 (s[i]=t[j]), and –1 when the output is 0 (s[i]≠t[j]). This can be achieved by setting the eight higher order bits to the inverse of the output of the sequence comparison unit and lowest order bit to 1, because 1 is 000000001 and –1 is 111111111 when represented using nine bits and in 2's complement form. A block of nine cells can be configured to perform such a function as shown in Figure 7A. These nine cells send a 9-bit value to the adder on its left. The control input is on the top and passed to all nine cells. The eight higher order bits are the reverse of the control input, and the lowest bit is set to one at all times. The 9-bit value sent to the adder is 000000001 (=1) when control input is 1, and 111111111 (=-1) when control input is 0. The adder adds this value to the F[i-1, j-1] input on its left, and sends the result to its right.

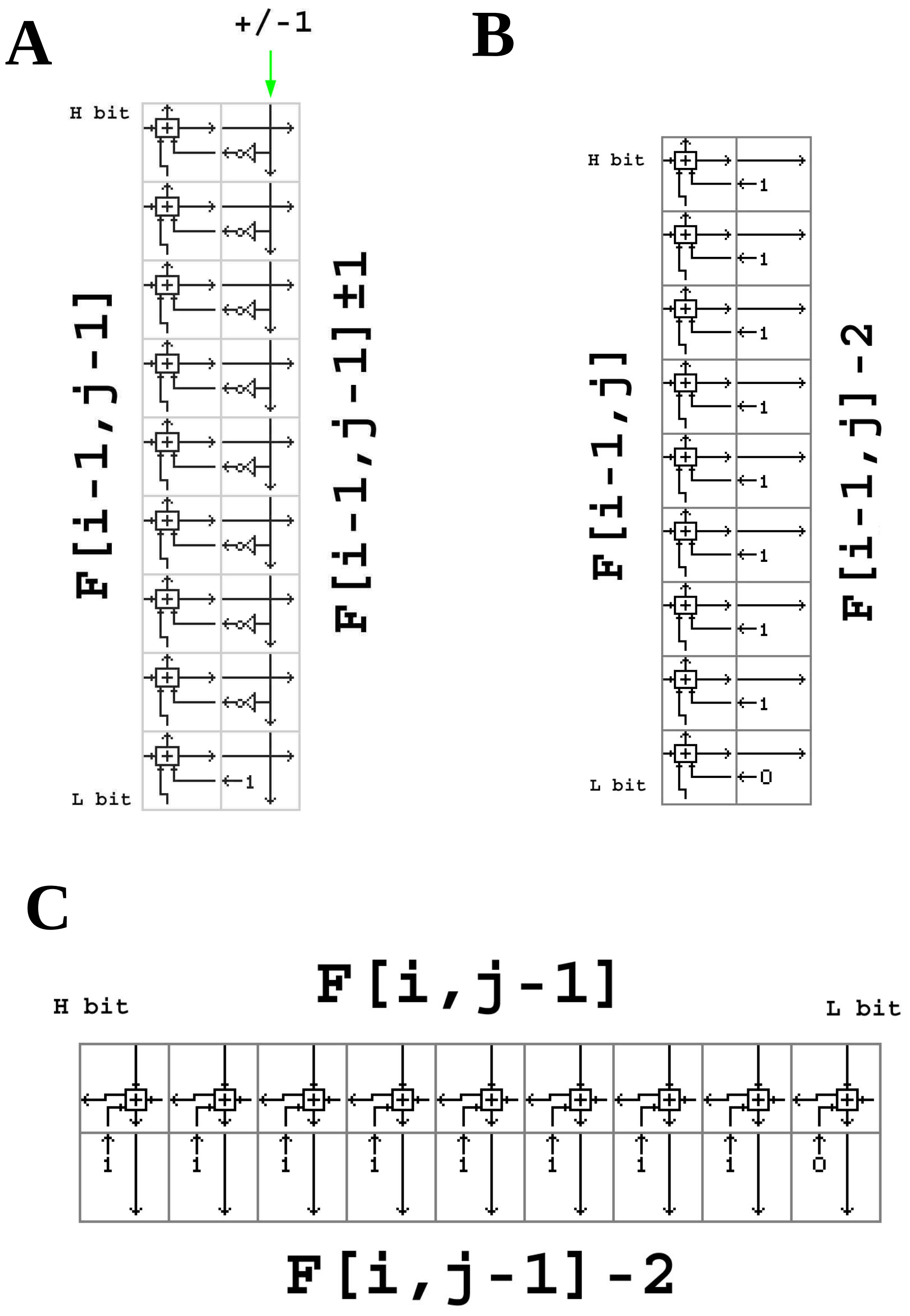
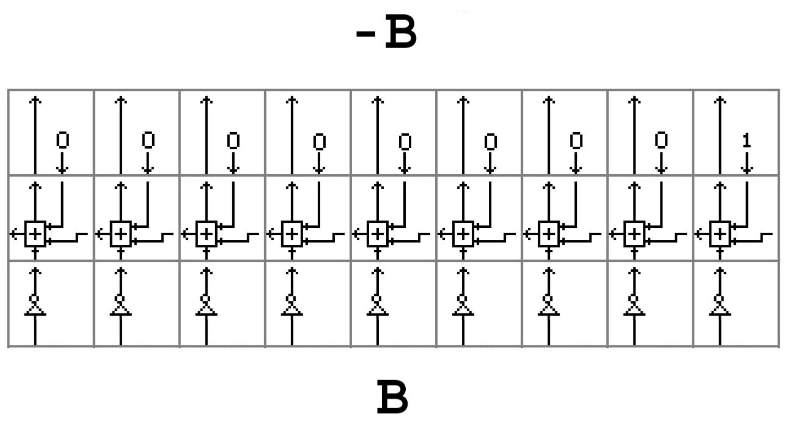


Figure 7. +/-1 unit and –2 unit design.

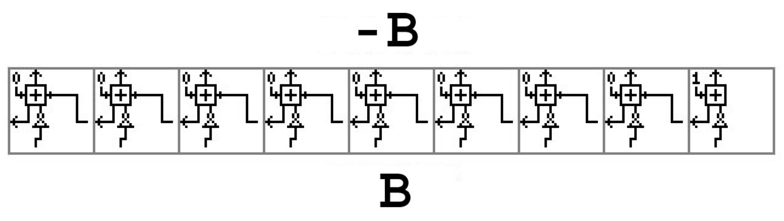
The –2 unit is very similar to the +/-1 unit. Because it always subtracts 2 from its input, there is no need for a control input. The adder in this unit will always have an input of –2. The 2's complement form of -2 is 111111110. The nine cells used to send +/-1 to the adder can be configured to send this value to the adder at all times. The other input to the adder will be F[i-1, j] from the neighboring process to the left, or F[i, j-1] from the neighboring processor above. Figure 7B shows a vertically configured –2 unit with high order bit on the top and low order bit on the bottom. This is used for F[i-1, j], which is passed to the unit from the left. Since F[i, j-1] comes from the top, a horizontally configured –2 unit in Figure 7C is used. The input is on the top of the unit with high order bit on the left and low order bit on the right, and output is on the bottom of the unit.

The sign bit of A-B can be used to compare A and B. If the sign bit of A-B is 0, then A-B is positive and $A \geq B$. If the sign bit of A-B is 1, then A-B is negative and $A<B$. A-B can be performed using an adder if –B is represented in 2's complement form. To represent –B in 2's complement format, every bit in B is inverted and then 1 is added. This operation can be performed using a modified adder as shown in Figure 8A. Data input is at the bottom of the unit with high order bit on the left and low order bit on the right. The bottom nine cells invert all nine bits in the data input and pass them to the nine-bit adder in middle of the unit. The top nine cells send 000000001 to the adder. The adder in the middle adds this 000000001 to the inverted data input and sends result to the top. The function of this 9x3 cellblock can also be performed using just nine cells, as shown in Figure 8B. Another adder can be added to this unit to perform the function of A-B, as shown in Figure 8C. Data B input is on the bottom of the unit. After data pass

A



B



C

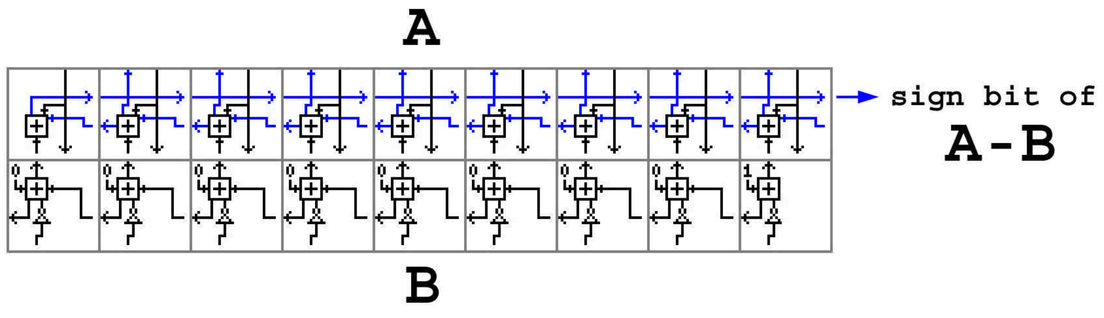


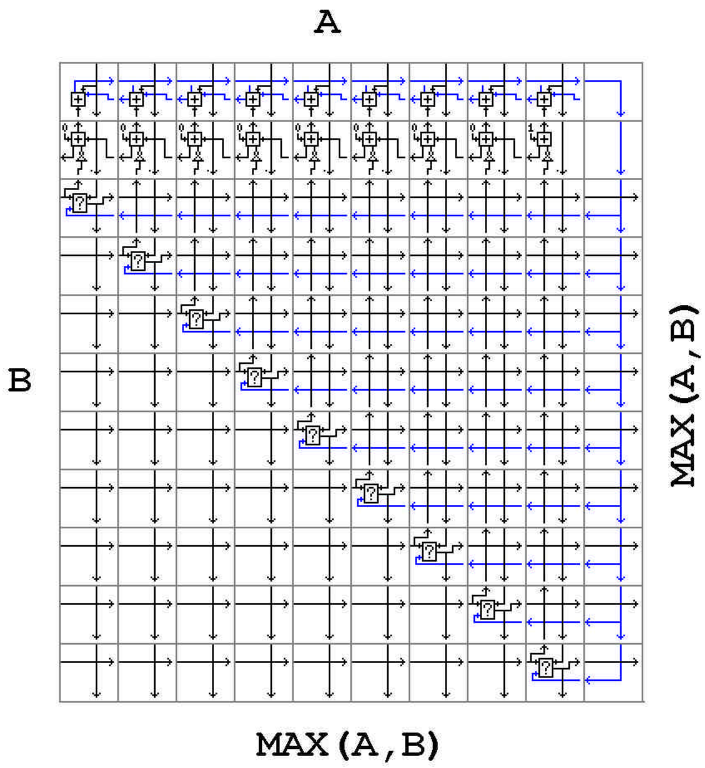
Figure 8. Modified adder to perform the function of A-B.

through the bottom nine cells, B becomes –B. –B is then passed to the adder on top. Data A input is on the top of the adder. The adder adds A to –B. The sign bit on the very left shows which number is larger. This sign bit is the output of this unit and passed to the right side. This output can be used to select the larger of the two inputs.

Figure 9A shows a MAX selection unit. This unit is composed of a 9x2 A-B unit on the top and a 9x9 selection unit on the bottom. Two data inputs A and B are on the top and left of the unit. Data B is passed to the A-B unit by the selection unit. The sign bit of A-B is passed to the selection unit as a selection signal. This signal is 0 when A≥B and 1 when A<B. The cells on the diagonal have three inputs: A from the top, B from the left and the selection signal from the right. It also passes B from its left to the A-B unit on the top. The output of these cells depends on the selection signal. If A≥B, they will receive selection signal 0 and output A. Otherwise, they output B. Thus, this MAX unit can be used to select the larger of its two inputs. To select the largest value from three numbers, as required in this custom processor, two of these MAX units can be linked together, as shown in Figure 9B. A and B are the inputs of the first MAX unit. The output of this unit is MAX (A, B). MAX (A, B) and C are the inputs of the second MAX unit. The output of the second MAX unit is MAX (A, B, C).

The Cell Matrix implementation of the custom processor is shown in Figure 10. It is composed of the basic processing elements shown in previous figures. The processor has six inputs and six outputs. Sequence data s[i] and t[j] are compared in the sequence comparison unit at the top left corner of the processor. The comparison result is then linked to the control input of the +/-1 unit. F[i-1, j-1] input is on the middle of the left edge and linked to the +/-1 unit. The output of +/-1 unit, F[i-1, j-1]±1, is connected to the

A



B

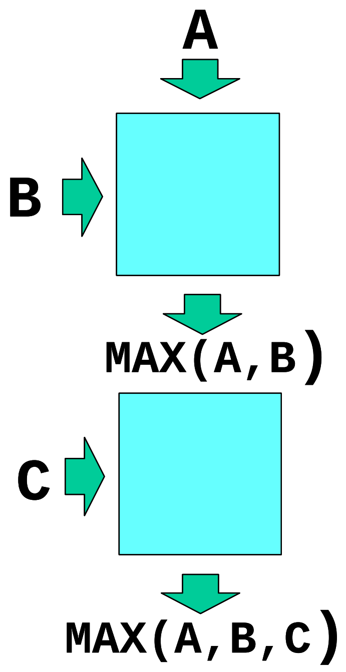


Figure 9. MAX selection unit.

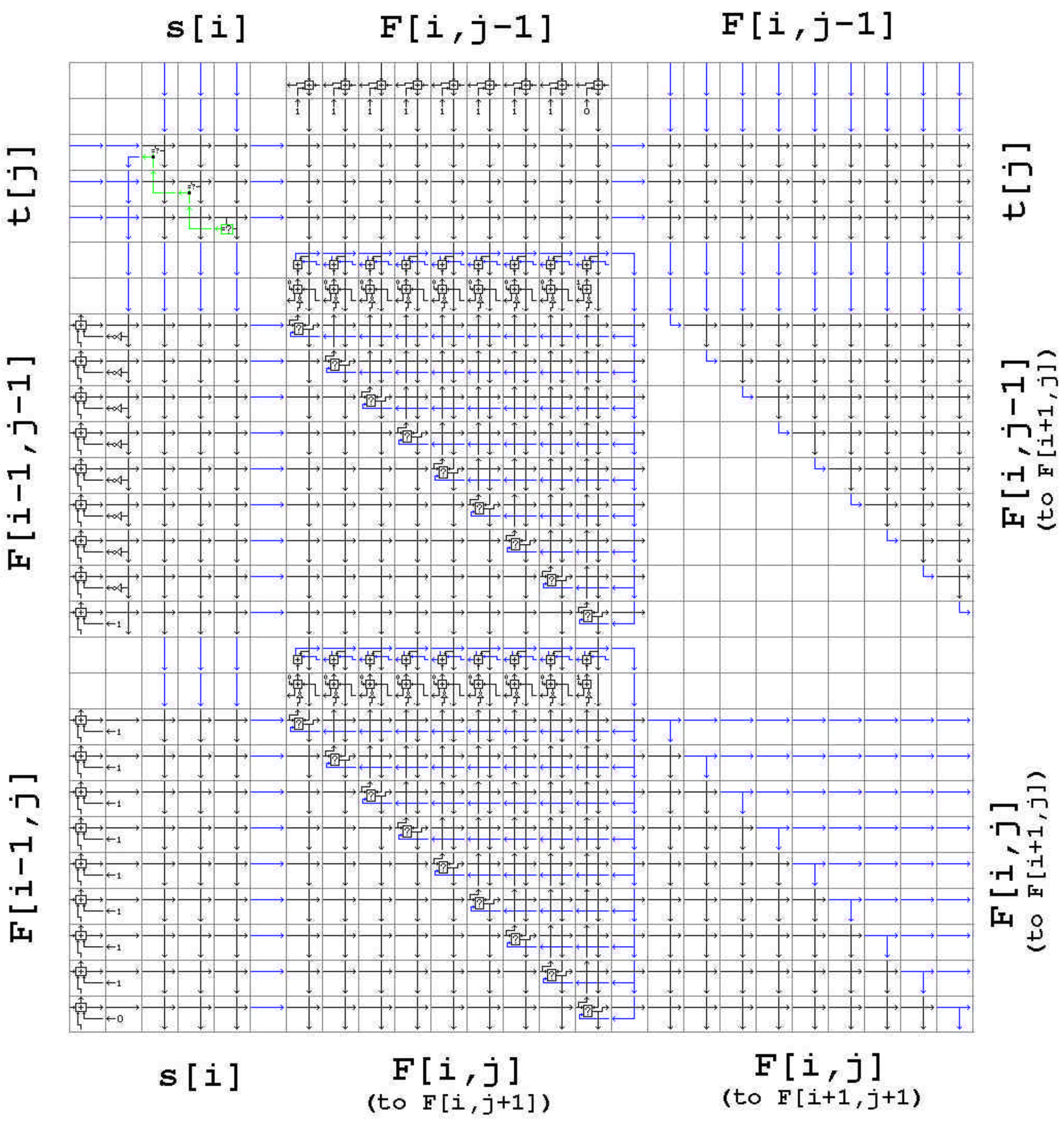
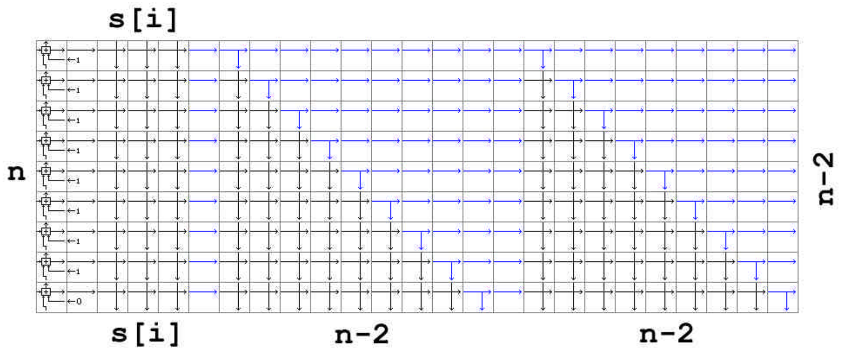


Figure 10. Cell Matrix implementation of the custom processor.

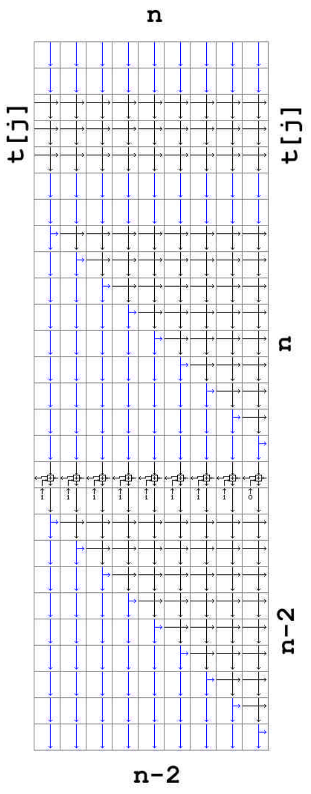
MAX selection unit. F[i, j-1] input is on the middle of the top edge. It is connected to a horizontally configured –2 unit. F[i, j-1]-2 is then connected to the MAX selection unit. F[i-1, j] inputs is also connected to a –2 unit, F[i-1, j] is the third input of the MAX selection unit. The output of the processor is the output of the MAX selection unit. On the top right corner is a direct pass for F[i, j-1]. This is necessary as processor [i, j-1] and [i+1, j] cannot communicate directly.

A 2D array using this custom processor configuration is thus able to perform DNA sequence alignment. As discussed in Chapter I, the algorithm requires that F[0, 0], F[0, n], and F[m, 0] to be initialized to F[0, 0]=0, F[0, n]= -2*n and F[m, 0]= -2*m. Cell Matrix circuitry can be added to the top and left edges of the 2D processor array to initialize these values. The repeating units of the circuitry are shown in Figures 11A and B. They use the -2 unit discussed previously to automatically generate the values required for initialization. The 2D processor array together with these initialization circuits is shown in Figure 11C. The two sequences are inputs on the top edge and left edge of this 2D processor array, and the right bottom corner of the processor array will have the optimal alignment score of these two sequences.

A



B



C

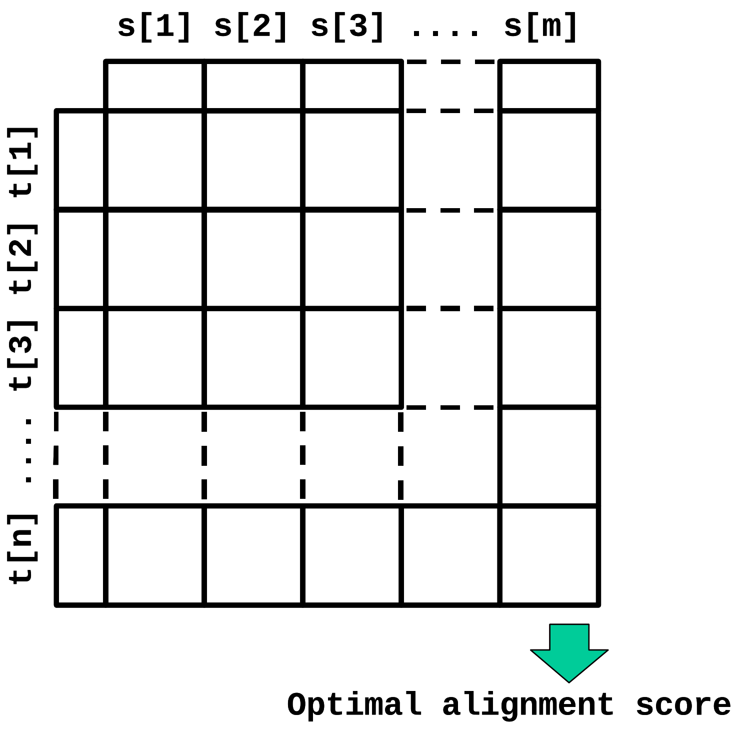


Figure 11. Cell Matrix circuits for initializing the processor array and the layout of the 2D processor array.

CHAPTER III

RESULTS AND CONCLUSIONS

Testing Results

The testing of the design and implementation was done by first testing the low-level components of the processor, and then testing the processor array using a representative sample of input sequences.

All the circuitries used in the custom processor and the custom processor itself were tested using the Cell Matrix Simulator™ from the Cell Matrix Corporation. This simulator is a testing and debugging tool. After a circuit is designed using the Layout Editor, the configurations of the cells in the circuit are written into a binary grid file, and then read into the simulator. The simulator provides a graphical user interface for the developers to test their circuit with different inputs, observe the behavior of the circuit, and modify the configuration of individual cells for debugging.

Figure 12 shows the circuit used in the MAX unit to obtain the sign bit of A-B. Each box in the figure represents a Cell Matrix cell. The arrows represent inputs and outputs of the cells - thin lines are 0's and thick lines are 1's. The inputs of the cells at the top left edge of the matrix can be changed by clicking on the arrows. Changes in the input are propagated through the circuit. Input B, on the bottom of the circuit, cannot be changed directly. Additional circuits were added to route B to the left edge of the matrix. In this figure, input A is 000110011 or 51, and input B is 001101010 or 106. The output at the top right corner shows the sign bit of A-B as 1, indicating A<B. Similar tests were done on other components of the processor to make sure they functioned as designed.

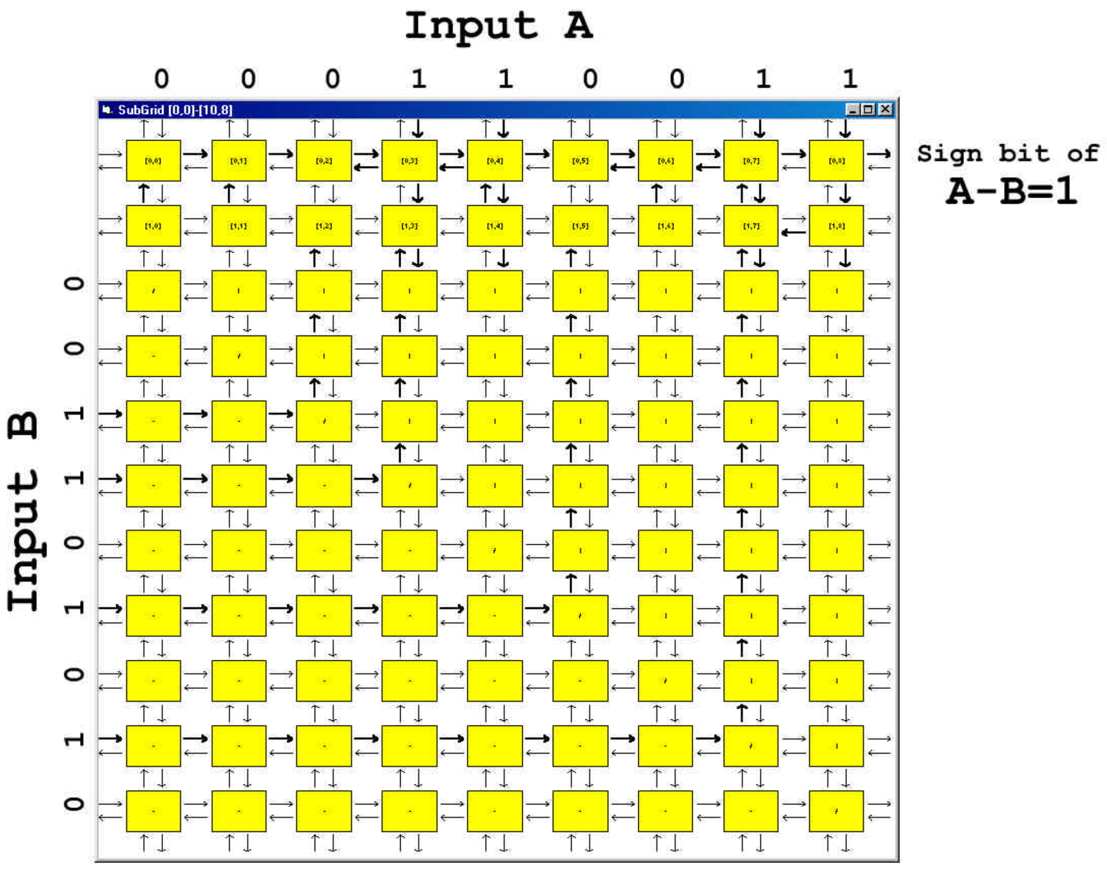


Figure 12. Cell Matrix circuit for A-B tested using a Cell Matrix Simulator™. A graphical window from the Cell Matrix Simulator™, thin lines are 0's and thick lines are 1's. Input A 00011011 is on the top and input B 001101010 is on the left, sign bit of A-B, which is 1, is on the top right corner.

The custom processor was configured into a 2D processor array as shown in Figure 11C to perform sequence alignment. This required a very large number of Cell Matrix cells ($675 \times n^2$, n is the length of the sequence). The simulator with its graphical user interface is designed for a small number of Cell Matrix circuits. It works well for testing and debugging such circuits, but it is slow for large circuit simulations. Another simulator designed for large-scale simulation is available from the Cell Matrix Corporation. This simulator provides similar functions through a command line interface and supports batch execution of commands stored in files. Useful commands include: l, load cell configuration stored in a binary grid file into the simulator; s, set the input of a specific cell (this can be used to set the sequence input); p, show the input/output of a block of cells (this can be used to exam the output of the processor array). This command line driven simulator can handle much larger Cell Matrix circuits and the simulation is much faster. The 2D processor array was tested with different sequence pairs, such as the example shown in Chapter I. The processor array correctly computed the optimal alignment scores for all sequence alignments tested (Figure 13). These tests showed that the design and implementation of the custom processor are correct.

Timing of the Alignment

The goal of this research was to design a Cell Matrix circuit that could find the optimal alignment score of two DNA sequences in $O(n)$ time. To obtain timing information about the sequence alignment circuit, the command line driven simulator was modified by Nicholas J. Macias of the Cell Matrix Corporation to incorporate the capturing of time units and timing results. After the input to the circuit is changed, i.e., a

Input sequences	Optimal alignment	Optimal alignment score (computed by the processor array)
GGCTTTA GGCTTTA	GGCTTTA ||||||| GGCTTTA	**7**
GGCTTTA TGCTTTA	GGCTTTA |||||| TGCTTTA	**5**
GGCTTTA GGTTTA	GGCTTTA || |||| GG-TTTA	**4**
GGTTTA GGCTTTA	GG-TTTA || |||| GGCTTTA	**3**
GACGGATTAG GATCGGAATAG	GA-CGGATTAG || |||| ||| GATCGGAATAG	**6**

Figure 13. Testing of the custom processor array.

new pair of sequences to align, the changes are propagated through the Cell Matrix circuit. It then takes some time for the circuit to stabilize. This time was measured by the simulator. The unit of this measurement, represented using symbol “t”, is the propagation delay of a single Cell Matrix cell, or the time for a Cell Matrix cell to respond to an input change. More formally, the measurement is of the time from input change to stable output change in the cell.

For a DNA sequence alignment circuit, the optimal alignment score was obtained after the processor array had stabilized. Thus, the time for the circuit to stabilize is the time for the circuit to compute the optimal alignment score. This processor array is guaranteed to stabilize, since the processors perform simple combinational logic and the communication between processors is also formed by simple combinational logic. Measuring this time for processor arrays of different sizes shows how the computation time increases with the sequence length.

The test timing results are shown in Figure 14. The time to compute the optimal alignment score increases linearly with the length of the input sequences: $T \approx 80t \times n$, where T is the total compute time, t is the time delay for a single Cell Matrix cell, and n is the length of input sequences. The results show that this design achieved the goal of finding the optimal alignment score for two sequences in $O(n)$ time.

Discussion

This implementation of the dynamic programming algorithm is the first sequence alignment system that achieves $O(n)$time complexity. The major cost here is hardware, not the computing time. Instead of computing the values in the score matrix F for two

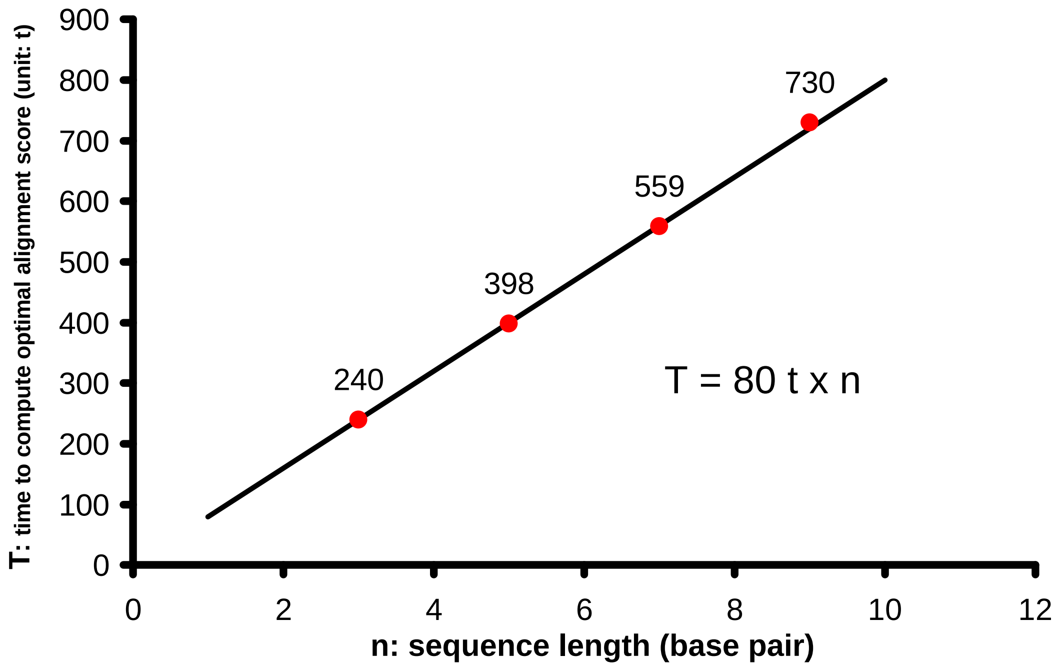


Figure 14. Timing of the sequence alignment.

sequences of length n one by one using a traditional CPU/memory computer with its single processor, a 2D $n \times n$ array of simpler custom processors was used. The problem of finding the optimal alignment score was broken down into smaller tasks. These smaller tasks were then distributed spatially over the processor array and performed in parallel by individual processors in the array. As the length of the sequence increases, the size of the processor array, or the hardware cost, also increases. Because the hardware cost increases with a complexity of $O(n^2)$, hardware cost becomes an important issue for long sequence alignments.

DNA sequences encoding protein domains are typically about 1000 base pairs long. Aligning two such sequences would require a processor array of size 10^6. Each custom processor is 675 Cell Matrix cells in this design. So the processor array would require about 7×10^8 cells. Current silicon techniques will scale up to about 500,000 cells in a single chip [3], which is only enough to align two sequences of 30 base pairs each. Thus, to use the Cell Matrix™ architecture, much denser manufacturing techniques are required [3]. With the advent of nanotechnology, biology-based computing, and other molecular engineering techniques, extremely large Cell Matrix configuration will be both possible and practical [3]. The development of the nanocircuit has just been named the breakthrough of the year by *Science* magazine [9]. Examples of the experimental nanocircuit include carbon nanotube transistors [1] and logic gates made of nanowires [4]. Although these nanocircuits are still very simple and rudimentary, they show feasibility of nanocomputing. When the technology for manufacturing extremely dense circuits becomes available, the amount of hardware required will be acceptable, and trading hardware for reduced computing time will be reasonable.

The timing result given by the Cell Matrix simulator is in terms of an abstract unit t, which is the typical delay of a single Cell Matrix cell. This sequence alignment system is a pure combinatorial circuit, and the computing time is a pure function of this propagation delay t. This time t will depend on the underlying technology. By building a small Cell Matrix on top of a FPGA and constructing a 128-cell feedback loop, the Cell Matrix Corporation has shown that this single cell propagation delay t is approximately 3.9 ns. Using this value, the computing time for alignment is about 80t per base pair, or about 0.3 μs per base pair. And this is an upper bound, since a FPGA implantation is slower than custom ASIC, and likely slower than future technologies. If a large enough Cell Matrix were available, it would align two sequences of 3 million base pairs each in under one second. Of course, such a Cell Matrix would be extremely large, i.e., $\approx 10^{14}$ cells.

An interesting observation in the testing is that the Cell Matrix™ architecture is a powerful parallel computing architecture. The simulator runs on a single CPU computer, and handles a single event at a time. Thus, events that should happen at the same time, or parallel events, are put into an event queue with a time tag. The size of the queue limits the maximum number of parallel events the simulator can handle. During testing, a Cell Matrix circuit used to align two sequences with five base pairs each overflows an event queue with 2 million events entries. And the circuit is a relatively small one, with less than 20,000 cells. This really shows the power of Cell Matrix™ architecture dealing with parallel computing problems.

Summary and Future Work

This research work implemented a dynamic algorithm for DNA sequence alignment on the Cell Matrix™ architecture and achieved $O(n)$ time complexity in finding the optimal alignment score. This work also showed that Cell Matrix™ architecture is a powerful parallel computing architecture.

Another important problem after finding the optimal alignment score is to recover the actual alignment from the score matrix F. Solving this problem will not only show how similar two sequences are, but also where the similarities are. This would be an interesting work based on this implementation. It would also be interesting to include some improvements of the alignment algorithm into the design, such as a nonlinear gap function (a big penalty for gap opening and a small penalty for gap extension, which is more appropriate for biological evolutions), and a more complex scoring system for protein sequence alignment.

REFERENCES

1. Bachtold, A., Hadley, P., Nakanishi, T., and Dekker, S. Logic circuits with carbon nanotube transistors. Science 294 (2001), 1317-1320

2. Doolittle, R.F., Hunkapiller, M.W., Hood, L.E., Devare, S.G., Robbins, K.C., Aaronson, S.A., and Antoniades, H.N. Simian sarcoma virus onc gene, v-sys, is derived from the gene encoding a platelet-derived growth factor. Science 221 (1983), 275-277

3. Durbeck, L. and Macias, N. The Cell Matrix: an architecture for Nanocomputing. Nanotechnology 12 (2001), 217-230

4. Huang, Y., Duan, X., Cui, Y., Lauhon, L., Kim, K., and Lieber, C. Logic gates and computation from assembled nanowire building blocks. Science 294 (2001), 1313-1317

5. Levenshtein, V.I. Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 6 (1966), 707-710

6. Macias, N. Ring around the PIG: a parallel GA with only local interactions coupled with self-reconfigurable hardware platform to implement an O(1) evolutionary cycle for EHW. The 1999 Congress on Evolutionary Computing (1999), 1067-1075

7. Macias, N. The PIG paradigm: the design and use of a massively parallel fine grained self-reconfigurable infinitely scalable architecture. Proceedings of the First NASA/DoD Workshop on Evolvable Hardware (1999), 175-180

8. Needleman, S.B. and Wunsch, C.D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology 48 (1970), 443-453

9. Service, R. Molecules get wired. Science 294 (2002), 2442-2443

10. Waterfield, M.D., Scrace, G.T., Whittle, N., Stroobant, P., Johnsson, A., Wasteson, A., Westermark, B., Heldin, C.H., Huang, J.S., and Deuel, T.F. Platelet-derived growth factor is structurally related to the putative transforming protein p28sis of simian sarcoma virus. Nature 304 (1983), 35-39