Prototyping Parallel Sequence Edit-Distance Algorithms in FPGA Hardware

David Churchill, Paul Gillard, Matthew Hamilton, and Todd Wareham

Abstract— **Over the last 15 years, genome projects worldwide have been producing ever-larger biological sequence datasets. One approach to dealing with these massive datasets has been to implement parallel analysis algorithms in special purpose hardware. Field Programmable Gate Arrays (FPGA) are ideal testbeds for designing and prototyping such hardware; however, there are a number of problems that can arise. We illustrate some of these problems relative to our experience in using the Xilinx Rapid Prototyping System to implement and test the parallel sequence edit-distance algorithm proposed by Lipton and Lopresti (1985).**

Keywords— **Parallel Algorithms, Edit Distance, Field Programmable Gate Arrays (FPGA).**

I. Introduction

Over the last 15 years, genome projects worldwide have been producing ever-larger biological sequence datasets. One approach to dealing with these massive datasets has been to develop parallel analysis algorithms. There are several techniques for parallelizing algorithms, ranging from general-purpose processor clusters to special-purpose hardware (see [3], [5], [10], [11] and references). The latter is of particular interest because it is very cost-effective for individual researchers. Field Programmable Gate Arrays (FPGA) are ideal testbeds for designing and prototyping such hardware; however, though many successes have been reported in the literature (see [10], [11] and references), there are a number of problems can arise for those wishing to design such hardware, some related to the application and some to idiosyncrasies of the FPGA prototyping systems themselves.

In this paper, we will illustrate some of these problems relative to our recent experience in using the Xilinx Rapid Prototyping System to implement and test the parallel sequence edit-distance algorithm proposed by Lipton and Lopresti [5]

Department of Computer Science, Memorial University of Newfoundland, St. John's, NL, Canada A1B 3X5

This work was supported by the Natural Sciences and Engineering Research Council of Canada through operating grant 228014 (TW) and by the Canadian Microelectronics Corporation (PG).

II. Background

A. The Sequence Edit-Distance Problem

Given two sequences of symbols over an alphabet, many applications require some measure of the similarity of (or alternatively, the distance between) those sequences. The most common manner of structuring such measures is in terms of sequence mutation operators. The three most basic operators involve changes to single symbols:

- **mutation**: change one symbol to another, *e.g.*, $A \rightarrow T$.
- **insertion**: insert a symbol, *e.g.*, $\epsilon \rightarrow T$.
- **deletion**: delete one symbol, *e.g.*, $A \rightarrow \epsilon$.

Each of these operations can in turn have associated weights. The **(weighted) edit distance** between two sequences s_1 and s_2 is defined as the weight of the sequence of mutation, insertion, and deletion operations that transforms s_1 into s_2 and has minimum summed operation-weight over all possible such operation-sequences.

The classical dynamic programming algorithm for computing edit distance (see [2, Chapter 11] and references) can be stated as follows: Let the weights of the mutation, insertion, and deletion operators be w_m, w_i, and w_d, respectively, and let $d(i,j)$, $0 \leq i \leq |s_1|$ and $0 \leq j \leq |s_2|$, denote the edit distance between the strings composed of the first i symbols of s_1 and the first j symbols of s_2. This quantity $d(i,j)$ can be computed by the recurrence

$$d(i,j) = \begin{cases} i & j = 0 \\ j & i = 0 \\ \min \begin{cases} d(i-1,j-1) \\ \quad +C(i,j), \\ d(i-1,j) + w_i, \\ d(i,j-1) + w_d \end{cases} & \text{otherwise} \end{cases}$$

where $C(i,j) = 0$ if the ith symbol of s_1 is the same as the jth symbol of s_2 and w_m otherwise. In the remainder of this paper, we will follow the literature, *e.g.*, [5], [11], by focusing on the restricted case where $w_m = 2$ and $w_i = w_d = 1$. All $d(i,j)$ values can be stored in a 2-D table in which $d(|s_1|, |s_2|)$ contains the edit distance between s_1 and s_2. The question then becomes, how do we fill in this table? The recurrence specifies the 0th row and column cell-values; the key

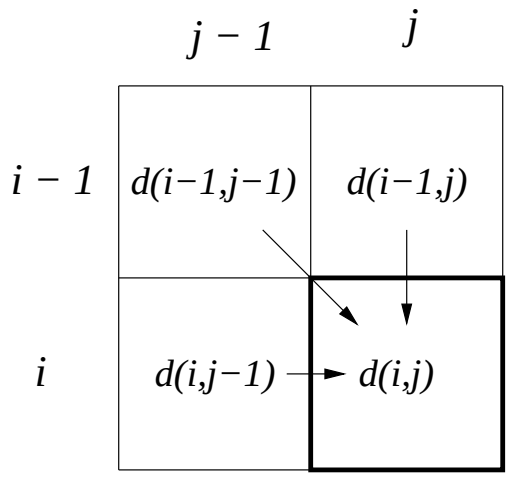

Fig. 1. Recurrence Cell Relationships

		S2	A	T	G	G	A
		0	1	2	3	4	5
S1	0	0	1	2	3	4	5
T	1	1	2	1	2	3	4
C	2	2	3	2	3	4	5
G	3	3	4	3	2	3	4

Fig. 2. Table Fill-In: Example #1

is then to recognize that this recurrence also allows us to fill in a given cell if the values for the cells immediately one cell up, to the left, and diagonally have already been filled in (see Figure 1). Starting from cell (1,1), one can then fill in the remainder of the table by row, column, or diagonal, working outwards one cell at a time from cell (1,1). Two examples of such filled-in tables are given in Figures 2 and 3.

B. Parallel Sequence Edit-Distance Algorithms

As noted at the end of the last section, there are a variety of orders in which the cells in an edit-distance table can be filled in, each of which constructs the value for a line of cells by using the values in the previous line of cells relative to the selected order. The observation that the values of the cells in each such line depend only on the values of the cells in the

		S2	T	G	G
		0	1	2	3
S1	0	0	1	2	3
A	1	1	2	3	4
C	2	2	3	4	5
G	3	3	4	3	4

Fig. 3. Table Fill-In: Example #2 (adapted from [5])

previous line is the basis for various linear-space implementations of the algorithm described above (see [2, Section 12.1] and references). However, if more than one cell-value can be computed at a time, this observation is also the basis for many parallel edit-distance algorithms.

A naive parallel algorithm implementing this strategy would use an $(|s_1|+1) \times (|s_2|+1)$ matrix of processors and propagate the values through the matrix according to the recurrence by diagonal starting at cell (1,1). At each timestep, processor (i,j) in the diagonal being computed would use the values stored in processors $(i-1,j)$, $(i-1,j-1)$, and $(i,j-1)$ in the diagonal computed in the preceding timestep (see Figures 1 and 4). As there are $|s_1|+|s_2|-1$ such diagonals, this algorithm runs in linear parallel as opposed to quadratic serial time. However, it uses a quadratic number of processors, which may be impractical when s_1 and s_2 are large.

Using the observation mentioned at the beginning of the chapter, we can reduce this to a linear number of processors, in particular, the $2\min(|s_1|,|s_2|)+1$ processors required to encode the largest diagonal in the matrix (or rather, the largest "crooked" diagonal composed of adjacent diagonal-pairs). The values stored in these processors at the beginning of a particular timestep correspond to the values stored along a particular diagonal of the table, and the job of each processor during the timestep is to compute the values of a particular cell in the next diagonal.

The first and simplest such scheme was proposed by Lipton and Lopresti in 1985 [5] (see Figure 5). The processors are organized as a two-channel two-way systolic array [7] in which each processor can communicate with the processors to its immediate left and right in the array. The symbols in s_1 (interpolated

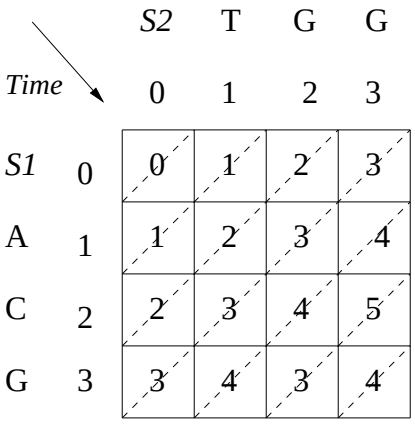

Fig. 4. Table Fill-In: Example #2 (Parallel) (adapted from [5])

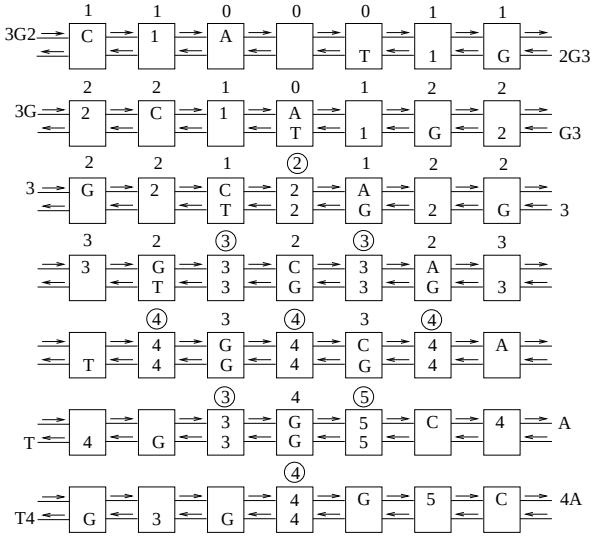

Fig. 5. Systolic Array Computation (adapted from Figure 1 in [5])

with integers indicating the sequence-position of each symbol) are fed in from the left of the array on the upper channel and a similarly-interpolated version of s_2 is fed in from the right on the lower channel; this corresponds to setting up the 0th row and column in the dynamic programming table. Once values in the two channels overlap in a processor, two situations are possible: the channels specify one or more numbers or a pair of symbols. Processors in which numbers are present correspond to cells in the previous diagonal, and processors in which a pair of symbols are present correspond to a cell in the diagonal being computed; in this latter case, the new value for such a processor l (corresponding to $d(i,j)$) is computed using the values stored in processors $l-1$ ($d(i,j-1)$) and $l+1$ ($d(i-1,j)$) as well as the value of processor l at the previous timestep ($d(i-1,j-1)$). A portion of the computation of such an array relative to the strings in Figure 3 is shown in Figure 5. In this figure, note how the middle portion of each row encodes a diagonal of the table, such that the circled values in this figure correspond to a version of the table in Figure 3 rotated 45^o clockwise. At the end of the computation, the final values pumped out of the sides of the array by both channels correspond to the edit distance between the two given sequences.

Many other systolic array architectures have been proposed for edit-distance and related sequence comparison problems in the years since 1985 (see [3], [10], [11] and references); however, given the simplicity of the scheme proposed by Lipton and Lopresti, we decided that this was the one best suited for our initial implementation efforts relative to FPGAs.

III. Prototyping a Parallel Algorithm Using FPGA

To implement the algorithm described in Section II, we used the FPGA system available to us, namely the Xilinx Rapid Prototyping System [9]. This system runs the Xilinx ISE (Version 6.1.03i) development [4] and ModelSim (Xilinx Edition II v5.7g) hardware simulation packages; together, these packages provide various methods for specifying circuits, creating hardware images, downloading such images to chips, and testing the resulting chips. This system has been paired with the Xilinx Virtex XCV2000E FPGA chip, which is connected to the system by a parallel-port interface.

Our prototyping experience can be broken into four stages: system set-up, hardware algorithm specification, hardware algorithm simulation, and hardware algorithm execution. Details of each stage, including the problems we encountered, are described in the following four sub-sections.

A. System Setup

As the Xilinx system and hardware were already in place in the MUN Computer Science VLSI Laboratory, the only major task we had in this stage was to ensure that all relevant software was installed. This was hampered by unexpected problems with software licenses. The Xilinx system at MUN operates under a license provided through a FlexLM server that is supposed to cover all versions of the ISE software and allow us to use all of the features (including ModelSim) once installed; however, this server did not in-

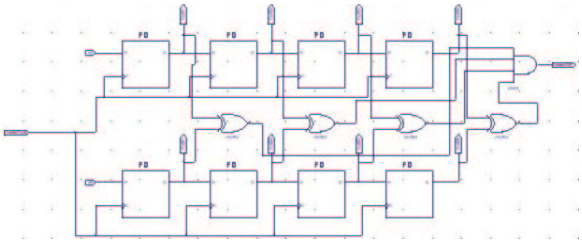

Fig. 6. 4-Bit Shift Register Comparator: Modular Design

```
Define ports:

  in_CLK;
  in_RESET;
  in_value1;
  in_value2;
  out_allEqual;
  vector r1(3 downto 0); // register 1
  vector r2(3 downto 0); // register 2

Define behavior:

  process (CLK, RESET)

    if RESET = 1 then
        r1 = "0000"; r2 = "0000";
    elseif CLK='1' and CLK'event then
        r1(3) <= r1(2); r1(2) <= r1(1);
        r1(1) <= r1(0); r1(0) <= in_value1;
        r2(3) <= r2(2); r2(2) <= r2(1);
        r2(1) <= r2(0); r2(0) <= in_value2;
        if (r1=r2) then
            allEqual = '1';
        endif
    endif
```

Fig. 7. 4-Bit Shift Register Comparator: VHDL

teract well with the Windows XP operating system under which we were running the Xilinx system – at first the software would not detect a license at all, then it would not get past compile-time due to license problems, and finally we realized that we could not get a full license to ModelSim. This entailed using an evaluation copy, which only worked under the administrator account on the computer, and hence required the presence of system administrators whenever the machine was reset. This resulted in an initial block delay of two weeks and then shorter delays throughout the summer.

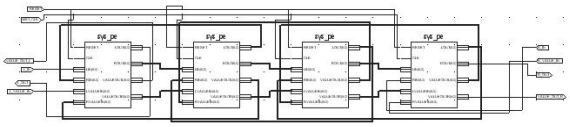

Fig. 8. Systolic Array (4-PE Configuration)

B. Hardware algorithm specification

The Xilinx system supports two distinct forms of hardware prototype specifications, modular component design and hardware description languages. Modular component design involves coding the simplest parts of the algorithm in gate-level logic and then building more and more complex components in a hierarchical manner from simpler ones until a component is derived that implements the whole algorithm. Hardware description languages are programming languages that allow you to code an algorithm at a high level which will then be translated by a compiler into a hardware description for whatever chip-system you are using.

To assess the relative advantages and disadvantages of each approach (as well as gain familiarity with the Xilinx system), we decided to implement several basic circuits under both approachs. One such circuit was a 4-bit shift-register comparator, which checks if the bits stored in two 4-bit shift registers are identical. The description of this circuit derived using modular design is given in Figure 6. This approach works very well in designing small systems and would seem ideal for larger systems such as systolic arrays composed of many copies of the same processing element (PE); however, in our experience, it was very hard to design even a simple PE such as that used in [5] starting from the gate level, let alone the PE underlying parallel hardware for more complex types of sequence comparison. Such PE are much more easily described using a hardware specification language. The language we used in our project was VHDL (VHSIC (Very High Scale Integrated Circuit) Hardware Description Language) ([1], [6]), which is very similar to many of the high-level programming languages in use today. The VHDL code for the comparator is given in Figure 7. In such a description, we just write down the various I/O ports and what we want to happen every time the internal clock ticks, and let the compiler take care of making an image that can be downloaded into the FPGA.

In the end, we adopted a hybrid approach: We used VHDL to implement our PE and then the ISE schematic tool to place and link these PE. A view of the final circuit is given in in Figure 8.

Fig. 9. Simulation Results: Waveform Graph

C. Hardware algorithm simulation

Once a compiled version of our VHDL was successfully saved to the system, the ModelSim package allowed us to view a waveform graph of all of the input and output markers from our schematic for each clock cycle (see Figure 9). While such graphs are very useful for debugging circuits that instantaneously compute functions for which each input has a well-defined output stated in binary, we soon realized that such graphs are very difficult to interpret in terms or circuits like our own in which inputs and outputs are expressed in terms of multi-symbol binary blocks that are sent and received over time. Our first first test version had 4 processor elements, each with 8 input/output lines which were all 5 and 6 digit binary strings. While its operation was easy to follow relative to diagrams such as that in Figure 5, the mental translation from the provided waveform graphs to such diagrams was much more difficult than originally planned and proved a major bottleneck in the project – indeed, we are still in this stage of the project.

D. Hardware algorithm execution

Though we were not able to create a working version of the Lipton and Lopresti algorithm for reasons sketched in the previous section, we were able to proceed to this stage with some of the simpler circuits we developed. Here, we encountered one final barrier. Once a circuit is entered onto an FPGA chip, drivers must be written to allow testing programs to send input to and receive output from the FPGA. These drivers would then be distributed along with copies of the FPGA board, and are thus an integral part of the system. Unfortunately, we were not be able to locate adequate documentation to write such drivers, even for simple circuits, and suspect the task of writing drivers for circuits with temporally-encoded inputs such as our own would be more daunting still.

IV. Conclusions and Future Work

Our major conclusion from this project is that though FPGA systems are very useful for prototyping hardware, many problems not described in the published literature await the new investigator. Though some of the problems described in this paper (in particular, those involving licenses and buggy operating systems) are easily solved by careful choices and dealings with vendors, others (such as the difficulty in visualizing and testing the operation of circuits such as that in [5] which incorporate temporally-encoded inputs and outputs) are much more difficult, and may possibly require the development of new suites of tools for FPGA prototyping systems. That being said, we are are still optimistic about the prospects for applying such systems to the development of special-purpose hardware for biological applications; in particular, once we have mastered implementing edit-distance computations relative to sequences, we look forward to implementing hardware for computing edit distances between more complex data structures such as trees [8].

References

[1] Ashenden, P.J. (1990) *The VHDL Cookbook*. Department of Computer Science, University of Adelaide.

[2] Gusfield, D. (1997) *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press.

[3] Hirschberg, J.D., Hughey, R., and Jarplus, K. (1996) "Kestrel: A Programmable Array for Sequence Analysis." In *ASAP'96*. IEEE Computer Society. 25–34.

[4] ISE Quickstart Tutorial. `ftp://ftp.xilinx.com/pub/documentation/ise6_tutorials/ise6tut.pdf`

[5] Lipton, L. and Lopresti, D. (1985) "A Systolic Array for Rapid String Comparison." In *Proceedings of the 1985 Chapel Hill Conference on VLSI*. Computer Science Press. 363–376.

[6] Perry, D.L. (1991) *VHDL*. McGraw-Hill.

[7] Quinton, P. and Robert, Y. (1991) *Systolic Algorithms & Architectures*. Prentice Hall.

[8] Shasha, D. and Zhang, K. (1989) "Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems." *SIAM Journal on Computing*, 18(6), 1245–1262.

[9] Xilinx. `http://www.xilinx.com/`

[10] Yamaguchi, Y. and Maruyama, T. (2002) "High Speed Homology Search with FPGAs." In *PSB 2002*. World Scientific Press. 271–282.

[11] Yu, C.W., Kwong, K.H., Lee, K.H., and Leong, P.H.W. (2003) "A Smith-Waterman Systolic Cell." In *FPL 2003*. Lecture Notes in Computer Science no. 2778. Springer-Verlag. 375–384.