Families of FPGA-Based Algorithms for Approximate String Matching*

Tom Van Court
Boston University, ECE Dept.
tvancour@bu.edu

Martin C. Herbordt
Boston University, ECE Dept.
herbordt@bu.edu

Abstract

Dynamic programming for approximate string matching is a large family of different algorithms, which vary significantly in purpose, complexity, and hardware utilization. Many implementations have been reported, but have typically been point solutions: highly specialized implementations that address only one or a few of the many possible options. We report a set of three component types that address different parts of the DP string matching problem. Multiple, interchangeable implementations are available for each component type. This allows each application to choose the feature set required, then make maximum use of the FPGA fabric according to that application's specific resource requirements. Synthesis estimates show a 4:1 improvement in time-space performance, depending on the options chosen for a specific matching task.

1. Introduction

Approximate matching (AM) between strings is important in many applications. In text databases, it allows searching on words that may be misspelled, that have variant spellings, or that are rendered into English in different ways. Bioinformatics applications use AM to find similarities between DNA (nucleotide) or protein (amino acid) sequences that have diverged through mutation or evolution. Hamming distance, the number of differing characters, is one way to measure differences between two strings, but does not tolerate insertions or deletions (*indels*). More generalized edit distances, with indels as well as character substitutions, are commonly handled using dynamic programming (DP) techniques.

Although hardware design for DP-based approximate string matching has been well-studied over the last 20 years [1],[2],[3],[4],[5],[6],[7],[8],[9], little is in general use. Especially, in the mid- to late-80s, special purpose hardware for genome analysis looked about to take off [3]. But there were two problems: the development of fast heuristic algorithms (BLAST being the best known) and the brittleness of the hardware solutions. The first of these is no longer an issue: although the various versions of BLAST remain the most widely sequence processing programs, MD-based algorithms are also firmly established in a complementary role. The problem of brittleness remains, however. The issue is as follows: MD-based AM is not a single algorithm, but rather a family of algorithms. As a result, *there is too great a gulf between what biologists actually do and what designers of application specific hardware have supplied*.

Actual DP AM usages vary widely in their input sets, scoring functions, recurrence relations, output of interest, and so on. Typical hardware realizations implement just one set of parameters and behavioral variations, often without stating which assumptions and variations

* This work was supported in part by the National Science Foundation through award 9702483 and the NIH through award RR020209-01; it was also facilitated by donations of software and equipment from Xilinx Corporation.

have been chosen. This does not meet the needs of the many potential users, it limits the applicability of the realization, and it locks out customizations that may be needed during the exploratory phase of a string application.

This paper presents a family of architectures that implement many different DP AM algorithms. The architecture defines three component types that address three major categories of distinctions between different algorithms. Any one realization of a DP AM accelerator consists of one compile-time choice of component definition in each of type, plus parameter settings where appropriate. Users of the string matching hardware get maximum freedom of choice in algorithms this way, without cost in clock rate or hardware allocation due to unused features or over-generalization..

Section 2 of this paper reviews DP AM hardware implementations from recent decades, and shows the general outline of the algorithm family. Section 3 decomposes the DP AM problem along three axes of behavior. It identifies component types that capture each of these categories of behavior, and shows how a DP string matching system is built in terms of the three abstract component types. Section 4 describes specific implementations of each component type. This section also addresses finer levels of parameterization for customizing the detailed behavior of each component type, and solutions to problems in implementing this family design using only standard VHDL. We conclude by reporting time and space performance for a small subset of the string matching systems that can be built from the proposed component libraries, showing a 4:1 space-time performance range for matching systems with different options. The main conclusions are that interchangeable component families are feasible and that they offer good opportunities for performance management.

2. Previous work

When the Needleman-Wunsch (NW) algorithm for DP AM was published in 1970 [10], it soon became the standard technique for AM in biological sequence matching. It also spawned many variations, including the Smith-Waterman (SW) technique for local alignment, “end space free” variants [11] for overhangs, and a theoretically unbounded number of gap-penalty strategies [12]. Because of its regular structure, limited data types, and simple computation, it has been a target for hardware acceleration at least since 1986 [1],[2],[3],[4],[5],[6],[7],[8],[9].

Each variation on DP AM answers a different biological question. It is puzzling, therefore, that so few reports on DP AM acceleration state just what was accelerated or its biological significance. Only one implementation [8] appears to address more than one matching task, and even that is limited to SW nucleotide comparisons with scoring constants limited to 0 or 1. At least eight different evolutionary models underlie scoring for DNA string comparison, phrased in terms of two, four, or more free parameters, not counting gap scoring models [13]. Amino acid scoring is at least as complex. This creates a gulf between accelerator design and the biologist’s control over what question is being answered. The combinatorics of the problem explain part of the gap: there are just too many useful variations. If a fully generalized accelerator could be designed, it would lose efficiency due to feature bloat. The second reason for the gap between is that biologists often have difficulty expressing their requirements in mathematical terms explicit enough for implementation [14]. No one implementation can address all AM problems efficiently, so a family of implementations is required.

3. DP string matching.

We follow the usual practice of considering DP AM as a rectangular grid of computation cells, with positions along each axis corresponding to character positions in the two strings. It is well known that the DP computation can proceed in a wave-front fashion, along a diagonal across that grid. Only the computation cells along that diagonal need to be represented in hardware. Each step re-allocates the computation hardware to the next diagonal in the grid.

There are three major ways in which DP string matching algorithms differ from each other. First and lowest-level is the component that defines the *character rule*. This embodies the type of each character in the string. It also defines the *substitution matrix* that rewards exact or near matches and penalizes mismatches between two characters. The second difference between algorithms is the *matching cell*, the component that implements one unit of the 2D recurrence relation by which whole strings are compared. Any matching cell can work with any string rule, since the recurrence relation depends on alignment score values and not on the type of the strings being matched. The highest level component is the *sequencer*, which controls the basic flow of string data and matching results through the system. The sequencer, in turn, works the same way irrespective of the matching cell used.

3.1 Character Rule components

A character rule implements the abstract data type representing the basic symbol in the strings being compared. One string, the *reference* string, has each of its characters stored in a character rule instance. The other string, the *test* string, flows systolically past the reference string for comparison. The data portion of this abstract type is the actual representation of the character. In bioinformatics applications, the most common data types are:

- Amino acids, twenty common 'characters' in a protein's one-dimensional structure,
- Nucleotides: A, C, G, and T (in DNA) or U (in RNA),
- Nucleotide wildcards, typically the IUPAC nucleotide ambiguity codes, and
- Codons, the nucleotide triplets that encode amino acids in the genome.

The character rule's *substitution matrix* is the scoring function that measures goodness of match between corresponding characters in the two strings. Different substitution matrices represent models of evolution, chemical function, statistical features, and evolutionary distance between the sequences. Matrices may be parameterized, as in the Kimura matrix for DNA where a parameter represents uneven AT/GC background probabilities [13].

3.2 Matching Cell components

The matching cell is the recurrence relation that defines the DP matching algorithm. Equation 1 shows the recurrence relation for the NW global alignment algorithm [12]. Here S_{ij} is the score for comparing test and reference strings up to character positions i and j, where position 0 is before the first character. The $\boldsymbol{s}(x,y)$ function is the substitution matrix value comparing character x in the test string to character y in the reference string. Penalty value S_{gap} represents the cost of skipping one character, for example when matching 'carts' to 'cats'.

First, note that this recurrence does not itself use the test and reference string data – it uses a function that uses them. That means that the matching cell definition has no knowledge of the character type or inner structure of the $\boldsymbol{s}(x,y)$ function; it needs to know only the range of scores returned by $\boldsymbol{s}$. Second, there is a separate instance of the character rule component for each matching cell. This allows many character comparisons to be evaluated in parallel. Third, the

S_{gap} values may be non-trivial functions of the gap length. Affine gap penalties are common, and have the form $S_{gap}=\{G_{init}$ *if length = 0, else* $G_{ch}\}$. The G_{init} term penalizes opening of a gap, and G_{ch} penalizes each increment of gap length. Finally, the $i=0$ and $j=0$ expressions vary according to scoring policies that skip the beginning of one or both strings. Similar rules, not shown, can also apply to the ends of the strings.

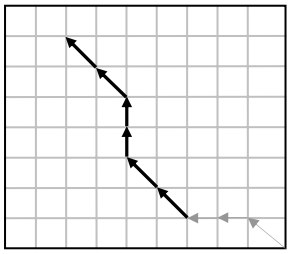
$$S_{i,j} = \begin{cases} \text{if } (i,j) = 0,0 & 0 \\ \text{else if } i = 0 & S_{0,j-1} - S_{gap} \\ \text{else if } j = 0 & S_{i-1,0} - S_{gap} \\ \text{else max} & \begin{cases} S_{i-1,j-1} + \mathbf{s}(x_i, y_i) \\ S_{i-1,j} - S_{gap} \\ S_{i,j-1} - S_{gap} \end{cases} \end{cases} \quad \text{Eq. 1}$$

The matching cell also generates *backtracking* state. Once the score for the best alignment has been found, *traceback* data determines the character relationships that led up to that score. For example, strings **abcde** and **abcabxde** might be aligned in two ways depending on scoring policy. Traceback state determines which alignment was best (capitalization shows matches):

```
ABCabxDE        or        abcABxDE
ABC---DE                  ---ABcDE
```

NW global (end to end) and SW local (best substring to substring) alignment have different recurrence relations. Local alignment uses a form of saturated arithmetic for scoring, where negative alignment scores are clamped to zero. The bigger difference is in the backtracking state needed for recovering the best alignment. SW matching may find substrings anywhere as the best local match. Traceback state must remember the path through current substring matching as in NW, but must also remember the globally best substring score and where it occurred. This is best implemented as a different matching cell altogether. Rules for backtracking must also be different because of the different results generated, so backtracking is logically part of the matching cell component.

NW traceback:
End to end alignment
of complete strings



SW traceback:
Locate best substring,
then align substrings

Figure 1: NW vs. SW backtracking rules

3.3 Sequencing component

The third component in DP string matching distinguishes between two major uses of matching: scoring and alignment. Scoring is a one-pass algorithm that just reports goodness of match scores, for example in phylogenetic applications [15]. Alignment performs that forward scoring pass, then a second backward pass to recover the exact character and gap positions that gave the best overall score.

The sequencing component directs the flow of data in each case. Clearly, the alignment sequencer is more complex than the scoring sequencer. The scoring sequencer can discard the traceback state and logic that generates it, but the alignment sequence must store the traceback information. When the forward pass is complete, the backtracking sequencer re-reads the stored traceback information in LIFO order.

The matching cell's definition does not depend on the type of character data being matched, as long as the matching cell can pass characters of arbitrary type to the character rule. Likewise, the sequencer can be defined independently of the matching cells that it coordinates. The data types of scores (saturating or not) and traceback state (for global or local alignment) are irrelevant to the sequencer. All that matters to the sequencer is that there are scoring and traceback data, and that the matching cell translates saved traceback state into an alignment.

3.4 String matching accelerator

A DP string matching accelerator is built from three independent component types: a string rule, a matching cell, and a sequencer, as shown in Figure 2. This independence comes from the fact that much of the data passed between them is *opaque* to the other components. A component that handles data opaquely may transfer or store the data, but can not perform any other operation on it. Unlike *transparent* data, opaque data has no accessible inner structure. Even the number of bits in the value may be unknown to the component that carries it, though the size may be known implicitly by the compilation tools.

Figure 2 is a simplified diagram of the traceback sequencer component. Other logic, not shown, handles the host interface, end-of-string logic, and other housekeeping functions. Note that this component is not a 'leaf' component; it is a control component that aggregates and coordinates inner, leaf components. The 'Recent Scores' registers and 'Traceback LIFO' RAM blocks store data values defined by the matching cell. Even though they are defined by the sequencer and inside it, the sequencer knows only the names of the opaque data types. The sequencer uses these storage elements to hold data that is specific to the matching cell, and that is only ever passed between matching cells. The numbered connections in Figure 2 are:

1. Traceback results (transparent). During the second pass, the matching cell interprets the stored traceback information as a path through the 2D DP array.
2. Test string characters (opaque), being streamed past the systolic matching array.
3. Reference string characters (opaque). This is used only for setting up the reference string. This signal does not necessarily carry the same data type as in the test string. For example, if the test string contains nucleotides then the reference string may hold IUPAC wildcards.
4. Comparison scores (transparent). This is a signed numeric value indicating goodness of match between a reference string symbol and a test string symbol.
5. Traceback state (opaque). During the forward pass, this records whether skipping or matching a character gave a better matching score. It may include other state: for example, the SW matching cell must first work back to the best substring match, then report on that substring alignment.
6. Scoring data (opaque). These values contain data needed for recording the best match, including scores of nearby characters, data for computing gap scores, etc. Different matching cells, implementing different policies defining 'best', require different data for computing the best score. This is a VHDL record that contains transparent and opaque data elements. The transparent data includes the numeric score representing the best match, i.e. the scalar result required by the host application.

Lines 1-3 send data to or accept data from the host. The scoring sequencer (not shown) is simpler than the traceback sequencer. It does not contain the Traceback RAM or line 1 for reporting the traceback path. Figure 2 shows that one instance each of the character rule and matching cell components, plus book-keeping data, form a single unit. The systolic matching array consists of a linear sequence of these blocks. The number of blocks will normally be the largest supported by available resources. The exact number depends on the resources required by each block, the resources claimed by the sequencer and overhead logic, and the capacity of the FPGA in which the array is implemented.

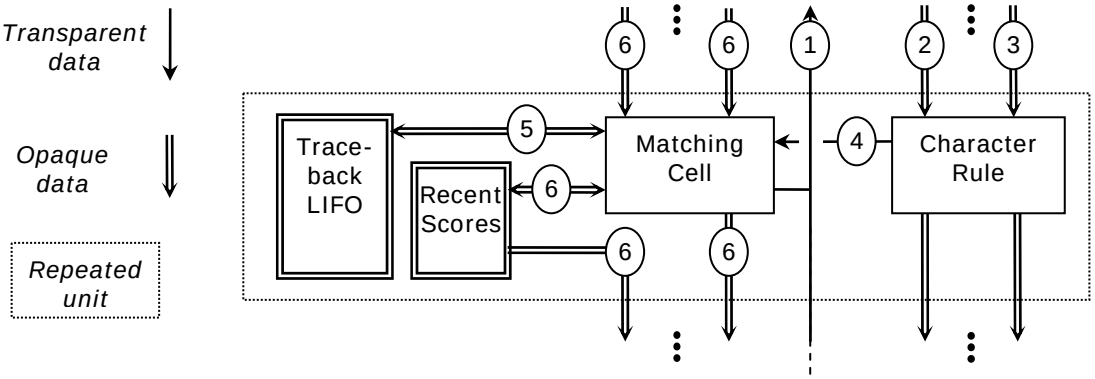


Figure 2: Traceback sequencer component structure

3.5 UML representation

We implement the DP AM application in VHDL, which is not an object oriented (*OO*) programming language. Still, some features of OO design can be represented with VHDL. The VHDL 'component' declaration, for example, defines the interface to an entity without specifying an implementation. That corresponds to the OO notion of an *abstract* class. Any one entity and architecture that implements the component interface corresponds to a *concrete* class. Structural VHDL is based on hierarchies of components containing other components, which corresponds to nested object composition. These mappings allow a UML class diagram to represent the logical structure of our application, as in Figure 3. Object names in that figure are descriptive only, and do not necessarily appear as programming symbols in the VHDL code. The number of matching cells is indeterminate, since it depends on resource availability in the FPGA and the resources claimed by each instance of each cell types chosen.

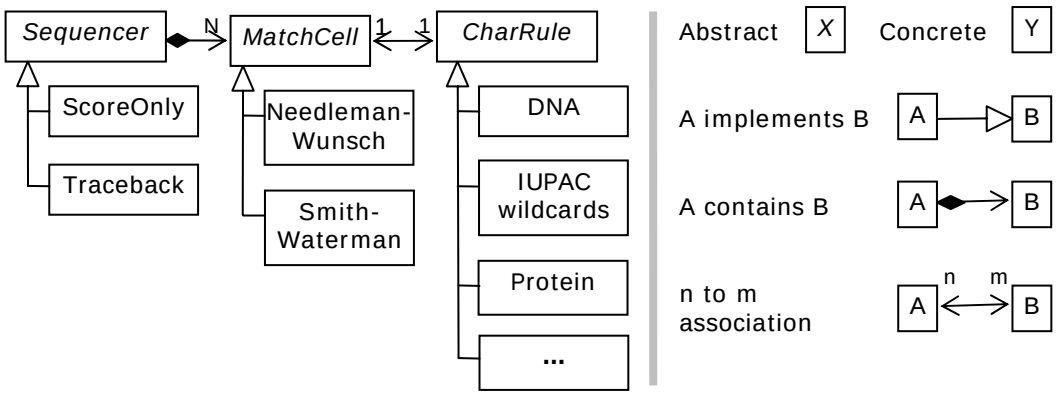


Figure 3: Logical structure of DP application family

Many more character rules exist than are shown. DNA strings may be aligned using Jukes-Cantor, Kimura, Tamura-Nei, or other rules [10]. Proteins may be aligned using BLOSUM, PAM, and other substitution matrices [16].

4. Component Implementation

The core of the DP AM logic consists mainly of the three component types described above. The challenge is to encapsulate the differences between implementations of each component type, so that switching one component type has no effect on other system components.

4.1 Component type selection

Careful use of VHDL allows one component definition to handle many disparate concrete implementations. For example, our matching cell component declaration includes:

Fragment 1

```
component match port(
    prev1, prev2:   in score1;
    prev12:         in score2;
    ...
    tbOut:          out traceback);
```

The `prev1`, `prev2`, and `prev12` values represent the $S_{i-1,j}$, $S_{i,j-1}$, and $S_{i-1,j-1}$ matching cell results. The `score1` type records the $S_{i,j-1}$ or $S_{i-1,j}$ score and `score2` is the $S_{i-1,j-1}$ score. The definitions of the `score1` and `score2` data types are not defined here, because they differ for NW and SW algorithms. NW matching uses declarations somewhat like the following:

Fragment 2

```
subtype score2 is
     integer range –MAXVAL to MAXVAL;
type score1 is record
     scoreVal: score2;
     gap1, gap2: boolean;
     end record score1;
subtype traceback is tbDir;
```

The `score1` type records more than just a matching score. For affine gap scoring, it also notes whether a gap has already been opened and in which string. Traceback data indicates whether the *i*, *j*, or *(i,j)* direction produced the best score. SW matching requires more state:

```
subtype alnScore is                          Fragment 3
      natural range 0 to MAXVAL;
type score2 is record
      scoreCur, scoreBest: alnScore;
      end record score2;
 type traceback is record
      toCur, toBest: tbDir;
      isBest: boolean;
      end record traceback;
```

The `score1` record is syntactically the same as before, even though the `score2` value within it has a different definition. SW alignment scores (unlike NW scores) are non-negative, as shown by `alnScore`. The `score2` record notes the alignment score for the substring being processed and also the globally best alignment score, as seen from the current point. SW traceback data notes the direction of this substring's best alignment score, whether the current position is the best score known so far, and the direction towards the best substring alignment previously known. The important fact here is that these NW and SW type definitions are interchangeable in the sequencer, where they are used as opaque types.

VHDL can not handle this change of type definition within the architecture/configuration or generic parameter model. One practice [17] would handle such differences by declaring the component port signals as `std_logic_vector` bitstring values. Scatter and gather logic in the entity body would break out or re-assemble fields within the bitstring signals. Pervasive use of bitstrings is effectively the same as using untyped data, however. It makes the intent of each signal impossible to determine without examination of all origins and uses of that value – a maintenance nightmare, reminiscent of abuses of PL/1's `unspec()` or C type casting. VHDL is a strongly typed language, and we prefer to use that feature of the language.

We change matching cells by replacing the pair of files that defines the cell. The first of those files is the matching cell's package definition, including the types shown. The second file contains the component body. The component definition in Fragment 1 is in a separate file and that is not replaced – it just uses the definitions in the replaced files. The same technique is used to select among sequencer and character rule implementations.

4.2 Component hierarchy

In common usage, the terms 'component' and 'leaf component' seem interchangeable. Traditional thinking holds that "*Reuse is in the first place a matter of reusing functionality, not structure*" [18]. Parameterization is defined in terms of "*… feature[s] that can be modified … without affecting the application's essential functionality*," where examples include buffer sizes or ROM dimensions [19].

In this application, the sequencers are reusable non-leaf components that define structure. They are reused by selecting the inner components they aggregate, which critically modify the functionality. Using components for structure and using behavior as a parameter is common in software design. This specific form of structure reuse demonstrates the *Strategy* design pattern [20], in which control flow and low-level behavioral elements are independently swappable. Compile-time selection of strategy objects is an admissible form of the design pattern, and is suited to hardware implementation. Other authors have also recognized the value of design patterns in hardware design [21][22][23], so one may look forward to support for these high-level design constructs in the future.

4.3 Component customization

VHDL compile-time customization is typically based on generic parameters. Generic parameter values may be selector values that choose between different component behaviors or may be numeric values. Character rule components use generic values to control the substitution matrices. Matrices usually map log-probabilities into some range of integer scores, using some parameterized function. Many models have additional parameters describing statistical or biological assumptions. The Jukes-Cantor model, for example, is defined in one parameter that lumps all evolutionary effects together [10]. The Tamura-Nei model has at least six parameters describing nucleotide and mutation probabilities.

Model-specific parameters become difficult to represent in standard VHDL. The component declaration defines a list of generic parameters. All of that component's architectures must use that one set of generics. If any architecture requires a new generic, that must be reflected in the component declaration and in all other architectures. It is clear, from the examples given, that the Jukes-Cantor and Tamura-Nei models require different lists of generic parameters. If other matrices were added in the future, they could require different numbers of generics with yet other meanings. That could mean that the requirements of each character rule affect the implementations of all other character rules.

The Dependency Inversion Principle [24] of design states that interfaces are the stable architectural elements and concrete implementations are subsidiary to the interface definitions. Changing the component interface for each new implementation would violate this principle. It would also violate the Open Closed Principle, that the system is open to new component implementations but closed to modification of known-good components. For now, we address this problem by using one string-valued generic parameter for the character rule component. That string encodes control values of any number and type. Each character rule implementation parses that one generic differently, using functions written in standard VHDL. This allows flexible lists of control values within an inflexible list of generic parameters.

We use the same scheme for parameterizing the matching cells. Our implementation of NW supports different policy options for comparisons where the end of one string overhangs the end of the other. These options do not apply to SW matching.

A better solution would use VHDL's facilities for checking generic parameter numbers, types, and values. Instead, this solution requires all checking to be done by the component architecture that parses the control string. This solution does, however, support any future parameter set using a fixed interface definition.

5. Conclusions

We have implemented the scoring sequencer, SW and NW matching cells, and a selected set of character rules. It would be infeasible to test all possible combinations: scoring vs. alignment, NW (with 15 end-rule variations) vs. SW, and eight character rule implementations that differ in more than parameter values. That yields 256 different implementations, not counting different substitution matrices (e.g. BLOSUM vs. PAM) or different numbers of bits in the scoring data paths.

These tests synthesize several different combinations of cell types, in order to evaluate chip utilization and basic clock rates allowed by each combination. For purposes of this study, the design blocks have not been tuned for maximum performance. Instead, we explore performance gains due to design flexibility. Assumptions about ranges of score values can vary the width of the score datapaths, but were held constant in these tests. The repeated unit consists of a

matching cell and an instance of the character rule component, so results are reported for the pair. The 'Cells' column in table 1 reports the number of these cells (assuming no overhead logic) that would fit into a Xilinx Virtex-II Pro XC2VP30 FPGA.

The IUPAC wildcard character rule allows the reference string to accept any of the 15 non-null subsets of nucleotides at each character position, so compares four-bit wildcard encodings to two-bit nucleotide encodings. RAM table character rules have substitution tables that can be reloaded in a running system. A 'fixed table' is symmetric substitution matrix implemented as ROM lookup or logic evaluation, according to compiler choice. It is possible that different values of table entries would have led to different resource utilization, but this has not been explored. 'Exact match' and 'IUPAC wildcard' character rules are implemented as logic functions, not lookup tables.

Table 1: Performance Estimates (Synthesized)

Matching Cell	Character Rule	String Type	Logic (slices)	Clock (ns)	Cells per XC2VP30	Speed GCUPS
NW	Exact match	DNA	109	12.9	125	9.68
NW	IUPAC wildcard	DNA	108	13.7	126	9.19
NW	Fixed table	DNA	111	14.6	123	8.42
NW	RAM table	DNA	108	16.8	126	7.50
SW	Exact match	DNA	190	13.3	72	5.41
SW	Fixed table	DNA	193	15.9	70	4.40
SW	Exact match	protein	205	13.0	66	5.07
SW	Fixed table	protein	239	25.5	57	2.23

Gate counts vary over a range at least 2:1, and cycle times vary nearly 2:1, according to the string matching task chosen. In other words, a hardware accelerator for the simplest kind of comparison gives a 4:1 advantage in space-time product over the most complex comparisons. A related performance measurement is billions of cell updates per second (GCUPS), the product of cell count and frequency.

These results can not easily be compared with other implementations. For example, [9] reports a hand-tuned and -placed system using another Xilinx FPGA. That reports 814 GCUPS, although the authors note that the number was never reached in practice because of slow system interface logic. That implementation was highly inflexible, however. Any change in the alphabet, substitution matrix, or gap penalty would have required new hand layout. Only fixed gap penalties could be supported, not affine gap scores. The system generated only scores, not full alignments. Also, although it reports a "Smith-Waterman" implementation, the logic cell appears to hold only enough state for a global (Needleman-Wunsch) alignment, with no possibility of local alignment or of different strategies for overhanging ends.

5.1 Future directions

There are large numbers of configuration options, such as NW end costs and score bit-widths that can also be varied; costs have not been established for all combinations. New character rules are possible, such as codons vs. amino acids. They raise new issues, such as the possibility of gap penalties that penalize codon frame shifts. These implementations all allow the reference strings to be reloaded in a running system. Comparisons would be simpler and faster, however, if the reference strings were hard-coded into the logic of the character rule cells as in other

systems [8],[9]. The current implementations are not highly tuned, so resource usage and clock rates may improve in the future. In the long run, this mechanism offers an unprecedented vehicle for exploring tradeoffs of hardware efficiency vs. application features.

Smith-Eggerton (SE) repeated matching [12] is an interesting variation, but is based on a calculation wavefront that lies vertically across the DP grid. These DP calculations are based on a wavefront running diagonally across the logical grid. SE could be probably accommodated with a different organization of the DP grid, but we have not investigated the changes that would be required. We have examined a modified SE algorithm with a diagonal wavefront, but have not fully characterized that algorithm's string-matching performance.

5.2 Summary

Hardware implementations of approximate string matching algorithms have typically ignored the variety of tasks to which DP matching is applied. We show that a family of hardware components, tuned for interoperability with each other, is a practical way to offer a wide variety of options. We have also shown that, by tailoring each component to a specific task, that the "generality penalty" can be avoided: each string matching application pays only to the cost of its own requirements, not the cost of other possible options. Considering both clock rate and number of computation units available, tailored matching cells offer a 4:1 range in time-space performance.

We also observed that several object-oriented design principles were very helpful in this implementation, including the Open-Closed principle, the Dependency Inversion principle, and use of the Strategy design pattern. These were directly applicable to standard VHDL and a standard development environment. This gives real cause for optimism about the transferability of modern software design techniques to large, complex hardware design, and suggests several ways in which minor tool changes could have significant effect on design productivity.

6. References

[1] Liptov, Richard and Daniel Lopresti. "Comparing Long Strings on a Short Systolic Array" in Systolic Arrays (Will Moore, Andrew McCabe, Roddy Uquhart, eds.). Adam Hilger 1986.

[2] Lopresti, Daniel P. "P-NAC: A Systolic Array for Comparing Nucleic Acid Sequences." Computer 20(7)98-99. IEEE 1987

[3] Roberts, Leslie. "New Chip May Speed Genome Analysis." Science 244:655-666, 12 May 1989.

[4] Chow, E. T. Hunkapiller and J. Peterson. "Biological Information Signal Processor." Proc. Application Specific Array Processors, 1991.

[5] Hoang, Dzung T. "Searching Genetic Databases on SPLASH 2." Proc. Workshop on FPGAs for Custom Computing Machines. 1993.

[6] Borah, Manjit, Raminder S. Bajwa, Sridhar Hannenhalli, and Mary Jane Irwin. "A SIMD Solution to the Sequence Comparison Problem on the MGAP." Proc. Application Specific Array Processors. 1994.

[7] Blüthgen, H.-M. and T. G. Noll. "A Programmable Processor for Approximate String Matching with High Throughput Rate" in Proc. Application Specific Systems, Architectures, and Processors. 2000.

[8] Guccione, Steven A. and Eric Keller. "Gene Matching Using JBits™." in Proc. 12th Field Programmable Logic and Applications. Springer, Berlin. 2002.

[9] Yu, C. W., K. H. Kwong, K. H. Lee, and P. H. W. Leong. "A Smith-Waterman Systolic Cell" in Proc. 13th Field-Programmable Logic and Applications. Springer, Berlin. 2003.

[10] Needleman, S. B. and C. D. Wunsch. "A General Method Applicable to the Search for Similarities in the Amino Acids Sequences of Two Proteins," Journal of Molecular Biology 48:443-453. 1970.

[11] Gusfield, Dan. Algorithms in Strings, Trees, and Sequences. Cambridge University Press. Cambridge UK. 1997.

[12] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press, Cambridge UK. 1998

[13] Nei, Masatoshi and Sudhir Kumar. Molecular Evolution and Phylogenetics. Oxford University Press, Oxford UK. 2000

[14] Bialek,. William and David. Botstein. "Introductory Science and Mathematics Education for 21st-Century Biologists." Science 303: 788-790. 2004.

[15] Felsenstein, Joseph. Inferring Phylogenies. Sinauer Associates, Inc., Sunderland MA. 2004.

[16] Mount, David W. Bioinformatics: Sequence and Genome Analysis. Cold Spring Harbor Laboratory Press, Cold Spring Harbor NY. 2001

[17] Xilinx, Inc. Synthesis and Verification Guide, ISE 6.2i. Xilinx Inc., San Jose CA. 2003.

[18] Schaumont, Patrick, Radim Cmar, Serge Vernalde, Marc Engles, and Ivo Bolsens. "Hardware Reuse at the Behavioral Level", Proceedings of DAC 99, 1999.

[19] Givargis, Tony D. and Frank Vahid. "Parameterized System Design." CODES 2000.

[20] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, Reading MA. 1994

[21] Åström, Pontus, Stefan Johnasson, and Peter Nilsson. "Application of Software Design Patterns to DSP library Design," Proc ISSS `01, ACM 2001.

[22] Damaševičius, Robertas, Giedrius Majauskas, and Vytautas Štuikys. "Application of Design Patterns for Hardware Design," Proc. DAC 03, ACM 2003.

[23] DeHon, A., J. Adams, M. DeLorimier, N. Kapre, and Y. Matsuda. "Design Patterns for Reconfigurable Computing," Proc. Field-Programmable Custom Computing Machines 2004.

[24] Martin, Robert C. Agile Software Development: Principles, Patterns, and Practices. Pearson Education, Upper Saddle River NJ. 2003