

Families of FPGA-Based Accelerators for Approximate String Matching¹

Tom Van Court²

Department of Electrical and
Computer Engineering
Boston University
tvancour@bu.edu

Martin C. Herbordt

Department of Electrical and
Computer Engineering
Boston University
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 reported impressive speed-ups, but have typically been point solutions – highly specialized and addressing only one or a few of the many possible options. The problem to be solved is creating a hardware description that implements a broad range of behavioral options without losing efficiency due to feature bloat. We report a set of three component types that address different parts of the approximate string matching problem. 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. Multiple, interchangeable implementations are available for each component type. We show that these methods allow the efficient generation of a large, if not complete, family of accelerators for this application. This flexibility was obtained while retaining high performance: We have evaluated a sample against serial reference codes and found speed-ups of from 180x to 500x over a high-end PC.

1. Introduction

Approximate matching (AM) between strings is essential to many important 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 [1-10], little is in general use. This is surprising, perhaps, given that as early as 1989, special purpose hardware for genome analysis appeared ready to become a mainstream technology [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

¹ 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 from Xilinx Corporation.

² Corresponding author.

first of these is no longer an issue: although the various versions of BLAST remain the most widely used sequence processing programs, DP-based algorithms have also become firmly established in a complementary role. The problem of brittleness remains, however. The issue is as follows: DP-based AM is not a single algorithm, but rather a family of algorithms. As a result, *there has been 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, and output of interest. Typical hardware realizations implement just one set of parameters and behavioral variations, often without stating which assumptions and variations 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 exploratory use of a string application. The fundamental problem, therefore, is not the implementation of a single high-performance solution, but rather of a family of solutions that span the application domain while retaining high performance, and only add incremental design cost.

This paper presents a family of structures that implement various DP AM algorithms. The architecture defines three component types that address three major distinctions between different algorithms. Any one realization of a DP AM accelerator consists of one compile-time choice of component definition in each type, plus parameter settings where appropriate. This way, users of the string matching hardware get maximum freedom of choice in algorithms without cost in clock rate or hardware allocation due to unused features or over-generalization. The solution's flexibility derives from the application of techniques common in software engineering (such as use of design patterns and data encapsulation) and an unconventional use of VHDL's strong typing. The primary contributions of our work are a family of DP AM string matching algorithms and a demonstration of the underlying design techniques.

The rest of this paper is organized as follows. Section 2 briefly reviews DP AM hardware implementations while Section 3 outlines the algorithm family. Section 4 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 5 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 describes solutions to problems in implementing this family design using only standard VHDL. We conclude by reporting time and space performance for a subset of the string matching systems that can be built from the component libraries, showing performance gains of from 180x to 500x over a high end PC.

2. Previous work

When the Needleman-Wunsch (NW) algorithm for DP AM was published in 1970 [13], 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 [14] for overhangs, and a theoretically unbounded number of gap-penalty strategies [15]. Because of its regular structure and limited data types, DP AM has been a target for hardware acceleration at least since 1986 [1-10].

Although each variation on DP AM answers a different biological question, reports on DP AM acceleration generally have not indicated the specific task being accelerated or its biological significance. Only one implementation [8] appears to address more than one matching task. Even that is limited to SW nucleotide comparisons with scoring constants limited to 0 or 1,

whereas at least eight different evolutionary models underlie scoring for DNA string comparison [16]. Amino acid scoring is no less 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 much 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. No one implementation can address all AM problems efficiently, so a family of implementations is required.

3. DP/AM Overview

The Needleman-Wunsch algorithm for aligning two strings is normally presented as a 2D array, like that shown in Figure 1. Each axis represents one of the strings to be aligned, and steps along each axis represent character positions within the string. The algorithm proceeds as if there were a cursor in each string. When both cursors step concurrently, that represents a match in one character position, whether or not the characters in that position are the same. If one string's cursor steps but the other cursor holds its position, that represents a character in the first string being skipped, i.e. a gap being opened in the comparison. Figure 1 illustrates the comparison of two hypothetical sequences GCGATCT and GCATTTA.

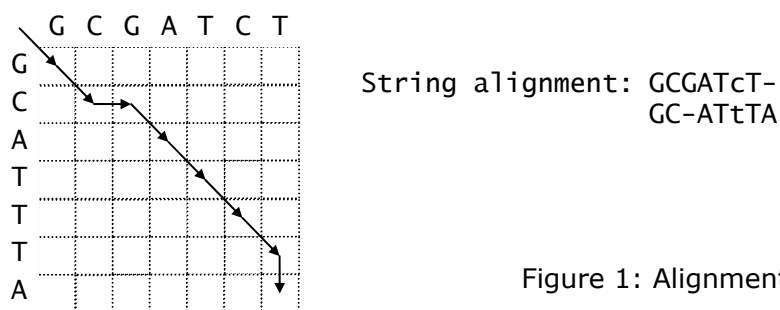


Figure 1: Alignment example

The aligned strings are written as GCGATcT- and GC-ATtTA, following the convention that a gap is written as a hyphen, and exact matches as uppercase. This alignment demonstrates three significant features. First is the indel at the third position in the alignment. The name *indel* comes from the assumption that these two strings descend from a common parent, which no longer exists. The indel could represent insertion of the letter G into the first string, or deletion of a letter G from the second. Lacking knowledge of the common ancestor, the insertion and deletion are equally likely. The second interesting feature is the mismatch at the sixth position of the alignment. Here, the mismatch is assumed a change of character, rather than insertion or deletion, because of the high-quality matches flanking the mismatch. Again, available information gives no reason to pick one or the other as being the 'right' value. The third interesting feature is the indel at the end of the strings. In English, it is relatively common to see a prefix or suffix on a word, leaving the root word readily recognizable. Likewise, genetic events that cut a string short or append new letters are also relatively common, and may have less biological importance than other kinds of differences. Although this 'end gap' is written the same way as an interior gap, it may not have the same biological significance.

The alignment is drawn as one path through the 2D array of possibilities. Finding the path is an iterative process that scores all cells of the array and determines the highest-scoring path through the array. Comparison starts as if the cursors in the two strings were set to position 0,

the position just before the first character in each string. The score S_{ij} for grid cell (i, j) is computed using the following recurrence relation [15]:

Equation 1: Needleman-Wunsch recurrence relation

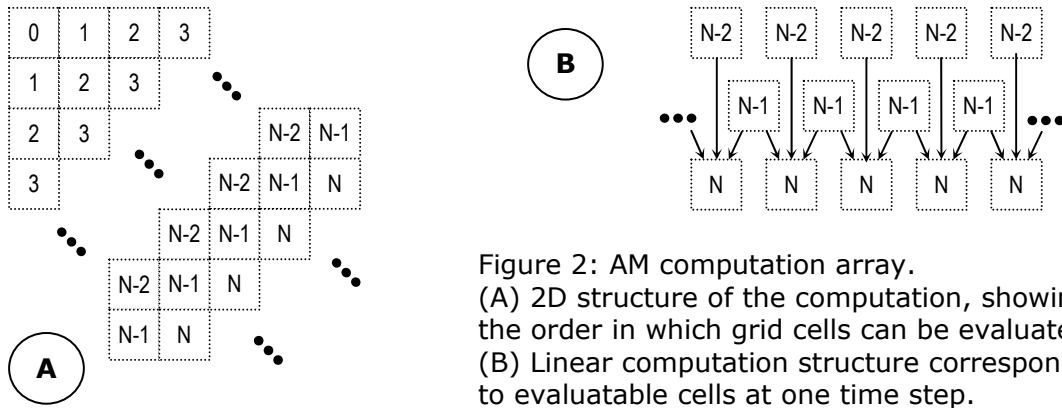
$$S_{i,j} = \begin{cases} \text{if } (i,j) = 0,0 & 0 & (1) \\ \text{else if } i = 0 & S_{0,j-1} - S_{gap} & (2) \\ \text{else if } j = 0 & S_{i-1,0} - S_{gap} & (3) \\ \text{else max} & \left\{ \begin{array}{l} S_{i-1,j-1} + s(x_i, y_j) \\ S_{i-1,j} - S_{gap} \\ S_{i,j-1} - S_{gap} \end{array} \right\} & \begin{array}{l} (4) \\ (5) \\ (6) \end{array} \end{cases}$$

Line (1) is the base step of the recurrence. It happens before any comparison and initializes the score. Lines (2) and (3) represent the left end-gap, where one string may have a prefix that the other lacks, i.e. where one string's cursor is inside the string but the first character of the other string is not yet involved in the alignment. Lines (4-6) represent the interior of the array, where the decision is made to extend the alignment by one position along both strings (4), or to assume a gap in one string or the other (5 or 6). The comparison function $s(x_i, y_j)$ determines goodness of match between two characters, one from each string. Depending on the problem at hand, the function (or *substitution matrix*) is chosen to emphasize small differences between close relatives, find small similarities between distant relatives, compare chemical or structural similarity of amino acids, accommodate background probabilities of string symbols, or represent other biological and mathematical assumptions.

The S_{gap} value represents the penalty for skipping part of a string in performing the alignment. In fact, the S_{gap} value is often an affine function of the form $S_{gap} = S_{open} + S_{cont} * len$, for gap lengths $len \geq 1$, and $S_{open} > S_{cont}$. That reflects the intuition that cutting the string at all (opening the gap) is more significant than the number of characters that continue the gap.

Different gap costs – possibly zero – may be applied at one or both ends of the string, though this possibility is not shown in Equation 1. The end gap penalty is typically much lower than the interior gap penalty, representing the idea that a prefix or suffix changes the string's root meaning less than an interior misspelling.

The score at the lower right corner, S_{IJ} , represents the end-to-end goodness of match between the two strings. When asking the question, “Is string A more similar to B or to C ?”, the result depends only that score for the A/B alignment and the A/C alignment. Other times, however, the experimenter is interested in seeing which parts of the two strings are similar. In that case, a second pass is made over the computation array, starting with that final score S_{IJ} . According to lines (4-6) of Equation 1, that score could have been derived from either $S_{I-1,J-1}$, $S_{I-1,J}$, or $S_{I,J-1}$. The traceback step determines which of those scores is highest, i.e. which represents the best partial alignment up to that point. Traceback continues, following the highest preceding score, back to the upper-left corner where computation began. That path, consisting of upwards, leftwards, and upper-left diagonal steps, represents the sequence of cursor positions along the two strings that led to the optimal alignment.



4. DP string matching

4.1 Basic Implementation

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, the reference string R and the query string Q . Each (i,j) position in the array represents alignment of character r_i with character q_j , so the array as a whole compares every reference string character to every character in the query string. Because of the dependencies in the DP recurrence relation of Equation 1, computation can proceed in a wave-front fashion along a diagonal across that grid Figure 2A. Only the computation cells on that diagonal require hardware, at any one time. Figure 2B shows those computation cells, along with storage for the input values on which those computations depend.

Each time-step in the computation array computes all elements of a diagonal. The scoring results computed for diagonal N at one time step become inputs from diagonal $N-1$ at the next step and input $N-2$ at the time step after. Computation cells, labeled N in Figure 2B, each hold two characters, one from each of the strings. At each time step, the query string shifts across the array by one character position. At the end of the computation, every query string character has been co-resident with every test string character in some computation cell. That means that the computation corresponding to each cell in the 2D array has been represented, at one time or another, in the linear computation array.

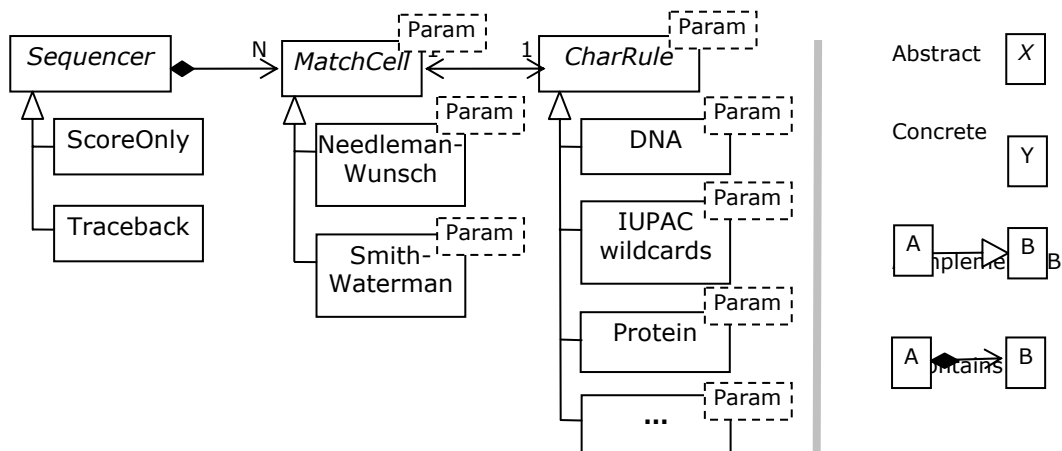


Figure 3: Logical structure of DP AM application family

Figure 3 illustrates the 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.

Although Figure 3 uses object oriented (OO) notation, we implement the DP AM application in VHDL which is not an OO language. Still, some features of OO design match well to VHDL. The VHDL ‘component’ declaration, for example, defines the interface to an entity, its IOs and their types, without specifying an implementation. That corresponds to the OO notion of an *abstract* class. An architecture that implements the component interface corresponds to a *concrete* class. UML class parameterization corresponds closely to VHDL generics. Structural VHDL is based on hierarchies of components containing other components, which corresponds to nested object composition. 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.

4.2 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. 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.

Characters in the two strings need not be of the same type. For example, the reference and query strings may be wildcards and literals, or amino acids and codons. An additional constraint is not shown in Figure 3. In any one system, all character rule instances must be of the same type (or VHDL architecture).

The substitution matrix is also part of the character rule. It is the scoring function that measures goodness of match between corresponding characters in the two strings. Despite its name, it may be implemented as a logical function instead of an actual matrix lookup table. Different substitution matrices represent different models of evolution, chemical function, statistical features, and evolutionary distance between the sequences. Some matrices are defined in terms of parameters, for example the Kimura matrix for DNA with a parameter representing uneven AT/GC background probabilities [16].

Many more character rules exist than are shown in Figure 3. DNA strings may be aligned using Jukes-Cantor, Kimura, Tamura-Nei, or other rules [13]. Proteins may be aligned using BLOSUM, PAM, and other substitution matrices [17].

4.3 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. 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 $s(r,q)$ function; it needs to know only the range of scores returned by s . The S_{gap} values in lines (2, 3, 5, 6) may be non-trivial functions of the gap length. Affine gap penalties are common, and have the form $S_{gap} = \{S_{open} \text{ if length} = 0, \text{ else } S_{cont}\}$. The S_{open} term penalizes opening of a gap, and S_{cont} 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. Indels at the beginning or end of a sequence may have different significance than interior gaps, so may be scored differently.

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 and SW alignment have different recurrence relations. Local alignment uses saturated arithmetic for scoring, where negative alignment scores become zero. Local alignment also has fixed rules regarding end gaps, where global alignment allows several different choices of end gap treatment. 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 remembers the path through current substring match as in NW, but must also remember the globally best substring score and where it occurred. We implement this 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.

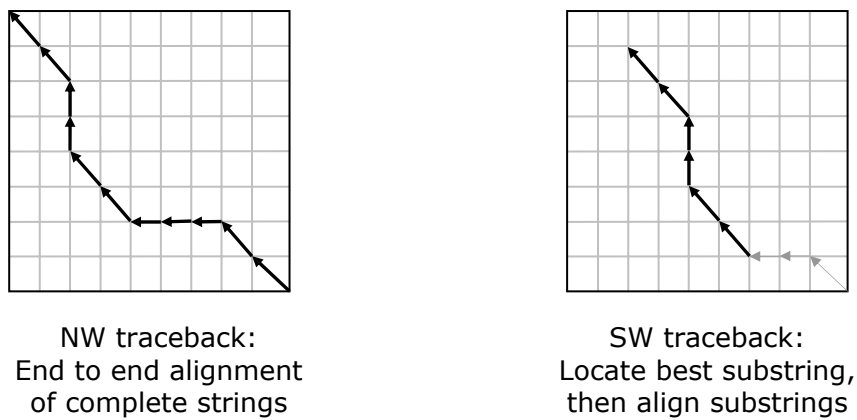


Figure 4: NW vs. SW backtracking rules

As with the character rule, Figure 3 omits the requirement that, in any one system implementation, all instances of the matching cell must be of the same type: NW or SW.

4.4 Sequencing component

The third component distinguishes between two major uses of matching: scoring and alignment. Scoring is a one-pass algorithm that just reports goodness of match values, for example in phylogenetic applications [11]. Alignment performs that forward scoring pass, then a backward pass to recover the exact character and gap positions that gave the best 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 sequencer 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.

4.5 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 5. 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.

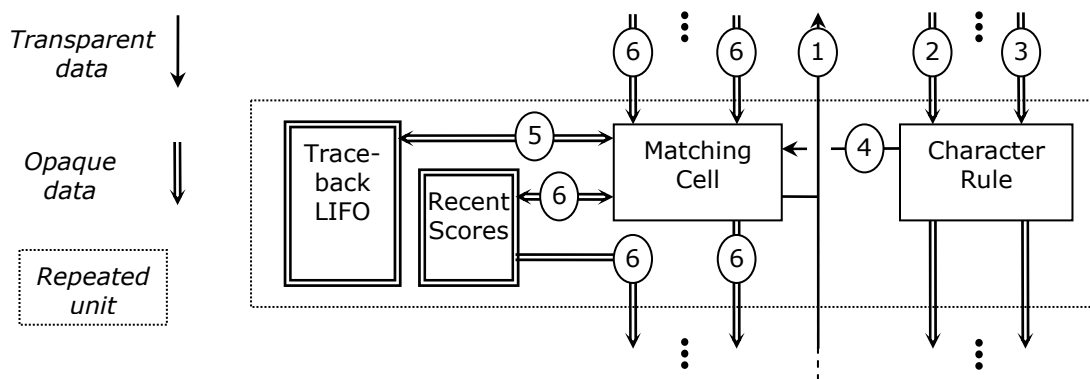


Figure 5: Traceback sequencer component structure

Figure 5 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 5 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 pathway is used only for loading the reference string.
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 5 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.

5. 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.

5.1 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:

```

component match      port(                               Fragment 1
  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:

```

subtype score2 is                                         Fragment 2
  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 [18] 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's type casting. VHDL is a strongly typed language, and we prefer not to defeat 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.

5.2 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*" [19]. 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 [20].

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 [21], 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 [20-22], so one may look forward to support for these high-level design constructs in the future.

5.3 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 [13]. The Tamura-Nei model has several different parameters describing nucleotide and mutation probabilities.

Each different implementation of the character rule requires a different number of generic parameters, with different data types and meanings, for proper parameterization. This is difficult to represent using standard VHDL, however. The most natural VHDL representation would use one 'component' declaration for the arbitrary character rule, and a different architecture for each specific rule. VHDL language rules, however, require that the component declaration and all architecture declarations have exactly the same set of generic parameters.

This leads to maintenance problems. An architecture that implements a character rule with fewer generic parameters must define all the generic parameters needed by all other character rules, in order to match the shared component declaration. If a new character rule (and corresponding architecture) require a new generic parameter, then the component declaration

must be modified – as well as the entity declarations for all other architectures of the character rule, in order to be syntactically compatible with their shared component declaration.

The Dependency Inversion Principle [25] 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 string-handling 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 different solution would use VHDL’s facilities for checking generic parameter numbers, types, and values. Instead, this solution here requires all checking to be done by the component architecture that parses the control string. This solution is necessary, however, to support any future parameter set within a fixed interface definition.

6. Results

We have implemented the scoring sequencer, SW and NW matching cells, and eight character rules. Together, these allow the generation of 256 different AM accelerators. This does not include the additional capabilities of varying the substitution matrices (e.g. BLOSUM vs. PAM) and the numbers of bits in the scoring data paths. We now discuss performance with respect to four issues: performance, programmability, performance plus flexibility, and generality.

Matching Cell	Character Rule	String Type	Logic (slices)	Clock (ns)	Cells per XC2VP70	Speed GCUPS	Speedup
NW	3GHz Xeon PC implementation (all NW models)					0.046	---
NW	Exact match	DNA	109	12.9	303	23.48	510
NW	IUPAC wildcard	DNA	108	13.7	306	22.33	485
NW	Fixed table	DNA	111	14.6	298	20.41	443
NW	RAM table	DNA	108	16.8	306	18.21	395
SW	3GHz Xeon PC implementation (all SW models)					0.029	---
SW	Exact match	DNA	190	13.3	174	13.08	451
SW	Fixed table	DNA	193	15.9	171	10.75	370
SW	Exact match	protein	205	13.0	161	12.38	426
SW	Fixed table	protein	239	25.5	138	5.41	186

1. Performance. We implemented a sample of these accelerators on a Xilinx XC2VP70 -5 FPGA and evaluated them with respect to chip utilization and basic clock rates. These results, as well as comparisons with a 2004-era PC, are shown in Table 1. The goal of this study being

to explore performance gains due to design flexibility, the design blocks have not been tuned for maximum performance. Still, we observed speed-ups of from 186x to 510x.

In general, ranges of alignment 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 XC2VP70 FPGA.

The character rules are now described. 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 a symmetric substitution matrix implemented as ROM lookup or logic evaluation, according to compiler choice. ‘Exact match’ and ‘IUPAC wildcard’ character rules are implemented as logic functions, not lookup tables.

2. Programmability. Perhaps the most critical issue is how much logic designer time is required to create an FPGA accelerator for a complex logic family. These applications were created in less than six months by a graduate student with modest logic design experience. This time includes developing tools and infrastructure. However, the most important metric is not design-hours; rather, it is the number of design-hours per accelerator use. In this context, there are two benefits of the approach described here over standard HDL-based logic design. The first is that dozens to hundreds of accelerators can now be generated *and optimized to the capacity of the FPGA* with no further intervention by the logic designer. These accelerators can, of course, be generated by any number of independent end users, and as their own experiments require. The second benefit occurs if logic designer intervention *is* again needed for a unique new feature. Since the structures are already in place, little additional time is required beyond the design of the particular component.

3. Performance plus flexibility. The speed-ups are satisfying given the logic design effort and the importance of the applications. However, for any one application instance, a hand-crafted circuit-level solution would certainly yield even better performance. Perhaps our key result is that this is of little consequence: time and again, high-performance point solutions have been introduced, but found to be too brittle for production use. In contrast, we achieve speed-ups of two orders of magnitude over entire ranges of family members.

4. Generality. This addresses the question: for an application family, should a single accelerator be built that does everything, i.e., that supports all of the applications within the family, or should accelerators be generated and optimized independently? Examining Table 1, we observe a range greater than 2:1 in the number of processing elements per FPGA, and a range near 2:1 in clock speed. Suppose, for the moment, that one PE could handle all of the string-matching tasks addressed in Table 1. In that case, the “exact match/DNA” case would be forced to run more than 4× slower than necessary. Instead, we show performance for each variation on the theme in terms of an accelerator specific to that variation. Simpler computations do incur the cost of circuitry needed for more complex comparisons. At the same time, complicated operations are not constrained to the PE cell area and datapath width of the simplest operation.

7. Conclusion

7.1 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 also show that, by tailoring each component to a specific task, the “generality penalty” can be avoided: each application pays only the cost of its own requirements, not the cost of other possible options.

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.

7.2 Future Work

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 [15] 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.

8. 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. 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] Dydel, Stefan and Piotr Bała. "Large Scale Protein Sequence Alignment using FPGA Reconfigurable Logic Devices." Proc. FPL 2004
- [11] Felsenstein, Joseph. Inferring Phylogenies. Sinauer Associates, Inc., Sunderland MA. 2004.
- [12] Page, Roderic D. M. (ed). Tangled Trees. The University of Chicago Press. Chicago IL. 2003.
- [13] 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.
- [14] Gusfield, Dan. Algorithms in Strings, Trees, and Sequences. Cambridge University Press. Cambridge UK. 1997.
- [15] 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
- [16] Nei, Masatoshi and Sudhir Kumar. Molecular Evolution and Phylogenetics. Oxford University Press, Oxford UK. 2000
- [17] Mount, David W. Bioinformatics: Sequence and Genome Analysis. Cold Spring Harbor Laboratory Press, Cold Spring Harbor NY. 2001
- [18] Xilinx, Inc. Synthesis and Verification Guide, ISE 6.2i. Xilinx Inc., San Jose CA. 2003.
- [19] Schaumont, Patrick, Radim Cmar, Serge Vernalde, Marc Engles, and Ivo Bolsens. "Hardware Reuse at the Behavioral Level", Proceedings of DAC 99, 1999.
- [20] Givargis, Tony D. and Frank Vahid. "Parameterized System Design." CODES 2000.
- [21] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, Reading MA. 1994
- [22] Åström, Pontus, Stefan Johnsson, and Peter Nilsson. "Application of Software Design Patterns to DSP library Design," Proc ISSS '01, ACM 2001.
- [23] Damaševičius, Robertas, Giedrius Majauskas, and Vytautas Štuikys. "Application of Design Patterns for Hardware Design," Proc. DAC 03, ACM 2003.
- [24] DeHon, A., J. Adams, M. DeLorimier, N. Kapre, and Y. Matsuda. "Design Patterns for Reconfigurable Computing," Proc. Field-Programmable Custom Computing Machines 2004.
- [25] Martin, Robert C. Agile Software Development: Principles, Patterns, and Practices. Pearson Education, Upper Saddle River NJ. 2003