

An FPGA-based Search Engine for Unstructured Database

**Benjamin West, Roger D. Chamberlain,
Ronald S. Indeck, and Qiong Zhang**

Benjamin West, Roger D. Chamberlain, Ronald S. Indeck, and Qiong Zhang, "An FPGA-based Search Engine for Unstructured Database," in *Proc. of 2nd Workshop on Application Specific Processors*, December 2003.

School of Engineering and Applied Science
Washington University
Campus Box 1115
One Brookings Dr.
St. Louis, MO 63130-4899

An FPGA-based Search Engine for Unstructured Database

Benjamin West
Department of Computer
Science and Engineering
Washington University
1 Brookings Drive
St. Louis, MO USA 63130
bmw3@ccrc.wustl.edu

Roger D. Chamberlain
Department of Computer
Science and Engineering
Washington University
1 Brookings Drive
St. Louis, MO USA 63130
roger@wustl.edu

Ronald S. Indeck
Department of Electrical
Engineering
Washington University
1 Brookings Drive
St. Louis, MO USA 63130
rsi@ee.wustl.edu

ABSTRACT

We present an FPGA-based search engine that implements the Smith-Waterman local sequence alignment algorithm to search a stream of input data for patterns up to 38 bytes long. This engine supports the sequence gap, match, and mismatch weightings necessary for genome sequence alignment, and it can perform inexact string matching on generic text. As the engine simply processes input data streaming through it, it can readily operate on data that are unindexed and unsorted, i.e., on unstructured databases. Furthermore, the engine can sustain a search throughput of 100 MB/sec¹, making it capable of processing input data at the maximum sustained throughput of the hard drive (or array of drives) storing the data.

For a performance demonstration we compare the throughput of the engine embedded in a prototype system with the execution times of a direct software implementation of the search engine's kernel. Although the prototype system limits the input data to 40.5MB/sec, we show how parallelism and pipelining allow this FPGA-based search engine to sustain a search throughput of 100 MB/sec, given a fast enough method for delivering input data, and thereby yield a performance gain over the software implementation of two orders of magnitude.

Categories and Subject Descriptors

B.6.1 [Hardware]: Logic Design; I.5.4 [Computing Methodologies]: Pattern Recognition—*Applications*; J.3 [Computer Applications]: Life and Medical Sciences—*Biology and Genetics*

General Terms

Design, Experimentation, Performance

¹GB = 10⁹ bytes, MB = 10⁶ bytes, KB = 10³ bytes, Gb = 10⁹ bits, Mb = 10⁶ bits, and Kb = 10³ bits.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

Keywords

FPGA, Smith-Waterman, biosequence analysis, string matching, unstructured database

1. INTRODUCTION

Researchers have sought to take advantage of recent improvements in hard drive storage capacity with large database endeavors such as the Human Genome Project. Here, terabyte and even exabyte-sized databases are searched repeatedly for approximate matches of unindexed, unstructured data [6]. Approximate string matching within unindexed, unstructured text data is also an important problem faced by the Intelligence community, which must sift through an enormous collection of information that is frequently in languages other than English, and in alphabets other than Roman.

However, shortcomings in the typical I/O path between hard drives and their host processors, illustrated conceptually in the top half of Figure 1, dramatically limit the ability to thoroughly search such databases in a reasonable amount of time. A clear solution to this I/O bottleneck is to bring the processing element closer to the storage medium, i.e., as close as possible to the data when they stream off the hard drive's magnetic read head [1]. Our prototype sys-

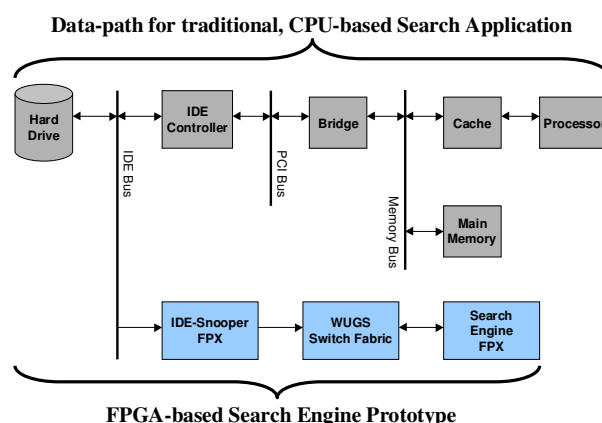


Figure 1: Traditional Hard Drive/Host Processor I/O Path, with Search Engine in Prototype System

tem containing the search engine, which is shown in the bottom half of Figure 1, attaches to the ATAPI/IDE pe-

ripheral bus that connects the hard drive to the bus controller in the host workstation. The prototype system is comprised of two FPGA-based boards and a Gb/sec interconnection network. The interconnection network is the experimental Washington University Gigabit Switch (WUGS) [8], and the FPGA board is the Field Programmable Port Extender (FPX) [5]. Both are components of a research platform for hardware-based networking applications, but from a more general perspective, they provide a platform where million-gate-capacity FPGAs may be interconnected with high-speed data-paths. The left-hand FPX shown in Figure 1 is programmed to capture data traversing the AT-API/IDE bus and forward them via the interconnection network to the right-hand FPX, which is programmed with the search engine itself. Using this prototype system, our search engine can sustain an overall throughput of up 40.5 MB/sec, although the sustained throughput through the FPGAs is 100MB/sec, and it can accept a search query string up to 38 bytes long.

The search engine is a parallelized, pipelined implementation of the Smith-Waterman local sequence alignment algorithm [7], an algorithm traditionally intended for bio-sequence analysis, e.g., genome or protein sequence alignment. However, our search engine operates on data at the byte level, rather than on data that is minimally bit-encoded for a bio-sequence symbol set, and so the engine can perform the Smith-Waterman alignment on generic data, e.g., ASCII text, to effect inexact string matching. The decision was made to implement the Smith-Waterman algorithm so it could handle both generic text-string matching and bio-sequence analysis, so that search engine could find use in a broader range of potential applications.

This search engine is designed to be applicable to the project Mercury system currently under development at Washington University. In this project we are exploring the benefits of using reconfigurable hardware inside the hard drive to perform searches of massive databases at or near the speed of raw data coming of the drive's read head [1].

Section 2 highlights recent work in the field of hardware-based searching. Section 3 provides an overview of the Smith-Waterman local sequence alignment algorithm. Section 4 details the design and internal components of the search engine and the prototype system containing it. Section 5 provides a comparison of the throughput of the search engine with that of its twin software implementation. Section 6 provides a summary of this paper.

2. RELATED WORK

The wealth of recent work in hardware-based searching demonstrates the magnitude of search throughput possible with even non-application specific hardware such as FPGAs. [3], for example, describes an implementation of the Smith-Waterman local alignment algorithm, very similar to the one in this paper, albeit with a systolic array of custom 12-bit VLSI processors, and with fixed gap and mis-match penalties. [2] is an FPGA-based implementation also similar to the one in this paper, which uses runtime re-configuration of the FPGA to specify gap and mis-match penalties. Finally, [4] uses the same FPX device to perform Regular Expression pattern matching on data passing through an Internet firewall, an implementation similar in motivation and spirit to this, but with the target data originating directly from the Internet rather than from a storage device.

3. SMITH-WATERMAN LOCAL SEQUENCE ALIGNMENT

The classic Smith-Waterman local alignment algorithm [7] is a dynamic programming method used in bio-sequence analysis which finds the best possible alignment between two strings of characters, the pattern string p and the target string t . The score of the alignment is judged by the gaps and mismatched characters that must be tolerated to make the alignment. The pattern string p is understood to be the search query argument, and the target string t the database upon which the search is performed. As this is local alignment, the database t can be magnitudes longer than p . The range of characters found in both strings depends on the nature of alignment that is sought; for genome alignments, for example, the range would be the four DNA bases "ATCG" (along with whatever wild-card characters are allowed), and for proteomic alignments, the range would be the approximately 20 amino acids (and wild-card characters). Indeed, this alignment method can also find application with regular text searching, where the range of characters could be those in the 8-bit ASCII character table or any other arbitrary character set. Furthermore, given the ability of the Smith-Waterman algorithm to find alignments that include gaps or mismatched characters, it readily lends itself to performing inexact searches on text, e.g., searches that account for variations in spelling or transliterations of the sought pattern string p . The Smith-Waterman local alignment al-

	t1	t2	t3	t4	t5	...	tj	...	tn
p1	d(1,1)	d(1,2)	d(1,3)	d(1,4)	d(1,5)	...	d(1,j)	...	d(1,n)
p2	d(2,1)	d(2,2)	d(2,3)	d(2,4)	d(2,5)	...	d(2,j)	...	d(2,n)
p3	d(3,1)	d(3,2)	d(3,3)	d(3,4)	d(3,5)	...	d(3,j)	...	d(3,n)
p4	d(4,1)	d(4,2)	d(4,3)	d(4,4)	d(4,5)	...	d(4,j)	...	d(4,n)
p5	d(5,1)	d(5,2)	d(5,3)	d(5,4)	d(5,5)	...	d(5,j)	...	d(5,n)
⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋱	⋮
pi	d(i,1)	d(i,2)	d(i,3)	d(i,4)	d(i,5)	...	d(i,j)	...	d(i,n)
⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋱	⋮
pm	d(m,1)	d(m,2)	d(m,3)	d(m,4)	d(m,5)	...	d(m,j)	...	d(m,n)

Figure 2: Dynamic Programming (DP) Matrix

gorithm finds the best possible alignment between p and t by arranging both strings along an axis of a 2-dimensional matrix of size $m \times n$, shown in Figure 2, where m is the length of the pattern string p and n the length of the target string t . Each element $d(i, j)$ in this matrix (called the Dynamic Programming (DP) matrix) represents the score for the i th character of p aligned with the j th character of t . This score is determined by the base cases and recursion shown in Equations 1 through 4.

$$d(1, 1) = B(1, 1) \quad (1)$$

$$d(i, 1) = \max[A; d(i-1, 1) + A; B(i, 1)] \quad (2)$$

$$d(1, j) = \max[d(1, j-1) + A; A; B(1, j)] \quad (3)$$

$$d(i, j) = \max[d(i, j-1) + A; d(i-1, j) + A; d(i-1, j-1) + B(i, j)] \quad (4)$$

A in Equations 1 through 4 is a user-defined constant representing the single-character gap penalty (usually negative), and $B(i, j)$ the scoring function dependent on the characters p_i and t_j . In the implementation used for this prototype, the scoring function $B(i, j)$ simply returns a user-defined constant (usually positive) when the characters p_i and t_j match, and another constant (usually negative), when the characters don't match. Thus, these two constants, called B_{match} and $B_{nomatch}$, respectively, represent the single-character match score and mismatch penalty that are used in calculating the overall alignment scores for p and t .

It should be noted the recurrence defined in Equations 1 through 4 is not local with respect to both the target and pattern strings. Rather, the alignment is local with respect to the target string, but global with respect to the pattern. That is, the alignment scores computed by this recursion represent an alignment of the pattern with the entire target string, or put simply, the points in the target string where the pattern makes an acceptable match. By comparison, a recursion that is local for both the pattern and the target strings could compute scores representing an alignment of a subset of pattern with a subset of the target string. An alignment occurs in the DP matrix when a particular

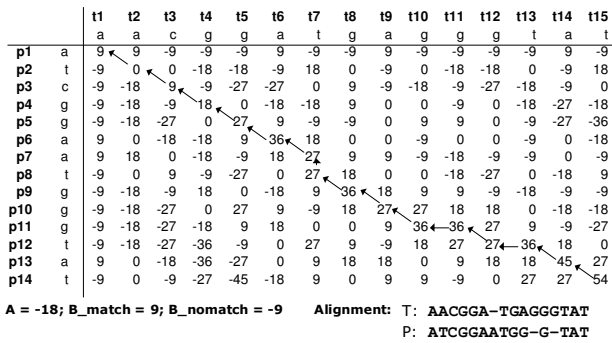


Figure 3: Alignment Extraction by Tracing Back Through DP Matrix

element $d(i, j)$ exceeds a user-defined constant, *Threshold*. This threshold can be used to specify the number of characters and gaps (as both determined by the constants A , B_{match} , and $B_{nomatch}$) that the desired alignment should have. The alignment can then be derived by following the pointers between adjacent $d(i, j)$ elements from the element that exceeded the threshold, back up to the first row, p_1 , or back to the first column, t_1 , of the DP matrix. A pointer for a particular $d(i, j)$ element points back to the element west, northwest, or north of it, that was selected by the $\max()$ function in Equation 4 in calculating $d(i, j)$. Figure 3 illustrates such a path taken along these pointers, to find the alignment that resulted in the element $d(14, 15) = 54$. Each diagonal arrow pointed toward an element $d(i, j)$ represents an exact alignment of the characters p_i and t_j in the final alignment. Each vertical arrow represents a gap inserted before the character t_j in the target t to maximize alignment, and each horizontal arrow a gap inserted before the character p_i in the pattern p . Indeed, there may be multiple possible paths of pointers to follow back up to the first row or column (as there are in Figure 3), and this would represent multiple alignments that result in the same value for the particular $d(i, j)$ that exceeded threshold.

As shown in Equation 4, and in the arrows radiating from $d(4, 4)$ in Figure 2, the value of the DP matrix element $d(i, j)$ depends on only the matrix elements to the west, northwest, and north of it (along with the characters p_i and t_j and the constants A , B_{match} , and $B_{nomatch}$). This is an important property of the Smith-Waterman algorithm, since it enables a straightforward implementation in parallel hardware, in that the matrix may be implemented as a systolic array of atomic processing elements (PEs), where each PE is responsible for a single $d(i, j)$ element. However, because the length of the target string t , i.e., the dimension n , is expected to be significantly larger than the length of the pattern string p , i.e., the dimension m , implementing the entire DP matrix would likely require an impractical amount of hardware. Thus, existing implementations typically involve some form of partitioning along the t axis (and possibly also the p axis), such that only a portion of the matrix is computed at a time.

4. SEARCHENGINE PROTOTYPE SYSTEM

4.1 Overview

The search engine prototype system, illustrated with a top-level block-diagram in Figure 4, consists of three primary components:

- FPX [5] programmed to snoop the ATAPI/IDE bus and forward data output from the hard drive to the interconnection network
- WUGS [8] switch fabric, used as the interconnection network between both FPXs
- FPX programmed with the search engine

As described in Section 1 the hard drive streams data to be searched onto to the ATAPI/IDE bus, where they are captured and buffered by the Bus Snooper FPX. The Snooper FPX repackages the data into ATM cells and forwards them via the WUGS switch fabric to the second FPX containing the search engine. The second FPX performs Smith-Waterman sequence alignment on the captured data as it streams from the switch fabric, and sends match results at regular intervals to a workstation connected to the switch fabric. The workstation which accepts the match results may be the same machine that houses the ATAPI/IDE bus being snooped.

4.2 IDE Bus Snooper FPX

The Bus Snooper FPX samples all signals on an ATAPI/IDE bus, interpreting the ATAPI/IDE protocol so it can locate data bursts from the hard drive and buffer the contents of those bursts in the FPGA's SRAM. The Snooper then forwards buffered data into the WUGS switch fabric, which routes those data to the search engine FPX. Voltage incompatibility prevents the Bus Snooper FPX from sampling the ATAPI/IDE bus directly, and so it only passively observes bus. Read or write commands issued to the hard drive being snooped would traverse a separate control path, i.e., by having the host workstation mount a file-system on the hard drive and then read out the target database with the UNIX tool "cat."

The 40.5MB/sec throughput limitation mentioned above is the maximum rate at which the hard drive could dump

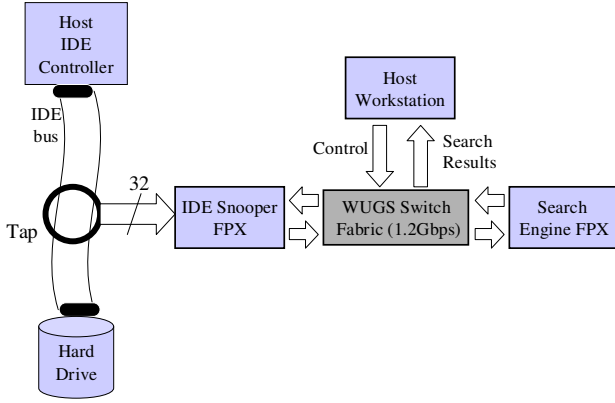


Figure 4: Block Diagram of Prototype System with Search Engine

consecutive data onto the ATAPI/IDE bus. The search engine FPX and the WUGS switch fabric, nevertheless, can sustain a throughput of 100MB/sec. As the prototype system was envisioned as an experimental device meant to prove the concept of the FPGA-based search engine, we found these shortcomings acceptable.

4.3 Search Engine FPX

The FPX programmed as the search engine has three primary components downloaded onto its FPGA:

- Control Module, responsible for routing data and control parameters to the other two components of the search engine
- Snapshot Manager, responsible for extracting the state of the search engine’s kernel, and for transmitting that with a match flag back to the host workstation
- Systolic Array, the search engine’s kernel, which, given 4 bytes of target data each clock cycle, computes 4 columns of the DP Matrix

The vast majority of FPGA gates used in this implementation are used in the Systolic Array. The primary purpose of the Control Module and Snapshot Manager is to accommodate conventions of the WUGS/FPX infrastructure (e.g., ATM data and control cell formats). It should be noted that future versions of this search engine will migrate to reconfigurable hardware development platforms more specialized than the WUGS/FPX infrastructure, ideally ones which let the search engine operate independently.

4.4 Systolic Array

Figure 5 shows a conceptual block diagram of the Systolic Array, which computes 4 columns of the DP matrix per clock cycle. At top of the array is a 32-bit register which accepts a group of 4 target characters from the Control Module on each clock cycle. To the left of the array’s first column are 8-bit registers holding the pattern characters, with each character aligned to a row of the array. The longest pattern which the array can accept is simply that with the same number of characters as the array has rows, i.e., a pattern m characters long. As explained below, this value m is actually parameterized, making the Systolic Array scalable up to the gate capacity of the FPGA.

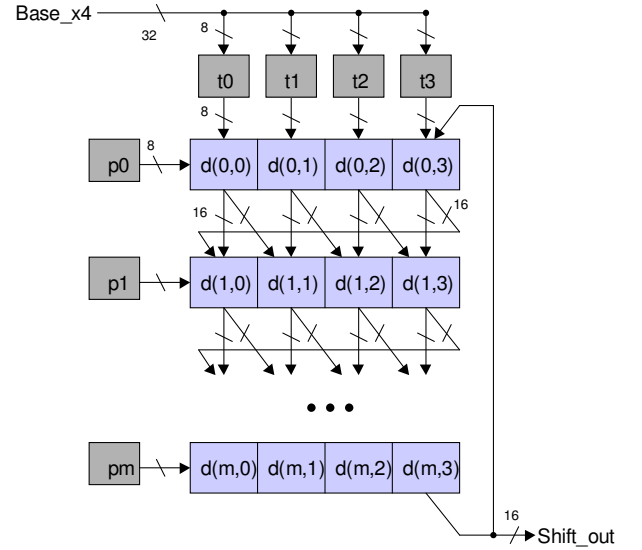


Figure 5: Block Diagram of Systolic Array

A width of 16 bits was chosen for all elements in the Systolic Array, including for the parameters A , B_{match} , $B_{nomatch}$, and the match threshold. Since the target and pattern characters only affect the value of $B(i, j)$, i.e., whether it is B_{match} or $B_{nomatch}$, the width of the target and pattern characters is independent of the width of elements in the array. To allow the search engine to operate directly on bytes, e.g., 8-bit ASCII characters, a width of 8 bits was thus chosen for the target and pattern characters. Although the search engine is intended to operate on streaming target data, i.e., such that the length of the target string can be effectively infinite, the signed 16-bit arithmetic of Systolic Array can indirectly impose a limit if calculation over- or underflows.

The 16-bit signals between each row illustrate the data dependencies between elements in adjacent rows, namely that computation of each element’s value depends on the values of the element’s North and Northwest neighbors in the row above. These would correspond to the values $d(i-1, j)$ and $d(i-1, j-1)$ from Equation 4, respectively. Because of these dependencies, the rows of the array must be computed in pipelined fashion, with each group of 4 target characters cascading down the array, one row per clock cycle. Note that each row actually has its own 32-bit register for holding target characters; Figure 5 only depicts that of the top row for clarity. Also note the Northwest signal must wrap around at the end columns, as the row-wise pipelining means the Northwest neighbor of the first row element is the fourth element in the row above.

Snapshot data are extracted from the array via the “shift_out” signal to the right of the array’s fourth column, with the signal’s branch that extends up to the top row turning that column into a large rotate register. This arrangement lets the array return to its original state once the snapshot extraction is complete, and thus resume computation where it left off.

An unusual feature of this implementation of the Smith-Waterman algorithm is that it computes the DP matrix in columns rather than in the more customary diagonals. This

of the inputs is selected as the maximum) would be the same across each equation. That is, if the result of the operation $F = \max[d(i-1, 0); d(i-1, -1)]$ is that the first of the two terms is greater, then the result of $F + A = \max[d(i-1, 0) + A; d(i-1, -1) + A]$ would be the same. Thus, when implementing the circuit that performs these maximum operations, one may save space by implementing all occurrences of F , $F + A$, $F + 2A$, etc. with a single comparator and then several multiplexers. Indeed, this is precisely the optimization implemented in the computational logic of each row in the Systolic Array, specifically for the terms F , H , and K . The tradeoff for this grouping of common factors is that more stages of logic are required than for the speed optimal circuit described above, increasing propagation delay. For the development of the search engine, both speed and space constraints were considered with equal weight, leading to a design with both speed and space optimizations blended.

A final note on the Systolic Array goes to the 3-stage pipelining of the per-row calculation illustrated in Figure 6. The terms $2A$, $3A$, and $4A$ are pre-computed, although not in the pipeline shown above since the parameter A is loaded into the Systolic Array several cycles before the systolic array would be enabled by the Control Module. The first stage of the pipeline accepts the 4 incoming target characters t_0 through t_3 and compares them with the pattern character p_i to calculate $B(i, 0)$, $B(i, 1)$, $B(i, 2)$, and $B(i, 3)$, which are then stored in registers between the first and second pipeline stages. The second stage then performs all calculation to reduce the remaining input values to 8 16-bit terms, namely the inputs to the 4 maximum operations shown in Equations 5 through 8, and stores the terms in additional pipeline registers. The third stage performs these remaining 4 maximum operations to obtain the final values for the row elements, and stores these values in the registers labeled “Cell0” through “Cell3” in Figure 6. After completion of the third stage, the new values of the registers “Cell0” through “Cell3” are compared to the user-defined match “Threshold,” with the comparison results for all elements in the Systolic Array OR-ed together and sent as a single bit to the Snapshot Manager.

4.6 Match Result Reporting

The Snapshot Manager handles the extraction of snapshot data and match results from the Systolic Array, and packages those data into ATM cells to send back to the host workstation. This snapshot reporting is done at periodic intervals of target data, with each snapshot containing a match flag indicating whether an element in the Systolic Array exceeded the user-defined *Threshold* since the last snapshot was made. This convention for result reporting allows the search engine to maintain a constant search throughput, regardless of the number matches the engine finds. Since the search engine can not feasibly perform the alignment extraction itself and still maintain its search throughput, the host workstation monitoring the snapshots must reconstruct in software the DP Matrix between adjacent snapshots, and then perform the alignment extraction procedure outlined in Section 3.

As the systolic array must be stopped to extract a snapshot of its state, it is necessary to make the snapshot period quite large (e.g., 10^6 target characters) to keep up the target data throughput. A larger snapshot period value does,

however, increase the off-line computation required to reconstruct the DP matrix between consecutive snapshots for alignment extraction. Nevertheless, if the hit rate of the pattern being sought is expected to be low, this off-line computation would be negligible.

5. PERFORMANCE COMPARISON

5.1 Overview

This section presents performance data collected for a software implementation of the Smith-Waterman algorithm, meant to imitate input and output of the Systolic Array of the hardware search engine described in Section 4. This section also points out specifically where the performance of this uniprocessor-based application suffers from constraints not present in the FPGA-based search engine.

The relevant configuration of the host workstation on which the tests were performed is as follows:

- SMP workstation with two Intel Pentium-III 933 MHz processors ²
- 512 MB RDRAM
- Redhat Linux version 7.2 operating system
- Promise TX-2 ATAPI/IDE controller
- Seagate ST320414A 20 GB ATAPI/IDE hard drive, formatted with an ext2 file-system
- ATAPI/IDE bus with an Innotec Design, Inc. ID620a bus analyzer

Execution time measurements were derived from the RDTSC cycle counter in workstation’s the Pentium processor to achieve sub-microsecond precision.

The experiment runs of the software implementation all used sets of artificially generated target data files of varying size. The use of separate files, as opposed to consecutive executions with the same input file, was necessary to prevent caching operations in the operating system and the hard drive itself from affecting the performance measurements. The varying size of the target data files also allowed the execution time measurements to be plotted against target data size, with the derivative of the resulting trend line yielding a sec/byte search throughput value. This throughput value could then be compared directly against that of the hardware search engine to obtain a quantitative performance gain. An ideal search throughput value, where processing time of the search application did not affect the throughput, but where the inherent limitations of the hard drive-CPU data-path shown in Figure 1 did have an effect, was obtained with the Linux tool “hdparm.” This throughput value was shown by the bus analyzer to be 40.5 MB/s, which is ideal because “hdparm” simply read an arbitrary, consecutive portion of the hard drive’s contents to test its transfer rate.

The target data files used in all performance tests were plain ASCII text files. The portions of the target data files that were not matches planted by the author were populated with a single character. This deterministic composition of

²The Smith-Waterman implementation was not compiled for multi-processing. Thus, its execution time on this machine would be comparable to that on a uniprocessor machine.

the target data files, as opposed to a non-deterministic composition one would find in random data or even genuine bio-sequence data, allowed the behavior of the software implementation to be precisely quantified in respect to the number of pattern matches in the target data. The files ranged in size from approximately 10 KB to 200 MB, although performance tests involving large numbers of planted matches started with larger target data files to accommodate the number of matches. The target data files were written to an ext2 file-system stored on hard drive, written in the same order in which they would be read during a performance test. This improved the probability that the target data files for a particular test would be written to a roughly contiguous area on the hard drive's platters, and thus minimize variation in the hard drive's access time.

5.2 Software Performance

This section presents the results of performance tests run on the author's C++ implementation of the Smith-Waterman algorithm, as a function of pattern length (and, thus, systolic array length). The Smith-Waterman implementation, dubbed "biocomp," is trivial, in that the algorithm has been implemented as-is, i.e. with $O(m \times n)$ execution time for a target of size n and a pattern of size m , and it mimics the input and output of the FPX search engine's systolic array. Nevertheless, these performance results illustrate quantitatively the performance gain the FPGA-based search engine enjoys through parallelism. "biocomp" was implemented such that the composition of the target and pattern string (and the resulting number of matches) would not alter any conditional branching during execution, thus making the specified pattern string and composition of the target data file irrelevant to the performance results. The Snapshot Period was set to 1 MB for all tests, making the overhead processing time required to parse each snapshot negligible compared to the total execution time.

Figure 7 shows "biocomp's" execution time on an array of target data files plotted against the files' size. The multiple lines represent performance tests with patterns strings of different lengths. For each performance test an approximation of the input byte/sec throughput was calculated using linear regression. Figure 8 shows the ATAPI/IDE throughput measured by the bus analyzer during the performance tests shown in Figure 7. The throughput for each data point was calculated by dividing the number bytes that traversed the IDE bus in each "biocomp" execution by the total time duration of the IDE activity observed during that execution. Figure 8 also lists the weighted average throughput values for each performance test, where the target data files' sizes were used for weighting. These throughput values correspond roughly with the estimated throughput values listed in Figure 7. The correspondence, however, is not exact due to the lack of perfect synchronization between the IDE bus events monitored by the bus analyzer and the "biocomp" execution time measurements. Furthermore, block granularity of the input data, i.e., at the file-system's level and at the physical level on the hard drive's platters, ensured that the amount of input data that traversed the IDE bus during each "biocomp" execution was greater than the size of the target data file. This artifact has a negligible effect on the average throughput values because of the weightings used. What is strikingly evident in Figures 7 and 8 is the dramatic dependency of "biocomp's" execution time on the pattern

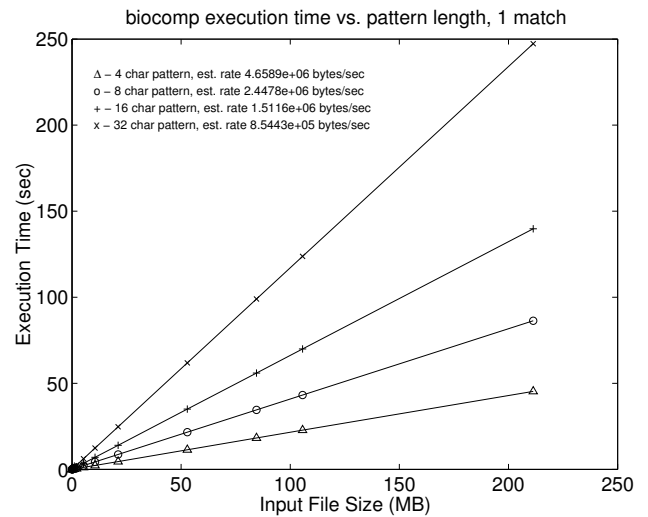


Figure 7: Smith-Waterman execution times vs. pattern length

string length. This is hardly a surprise, as the $O(m \times n)$ execution time makes the two values inversely related.

5.3 Comparison to Search Engine

Computing a regression line over the estimated search throughput values from Figure 7 versus the inverse of the pattern string lengths yields a slope of 1.716×10^7 [(target bytes/sec) \times (pattern bytes)], with the intercept 3.572×10^5 (target bytes/sec). This regression line is plotted in Figure 9. By this metric, the software implementation "biocomp" would exhibit a search throughput of 8.088×10^5 target bytes/sec for a 38-character pattern string (i.e., the maximum pattern length for the search engine). This gives the FPGA-based search engine, which has a constant search throughput of 100 MB/sec, a performance gain of 124 over its direct, albeit trivial, software implementation. (This gain is neglecting the 40.5 MB/sec limitation imposed by the hard drive itself, which would reduce the gain to 50.1.) Indeed, given the inverse relationship of "biocomp's" execution time to the pattern string length, this gain would only increase with larger pattern strings.

An alternate performance metric used specifically in bio-sequence analysis is the rate of cells (i.e., DP Matrix elements) updated per second, or CUPS. Below is a list of the CUPS values for the prototype, the trivial software implementation "biocomp," and a commercial bio-sequence analysis device.

- "biocomp," 38-byte pattern: 30.7 MCUPS
- Compugen Bio XL: 3.0 GCUPS³
- FPGA-based Search Engine, 38-byte pattern: 3.8 GCUPS

6. SUMMARY

We have presented in this paper an FPGA-based search engine which performs the Smith-Waterman local alignment algorithm on generic data at the byte level, and which is capable of sustaining a search throughput of 100MB/sec. We

³CUPS value advertised for smallest, single-board configuration. Source: www.cgen.com.

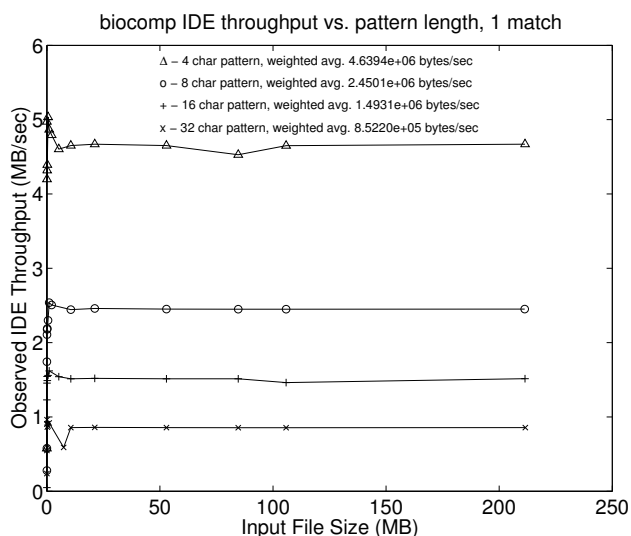


Figure 8: Smith-Waterman IDE bus throughput vs. pattern length

have also presented the performance of the search engine in a prototype system constructed from off-the-shelf components and experimental, high-speed reconfigurable hardware. In this prototype system we can sustain a search throughput that matches the maximum throughput from the hard drive storing the data, 40.5MB/sec.

This search engine and its prototype platform was borne out of the desire to investigate the benefits of placing search modules implemented in reconfigurable hardware as close to the hard drive's storage media as possible. The prototype system reflects this desire, as it sits on the ATA/IDE peripheral bus and directly snoops the data output from the hard drive. The prototype's proximity to the storage medium, along with the parallelism and pipelining of its FPGA-based implementation, allow the prototype to enjoy a performance gain of up to 124x over a comparable, uniprocessor-based software implementation.

The shortcomings evident in this prototype system, namely the throughput limit of the hard drive itself and the 38-byte limit on patterns because of finite FPGA gate capacity, are all appreciated by the authors as opportunities for future improvement. For example, optimizing the search engine for minimal bit-encoding of bio-sequence data would yield an implementation that lets us fit more rows of the Systolic Array on an FPGA, and thus accommodate longer patterns. In addition, the capacity could be developed to chain together multiple FPGAs programmed with the search engine, and thereby create a larger, virtual array that also allows longer patterns. Only cost would determine the extent of customization appropriate for an implementation of this search engine, as it could operate readily on an inexpensive, PCI-based FPGA evaluation board that reads target data from an off-the-shelf RAID, or it could be migrated to reconfigurable hardware inside the hard drive itself, thereby eliminating the complexity of interfaces like the Bus Snooper FPX and thus realizing an ideal data-path between the storage medium and the processing elements.

7. ACKNOWLEDGEMENTS

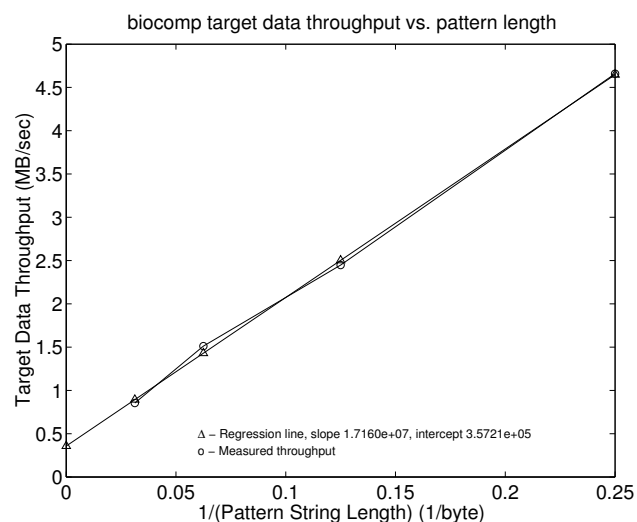


Figure 9: Smith-Waterman target data throughput vs. pattern length

The authors wish to thank Dr. Jeremy Buhler and Dr. John Lockwood for their assistance with this project.

8. ADDITIONAL AUTHORS

Additional authors: Qiong Zhang (Department of Electrical Engineering, Washington University, email: mzhang@ee.wustl.edu)

9. REFERENCES

- [1] R. D. Chamberlain, R. K. Cytron, M. A. Franklin, and R. S. Indeck. The Mercury System: Exploiting truly fast hardware in data mining. New Orleans, LA, September 2003. International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI).
- [2] S. A. Guccione and E. Keller. Gene matching using JBits. In *12th International Field-Programmable Logic and Applications Conference (FPL)*, Montpellier, France, September 2002.
- [3] D. Lavenier. Speeding up genome computations with a systolic array. *SIAM News*, 31(8):1-7, October 1998.
- [4] J. W. Lockwood, C. Neely, C. Zuver, J. Moscola, S. Dharmapurikar, and D. Lim. An extensible, system-on-programmable-chip, content-aware Internet firewall. In *Field Programmable Logic and Applications (FPL)*, page 14B, Lisbon, Portugal, Sept. 2003.
- [5] J. W. Lockwood, J. S. Turner, and D. E. Taylor. Field programmable port extender (FPX) for distributed routing and queuing. In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000)*, pages 137-144, Monterey, CA, February 2000.
- [6] J. Reynders. Computing biology. November 2001. In invited talk at 5th High Performance Embedded Computing Workshop.
- [7] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 141(1):195-7, 1981.
- [8] J. Turner, T. Chaney, A. Fingerhut, and M. Flucke. Design of a Gigabit ATM Switch. In *Proceedings of Infocom 97*, Mar. 1997.