Sequence Analysis With the Kestrel SIMD Parallel Processor

Leslie Grate, Mark Diekhans, David Dahle and Richard Hughey
Department of Computer Engineering
University of California, Santa Cruz, CA 95064
{leslie,rph}@cse.ucsc.edu www.cse.ucsc.edu/research/kestrel
Pacific Symposium on Biocomputing 2001 (pp. 263-74)

Abstract

Computer aided sequence analysis is a critical aspect of current biological research. Sequence information from the genome sequencing projects fills databases so quickly that humans cannot examine it all. Hence there is a heavy reliance on computer algorithms to point out the few important nuggets for human examination. Sequence search algorithms range from simple to complex, as does the representation of the biological data. Typically though, simple algorithms are used on the simplest of data representations because of the large computational demands of anything more complex. This leads to missed hits because the simple search techniques are often not sufficiently sensitive.

Here we describe the implementation of several sensitive sequence analysis algorithms on the Kestrel parallel processor, a single-instruction multiple-data (SIMD) processor developed and built at UCSC. Performance of the Smith-Waterman and Hidden Markov Model algorithms, with both Viterbi and Expectation Maximization methods ranges from 6 to 20 times faster than standard computers.

Keywords: Sequence analysis, Smith-Waterman, Hidden Markov models, SIMD parallel processing.

1 Introduction

The most familiar sequence analysis algorithm is BLAST [1], a very fast serial-machine sequence search algorithm. It is popular because it is fast and is available for use over the world-wide-web via a simple user interface. The output from a BLAST search is a score and an "alignment" between the input query sequence and the "good" hits found in the searched database. However, BLAST is a search algorithm, not an alignment algorithm, so these "alignments" are not as high quality as could be. More importantly, distantly related sequences that would be found by a true sensitive alignment algorithm are missed.[2]

[a] This work was supported in part by NSF grant EIA-9905322.

Alignment algorithms such as Smith-Waterman[3] (SW) and Hidden Markov models[3,4] (HMMs) require more computational effort and hence are not normally used to directly search databases. This leads to a conundrum — we want to search databases with a sensitive search method such as an alignment algorithm, but we do not because of the extra computer time needed.

This computational bottleneck can be relieved by specialized computer hardware designed to run these algorithms efficiently. Here we describe our use of the UCSC Kestrel parallel processor, a single instruction multiple data (SIMD) 8-bit processor designed and built at UCSC, on these alignment algorithms. While Kestrel was designed primarily for these algorithms, it was also designed to be as versatile as possible.[5] In addition to achieving high performance on these algorithms, Kestrel also accelerates a number of other applications, such as combinatorial chemistry fingerprint searching, neural networks and graph algorithms. The performance improvement over standard high-end serial computers can be as high as a factor of 20.[6,7,8]

2 Kestrel

The Kestrel Parallel Processor is currently implemented as a single-board system with 512 processing elements (PEs) shown in Figure 1. The system is composed of the PE array, an array controller, instruction memory, a PCI interface unit, and input and output queues to provide synchronization with the PCI bus. The system runs on WindowsNT, OSF, and Linux hosts with no significant performance differences.

Kestrel is a SIMD (single-instruction stream, multiple-data stream) parallel processor. Instructions are broadcast to the array but PEs can be independently turned on and off. Every clock cycle, each of the 512 PEs that is "on" performs the broadcast instruction on its local data. The PEs are connected linearly, each able to communicate with its left and right neighbors. Data can be fed in or collected at either end of the array, or broadcast by the controller to all processing elements. The PEs are small to keep the array physically small. The array has an 8-bit (1 byte) word size, and each PE has 32 registers and 256 bytes of local SRAM (static random access memory). The PE can perform most standard arithmetic/logic functions, has an integrated minimizer, a signed 8×8 multiplier and support for efficient multi-byte arithmetic. Each full custom VLSI Kestrel chip contains 64 identical PEs, Figure 2.

SIMD machines broadcast the same control signals to all processors in the array. For the highest efficiency, all data should be treated the same. The standard serial computer language `if-else` statement allows one of two parts of a program to execute: however in SIMD machines the `if` condition

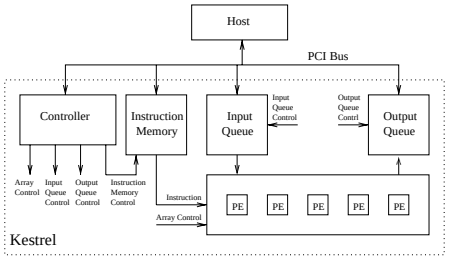

Figure 1: The Kestrel single PCI board system. (Left) A picture of the board. The PCI bus interface is in the upper right, center is the FPGA controller, and on the left side are the 8 Kestrel array chips. (Right) Kestrel system block diagram. Once a program is loaded into the controllers memory, database data is fed to the array through the input/output queues. This board functions in IBM pc's and DEC (Compaq) alphas.

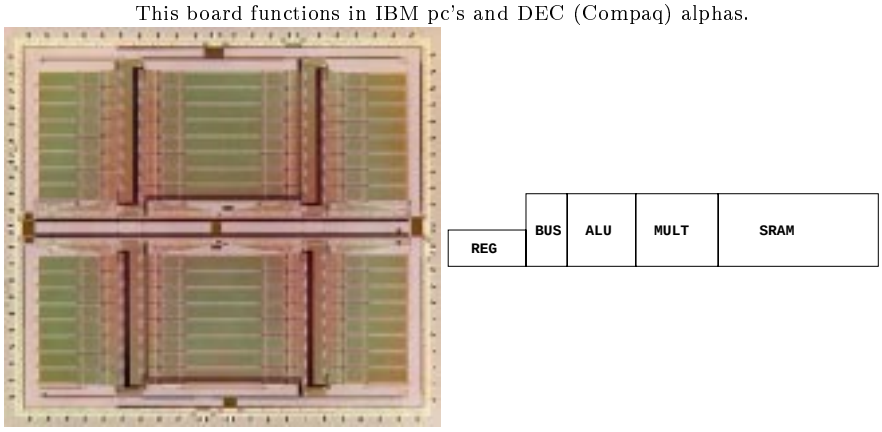

Figure 2: The Kestrel VLSI chip containing 64 identical PEs and the floor plan for each PE.

is parallelized, so that PE's that have a `TRUE` condition are "on" and execute the `if` part code, while the PE's that will operate the `else` part are "off". Inversion of "on/off" conditions activates the `else` part. Thus, all PE's see all instructions, but only execute some of them. Conditional statements on SIMD machines lead to increased rather than reduced instruction counts.

Kestrel must be plugged into a PCI bus in a host computer. We have two operational host types, older DEC Alpha's running the OSF operating system and standard PC's running both WindowsNT and Linux. The compute power of the host is not a factor for operation of the Kestrel boards, and cheap 300 Mhz PC's perform equally to the Alpha.

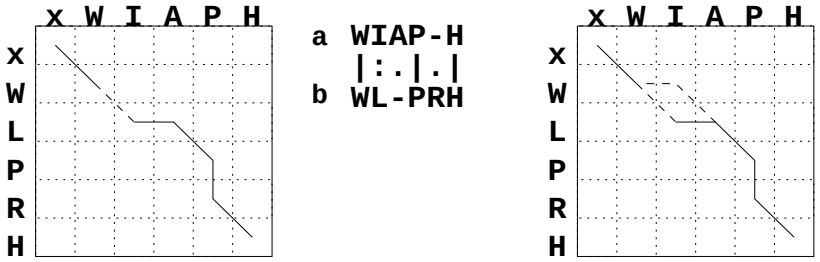

Figure 3: Small alignment example and resulting score matrix. The matrix lower right-hand element contains the score of the best alignment. In the alignment, | indicates a match, : a mutation, and . is delete (with respect to the top sequence) if the lower sequence has a - and insert if the upper sequence has a -. The left matrix is an example of the one best path (Viterbi) method matching this alignment, but maybe L should match A and I be the one that is deleted (both paths are shown in the right matrix), and biologically maybe this region of the sequence is not that important. In this case, the Expectation Maximization (EM or all paths) method would consider all paths through the cost matrix, including both these options.

3 Alignment Algorithms

We have implemented the scoring part of the Smith-Waterman and Hidden Markov Model[3,4] (both Viterbi and Expectation Maximization methods) alignment algorithms on Kestrel.

These alignment algorithms address the general problem of string-matching in the context of biological sequence data. The goal is to quantify or score "the" relationship between two strings. This is not well-defined, so different algorithms quantify different relationships. The fundamental assumptions these algorithms are based on are that the linear order of the characters matters, and that each character in a string can be analyzed independently from the others in that string. Characters are compared character-by-character with 3 possible outcomes: Match/Mutation, Delete, and Insert, and all possible configurations consistent with the linear order are examined using a two-dimensional matrix. Interrelated recurrence equations succinctly define the algorithm. A forward pass through the algorithm defines a two-dimensional matrix of scores and produces the score of the best alignment. The actual correspondences are then found by backtracking through this matrix. Figure 3 shows a small example of this matrix and a common method of showing an alignment. The two strings being compared are placed along the horizontal and vertical axes. Match/mutations occur along diagonals, inserts along vertical and deletes along horizontal.

At every matrix element, 3 values are computed corresponding to the best score so far if the characters defining the matrix element are matched, in-

serted or deleted. At the end of the algorithm, the lower right hand matrix element will contain the overall best score. This technique of using a matrix to efficiently solve recurrence equations is called dynamic programming. The following describes the Smith-Waterman algorithm:

$$I_{i,j} = \max \begin{cases} I_{i-1,j} + c \\ M_{i-1,j} + g \end{cases} \qquad D_{i,j} = \max \begin{cases} D_{i,j-1} + c \\ M_{i,j-1} + g \end{cases}$$

$$M_{i,j} = \max \begin{cases} I_{i-1,j-1} + d(a_i, b_j) \\ D_{i-1,j-1} + d(a_i, b_j) \\ M_{i-1,j-1} + d(a_i, b_j) \\ 0 \end{cases}$$

where $d(a_i, b_j)$ is the cost of matching character a_i to b_j, g is the cost of starting a gap, and c is the cost of continuing a gap. The inclusion of the the constant 0 in the Match equation implements "local" scoring, meaning that the algorithm finds the best matching sub-strings. This is a desirable feature for biological sequence alignment. Without the constant 0, the equations implement "global" scoring, where the best match is found for both strings in their entirety.

HMMs generalize the above equations.[4] The query sequence is generalized to a more complicated structure, called a "model" (the model part of HMM), that can statistically describe a group of sequences rather than just a single sequence. The characters in the sequence are generalized to "nodes" that contain position-specific match tables (the $d(a_i, b_j)$ values) and also a table for insert, instead of the global g and c. Between the nodes are transition costs. A small model is shown in Figure 4. The equations defining HMM Viterbi global scoring are:

$$I_{i,j} = e(I_j, x_i) + \min \begin{cases} M_{i-1,j} + tr(M_j, I_j) \\ I_{i-1,j} + tr(I_j, I_j) \\ D_{i-1,j} + tr(D_j, I_j) \end{cases} \qquad D_{i,j} = \min \begin{cases} M_{i,j-1} + tr(M_{j-1}, D_j) \\ I_{i,j-1} + tr(I_{j-1}, D_j) \\ D_{i,j-1} + tr(D_{j-1}, D_j) \end{cases}$$

$$M_{i,j} = e(M_j, x_i) + \min \begin{cases} I_{i-1,j-1} + tr(I_{j-1}, M_j) \\ D_{i-1,j-1} + tr(D_{j-1}, M_j) \\ M_{i-1,j-1} + tr(M_{j-1}, M_j) \end{cases}$$

Additions into the above equations implement Smith-Waterman-style Viterbi Local Scoring: [9]

$$M_{i,j} = \min \begin{cases} M_{i,j} \\ Jin_{i,j} \end{cases} \qquad Jin_{i,j} = \; Jin_{i,j} + NullTr(I, I) + NullE(I, x_i)$$

$$\text{if NOT (FirstTwoModelNodes) then } Jout_{i,j} = \min \begin{cases} Jout_{i-1,j-1} \\ M_{i,j} \end{cases}$$

$$\text{if (LastModelNode) then } D_{i,j} = \min \begin{cases} D_{i,j} \\ Jout_{i-1,j-1} \end{cases}$$

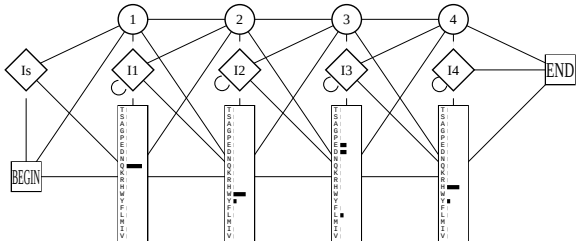

Figure 4: A tiny SAM HMM. The large rectangles with the amino acid distributions are the Match states, the circles are Delete and diamond is insert. The transitions are the lines between the states. Each state has 3 in and 3 out transitions in the left-to-right direction. Inserts have amino acid distributions (like Match) which are not drawn. Each vertical group of Delete, Insert and Match make up a node, with the number noted in the Delete state. For example, you can leave a match state and go into the insert directly above (in the same node) or the delete or match of the subsequent node.

where $tr(s1, s2)$ is a per-node 3x3 matrix of transition costs (the lines connecting nodes in Figure 4), $e(s, c)$ is a per-state alphabet-sized array of emission costs (the Match and Insert amino acid distributions). For Local scoring, $Jin(s)$ and $Jout(s1, s2)$ are the jump-in and jump-out costs and $NullTr$ and $NullE$ are the null model transition and emission costs (equivalent to 0 in the SW equations). As these equations use min, they implement the Viterbi or single-path method, the result being the score of the one best path through the cost matrix (as in Figure 3). The more sensitive Expectation Maximization (EM) method replaces min with sums of probabilities and the sums with multiplication of probabilities. The resulting score is the sum of the probabilities of all paths through the cost matrix, leading to the increased sensitivity of this method to distantly related sequences. The right hand cost matrix in Figure 3 is a tiny illustration of this. Suppose the biologically important parts of the sequence are the WPH and the rest does not matter. Then what really should be scored are the WPH, and the other letters should not be included. EM does this by including in the score all paths, so it would not matter how the IA and L are aligned. The Viterbi method must choose exactly one path, so the score will be less than the EM score.

The dynamic programming calculation easily maps to a linear array of processing elements and was the main reason for much of Kestrel's design.[7] A common mapping is to assign one PE to each character/node of the query string/model, and then to shift the database through the linear chain of PEs. As shown in Figure 5, the query is loaded into the PE array along with the d or e & tr values, and the database streamed through. The query is indexed by j

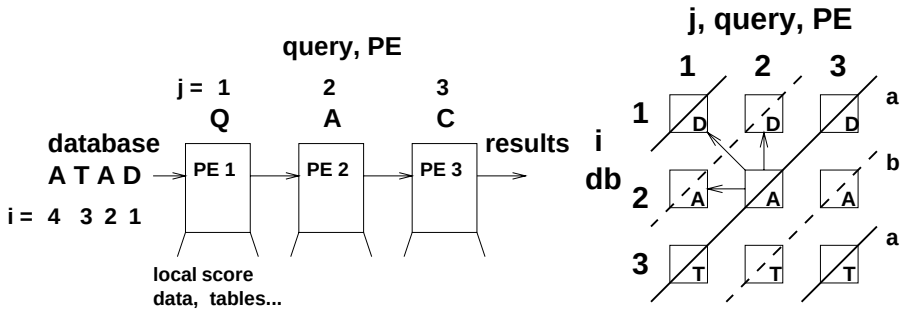

Figure 5: Left shows the mapping of the recurrence equations to the Kestrel PE array architecture. The query is loaded into the array, and stays fixed for the rest of the run. The database is streamed through. The query is indexed by j and sequence in the database by i. The recurrence only needs values that are easily available in neighboring PE's. Right shows how this mapping operates over time as the database is fed through it. The calculation proceeds along the diagonal of the query by database-sequence matrix. The center PE needs to use values from $i-1, j$ (for Insert, above), $i, j-1$ (for Delete, left) and $i-1, j-1$ (for Match, left above diagonal). The left and above values have just been computed the previous cycle, but the diagonal element (match) was computed two cycles ago, so it must be saved. This is done by having alternate storage locations for the match values, labeled as a (solid line) and b (dashed line). The a phase uses the diagonal a value, and the b phase uses the diagonal b value.

and sequence in the database by i. Neighboring PEs store the -1 values needed by the equations. With this mapping our implementations compute along the diagonal of the matrix of query (mapped to PE) by database sequence, Figure 5. Queries longer than the 512-PE array length can be handled by storing several adjacent characters in each PE's local memory. When one is scoring a number of queries against a database, part of the array is wasted unless the queries fill the array. The utilization of the array is improved by packing in as many queries as can fit because the overhead to keep the scores separate and extra output requirements are less than running each query separately. The twin problem of sequence alignment, finding the minimizing correspondence between two sequences, requires the saving of the selector bits of the minimizations and recirculation of sequence data[10], which we do not presently do.

We have implemented 6 main variants of the HMM algorithm on Kestrel. These are Viterbi Global 32-bit, Viterbi Local 32-bit, Viterbi Local 24-bit, Viterbi Local Multiple models 24-bit, EM Global 32-bit and EM Local 32-bit. The 24-bit algorithms require fewer instructions at the cost of loss of dynamic range. Mathematical overflow is a problem in the 24-bit HMM code, and those implementations perform overflow checking, so that when a value gets large, it stays large and does not wrap around. The EM method requires

multiplications, which Kestrel can handle because each PE has a multiplier. Smith-Waterman has enough dynamic range using only 16-bit values and our implementations support query lengths to 2555.

All these implementations have the same form: Initialization, Loading the query/model, and streaming the database through Kestrel iterating computation with result output. One result score is produced for each database sequence; therefore, there is much more input data than output data. Kestrel performance is weakly dependent on database composition and is constant with respect to query/model length (up to query length 511 for HMM's and modulo 512 for Smith-Waterman).

The Kestrel HMM and Smith-Waterman algorithms have been integrated into version 3 of UCSC's SAM HMM system.[4,11,12] We have a Smith-Waterman server available at our web site, www.cse.ucsc.edu/research/kestrel.

4 Example Smith-Waterman Code

The Kestrel assembly code in Figure 6 is a (relatively) easy-to-read example of just the SW recurrence equations using 16 bit values and is not the actual code used in the system. Each line of assembly code specifies one or more opcodes and modifiers, and an appropriate number of operands, to specify array functions. The code in actual use is more intricate in how it performs reading in the database and producing results. Because SW only needs about 20 instructions per database character, an extra 1 or 2 instructions is a significant reduction in speed.

Between every sequence in the database is a special "end of sequence" mark character. When a PE detects the mark, it must perform extra instructions to push the result value out (to the right in Figure 5) and then reset the registers so they are ready for the next sequence. This same type of check must happen to detect "end of database". While it is straightforward to do these checks in Kestrel, it adds many extra instructions. Efficiently performing these checks with the smallest impact on speed is quite tricky. The current implementation automatically switches between a fast inner loop (19 instructions) and a slow inner loop (23 instructions), depending on whether or not an "end of sequence" or "end of database" character is present in any of the active PEs. Because of this, Kestrel runs a little faster when database sequences are longer than the query.

Packing multiple query characters in each PE allows longer query sequences. Conceptually this is just like adding more PEs to the array, except these are "virtual" PEs. The query dependent cost tables (d array) and recurrence variables are stored in each PEs local memory, and appropriate codings

```
;; Insert <-- max (Insert+continue,MGC)
add          R$Inslo, R$Inslo, $CONTL
smaxc add mp R$Inshi, R$Inshi, $CONTH, R$MGChi
smaxc cmp    R$Inslo, R$Inslo, R$MGClo

;;  Delete <-- max (Delete+continue,MGC)
add          R$Dello, L$Dello, $CONTL
smaxc add mp R$Delhi, L$Delhi, $CONTH, R$MGChi
smaxc cmp    R$Dello, R$Dello, R$MGClo

;; Shift the sliding sequence;
;; read a new value from the queue;
;; Lookup the character cost in SRAM
move R$Seq, L$Seq, qtoarr, read(L$Seq)

;; Match <-- max(MDI+charcost,0)
;; Zero-threshold for local scoring
;; of Smith & Waterman
add          L$TMPlo, L$MDIlo, mdr
smaxc add mp L$TMPhi, L$MDIhi, smdr, #0
smaxc cmp    L$TMPlo, L$TMPlo, #0

;; Add the gap cost to the Match cost for
;; Delete and Insert calculations
add R$MGClo, L$TMPlo, $GAP_LO
add R$MGChi, L$TMPhi, $GAP_HI

;; Store the best score so
smaxc     L$score_hi, L$score_hi, L$TMPhi
smaxc cmp L$score_lo, L$score_lo, L$TMPlo

;; MDI <-- max (Match, Insert, Delete)
;; for future Match calculation.
;; Branch to start of nested loop
smaxc     R$MDIhi, L$TMPhi, R$Inshi
smaxc cmp R$MDIlo, L$TMPlo, R$Inslo
smaxc     R$MDIhi, R$MDIhi, R$Delhi
smaxc cmp R$MDIlo, R$MDIlo, R$Dello, endLoop

;; mp   = multiprecision ALU op
;; cmp  = topdown multiprecision comparator op
;; smdr = sign extension of the 1-byte MDR
;; smaxc= signed maximum with operand C
```

Figure 6: Example core assembly code for Smith and Waterman dynamic programming.

of the recurrence equations are used. Query characters on the "inside" of a PE never need to use values from another PE because the "adjacent" query character is in the same PE. Because HMMs have many more variables, we can only do multiple character packing in SW. The current maximum query length is 2555, a packing of 5 query characters per PE. Many more are possible for DNA due to the smaller alphabet; 4 rather than 20 (although with ambiguity characters the alphabet is larger). The extra computation is directly proportional to the number of packed characters plus a tiny bit of overhead.

5 Results

All results here compare a single 20 MHz Kestrel board to a 433 MHz DEC (Compaq) Alpha (Alpha 21164 series; the newer 21264 series is twice as fast).

On Smith-Waterman, Kestrel is up to 20 times faster (Table I). This makes Kestrel slightly slower than the much larger 16K-processor MasPar [6,7] and faster than the similarly sized SAMBA system.[13]

Table II lists the instruction counts for the current Kestrel implementations. These are the number of instructions to implement the recurrance equations followed by the number needed to output results (SW for queries less than 512 is heavily optimized for speed). EM instruction counts are approximate (± 50) as there are variants that trade off accuracy for speed. The output overhead for Multiple Model Viterbi Local 24-bit (VL24multi) is 19 instructions, plus occasional large bursts of output requiring hundreds of instructions depending on the size and packing of the models.

Table III lists the throughput of the Viterbi algorithm. Bytes per Second is a better measure of performance in Kestrel algorithms as Kestrel does not care about database sequence length. Kestrel performance is constant independent of model length and database size or composition. The Kestrel speedup is

Table I: Wall time in seconds to SW search a 10Mbase database on the 20 MHz Kestrel.

	Kestrel	433 MHz DEC alpha		
Query Size	≤ 512	32	128	512
SW	13	20	73	282

Table II: Cell Instruction Counts.

SW16	VG32	VL24	VL24multi	VL32	EMG	EML
19/4	73/5	80/5	85/19+	111/5	~500	~720

Table III: Viterbi Scoring Bytes per Second Time Comparisons.

Length	Serial VG	Serial VL	VG32	VL24	VL32
50	112903	102868	275K	246K	180K
64	92581	78458	275K	246K	180K
128	46290	40252	275K	246K	180K
256	32599	19698	275K	246K	180K
511	22255	9849	275K	246K	180K

Table IV: EM Scoring Bytes per Second Time Comparisons.

Length	Serial EMG	Serial EML	EMG	EML
100	21667	17333	28888	22807
250	9167	6833	28888	22807
500	4833	3667	28888	22807

nearly linear with the maximum about 25 when the array is full (model length around 510) using the Local 24-bit or Global 32-bit. Running multiple models at the same time in Kestrel, for example two 255 long models, gives a speedup of 20+ over the two (or more) serial program runs. Multiple models is only implemented in Local 24-bit because Local database search is more sensitive than global, so it is prefered.[2,12]

Table IV lists the throughput for the current EM algorithm. The speedups here are not as good as in Viterbi. This algorithm requires floating point operations on 32-bit values of which there are too many to keep in registers leading to excessive data movement between memory and registers. The maximum speedup is only about 6.

6 Future

The Kestrel-enhanced SAM system provides significant speed-up for large database searchs. Iterative methods such as SAM/Target98[12] require several such searches to complete a single query. Model library vs sequence database searches are also enhanced for shorter models and should be of interest to pharmacutical companies. Smith-Waterman search speeds of 1 megabyte-per-second allow searching 100 megabyte databases (such as the current size of the non-redundant protein database) in 100 seconds. This is fast enough for human interactive use.

The 20 Mhz Kestrel used here is just the begining. The current Kestrel VLSI chips are fully operational to 33 Mhz and the second generation board under construction will operate at that speed. This board will have 16 Kestrel chips, doubling the array length. The new board will also open up new applications that require on-board memory for storage and/or recirculation of data through the array. The new controller will be more efficent than the existing one and will add key features that will reduce the overhead associated with data output. All the algorithms described here will see performance improvements from these features.

A second generation VLSI chip design in a smaller feature size will allow both higher clock rates and more PE's per chip, possibly only needing a single large chip for 500+ PE's and on-chip controller. Then, Kestrel clock rates should reach into the 150+ Mhz range, leading to data rates on these algorithms at least 10 to 20 times faster. Such a system would truly be a low cost single-chip plug-in accelerator for sequence analysis.

7 Acknowledgments

We thank the entire Kestrel team for realizing a fully operational system, and our collaborators and users. The EM code was written by M. Diekhans, and the original Viterbi Global 32-bit code by D. Dahle. More information about Kestrel is available on our web site http://www.cse.ucsc.edu/research/kestrel.

8 References

1. S. F. Altshul, W. Gish, W. Miller, M. E. W., and L. D. J., "Basic local alignment search tool," *JMB*, vol. 215, pp. 403–410, 1990.
2. J. Park, K. Karplus, C. Barrett, R. Hughey, D. Haussler, T. Hubbard, and C. Chothia, "Sequence comparisons using multiple sequences detect three times as many remote homologues as pairwise methods," *JMB*, vol. 284, no. 4, pp. 1201–1210, 1998. Paper available at `http://www.mrc-lmb.cam.ac.uk/genomes/jong/assess_paper/assess_paperNov.html`.
3. T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Mol. Biol.*, vol. 147, pp. 195–197, 1981.
4. A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler, "Hidden Markov models in computational biology: Applications to protein modeling," *JMB*, vol. 235, pp. 1501–1531, Feb. 1994.
5. J. D. Hirschberg, D. Dahle, K. Karplus, D. Speck, and R. Hughey, "Kestrel: A programmable array for sequence analysis," *J. VLSI Sig-*

nal Processing, vol. 19, pp. 115–126, 1998.

6. D. Dahle, L. Grate, E. Rice, and R. Hughey, "The UCSC Kestrel general purpose parallel processor," *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, vol. III, pp. 1243–1249, 1999.
7. R. Hughey, "Parallel sequence comparison and alignment," *CABIOS*, vol. 12, no. 6, pp. 473–479, 1996.
8. A. Di Blas and R. Hughey, "Explicit SIMD programming for asynchronous applications," *Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 258–267, 2000.
9. C. Tarnas and R. Hughey, "Reduced space hidden Markov model training," *Bioinformatics*, vol. 14, no. 5, pp. 401–406, 1998.
10. J. A. Grice, R. Hughey, and D. Speck, "Reduced space sequence alignment," *CABIOS*, vol. 13, no. 1, pp. 45–53, 1997.
11. R. Hughey and A. Krogh, "Hidden Markov models for sequence analysis: Extension and analysis of the basic method," *CABIOS*, vol. 12, no. 2, pp. 95–107, 1996.
12. K. Karplus, C. Barrett, and R. Hughey, "Hidden markov models for detecting remote protein homologies," *Bioinformatics*, vol. 14, no. 10, pp. 846–856, 1998.
13. D. Lavenier, "Speeding up genome computations with a systolic accelerator," *SIAM News*, vol. 31, no. 8, pp. 6–7, 1998.