Kestrel: A Programmable Array for Sequence Analysis

Jeffrey D. Hirschberg
hirsch@cse.ucsc.edu

Richard Hughey
rph@cse.ucsc.edu

Kevin Karplus
karplus@cse.ucsc.edu

Don Speck

Computer Engineering, University of California, Santa Cruz, CA 95064

Abstract

Kestrel is a programmable linear systolic array processor designed for sequence analysis. Among other features, Kestrel includes an 8-bit word, a single-cycle add-and-minimize instruction, and efficient communication using Systolic Shared Registers. This paper describes Kestrel's functional units in detail, and examines each of their effects on system performance. With prototypes currently in the works, we expect to complete a full Kestrel array, with between 512 and 1024 processing elements, in 1997.

1: Introduction

Kestrel, named after the small, fast falcon found on the Santa Cruz campus of the University of California, is a project to develop a programmable linear systolic array for sequence analysis. The processing element (PE) architecture is complete, as well as a simulator for performance evaluation. We have fabricated two subunits, the multiplier and SRAM, and will fabricate a prototype chip with 1–4 full PEs in late 1996. A complete system, with 64 PEs per chip and 8–16 chips, is scheduled for late 1997. Kestrel will be an affordable system; a partially-populated Kestrel board could be priced in the low thousands of dollars, while a fully-loaded Kestrel board could cost in the low tens of thousands. This paper discusses the motivation for the Kestrel project and the PE architecture, including descriptions of the major architectural components that help Kestrel achieve its goals.

There are three main goals for the Kestrel project. The first is to develop a platform that is well-suited to biological sequence analysis. In projects such as the Human Genome Project, scientists need to analyze large databases containing billions of characters from DNA, RNA, and proteins. Many of the algorithms used in this analysis require scanning large segments of a database. Kestrel has been designed with these algorithms in mind.

The second goal is to provide a programmable architecture. In one sense, this is related to the goal of developing an efficient platform for sequence analysis. There are a great number of sequence analysis algorithms in computational biology, and programmability is required to accommodate these different algorithms within a single system. As new algorithms are developed, Kestrel will be able to execute many of them without the need for redesigning the architecture. Additionally, Kestrel will be able to execute unrelated algorithms suitable for linear arrays.

A third goal is to build a balanced system. The speed of data input and output (I/O) is a primary consideration in the design of any system, especially a massively parallel one. The speed at which the array operates needs to be balanced with the speed of I/O [18]. In the case of sequence analysis, the large databases involved are typically stored on disk. Most disks available today have a maximum sustained transfer rate of three to five megabytes. Therefore, we have structured the Kestrel PE architecture and cycle time to be in keeping with these transfer rates.

The next two sections will briefly review sequence analysis algorithms and architectures. Following this, we describe and justify the design choices that led to the Kestrel architecture. Finally, we evaluate the effects of these choices on performance.

1063-6862/96 $5.00 © 1996 IEEE

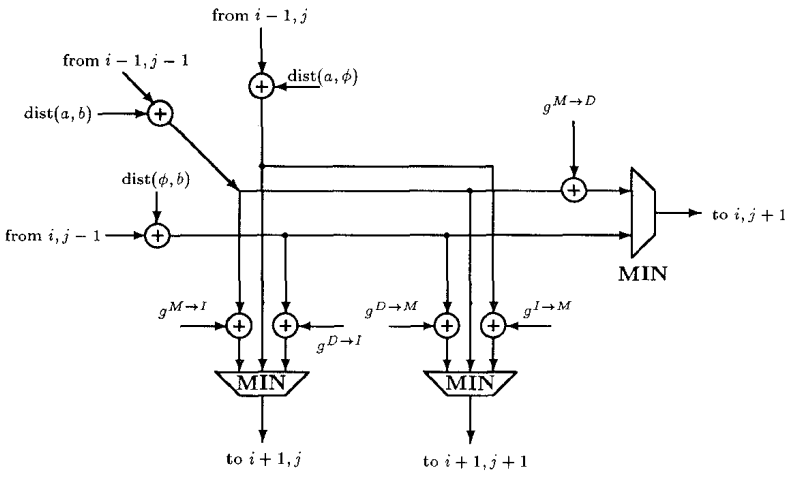

Figure 1. Dataflow for affine gap cost sequence comparison.

2: Sequence Analysis Algorithms

Many sequence analysis techniques rely on aligning sequences in the database to a model or to other sequences, often with a variant of the following edit-distance computation. Given two sequences of characters, a and b, this algorithm determines a total cost to transform one sequence into the other through three basic operations: deletion of a character, insertion of a character and mutation of a character (same as deleting a character and inserting a new character). The following dynamic programming equations are used to compute edit distance:

$$c_{i,j} = \min \begin{cases} c_{i-1,j-1} + \text{dist}(a_i, b_j) & \text{match} \\ c_{i-1,j} + \text{dist}(a_i, \phi) & \text{insert} \\ c_{i,j-1} + \text{dist}(\phi, b_j) & \text{delete,} \end{cases}$$

where $\text{dist}(a_i, b_j)$ is the cost of matching a_i to b_j, $\text{dist}(a_i, \phi)$ is the gap cost of not matching a_i to any character in b, and $\text{dist}(\phi, b_j)$ is the gap cost of not matching b_j to any character in a. Edit distance, the number of insertions or deletions required to change one sequence to another, can be calculated by setting $\text{dist}(a_i, \phi) = \text{dist}(\phi, b_j) = 1$, and $\text{dist}(a_i, b_j) = 0$ if $a_i = b_j$ and 2 otherwise. Though not often used in biology, edit distance is a common performance benchmark.

Sequence comparison using affine gap penalties, greatly preferred by bioligists, involves three interconnected recurrences of a similar form. Figure 1 shows a data flow graph for this computation. Variations and restricted forms of this recurrence are used in the classic sequence comparison methods [25, 28, 30]. In the most general form (a generalized profile or linear hidden Markov model [9, 17, 15]), all transition costs between the three states (in a run of matches, insertions, or deletions) and character costs are position-dependent. Using modulo minimization and the features of the Kestrel architecture allows us to execute the general form represented in Figure 1 in only 10 single-cycle instructions, without sacrificing programmability.

The alignment of two sequences, or a mapping of which characters in one sequence correspond to which characters in the other, is also important. The simplest means of generating a sequence alignment is, for each of the $O(n^2)$ $c_{i,j}$ values, to store the choices made during minimization. The space requirements can be reduced with a divide-and-conquer strategy [11, 24] or our new checkpoint algorithm [10].

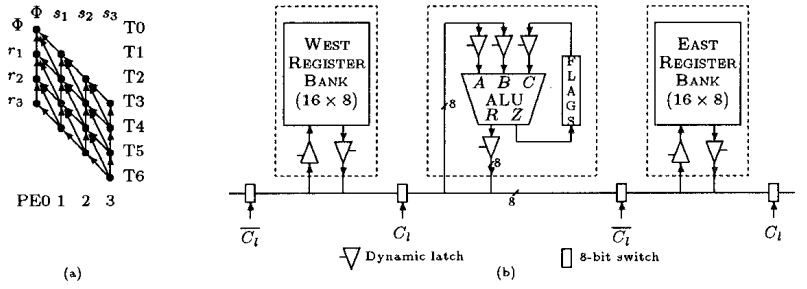

Figure 2. (a) Parallel mapping used by most fine-grain co-processors, and (b) B-SYS processing element [16].

3: Sequence Analysis Architectures

The dynamic programming calculation easily maps to a linear array of processing elements. A common mapping is to assign one PE to each character of the query string, and then to shift the database through the linear chain of PEs (Figure 2a), performing the computation in $2n - 1$ steps on n PEs. Alternate mappings have been used by BioSCAN and by P-NAC [22, 29].

A number of parallel architectures have been developed for sequence analysis. In addition to architectures specifically designed for sequence analysis, existing programmable sequential and parallel architectures have been used for solving sequence problems.

Single-purpose VLSI can provide the fastest means of running a particular algorithm with very high PE density. However, they are limited to a single algorithm, and thus cannot supply the flexibility necessary to run the variety of algorithms required for analyzing DNA, RNA, and proteins. P-NAC was the first such machine, and computed edit distance over a four-character alphabet [22]. More recent examples, better tuned to the needs of computational biology, include BioScan, BISP and Samba [29, 5, 20].

Reconfigurable systems are based on programmable logic such as field-programmable gate arrays (FPGAs) or custom-designed arrays. They are generally slower and have far lower PE densities than single-purpose VLSI but can be faster than supercomputers. They are flexible, but the configuration must be changed for each algorithm, which is generally more complicated than writing new code for a programmable architecture. PAM, Splash and Biocellerator are based on FPGAs, while MGAP and PIM have their own reconfigurable designs [2, 7, 12, 6, 3, 8].

Kestrel is an example of a programmable co-processor. The goal of programmable co-processors is to achieve flexibility while providing performance on a level with single-purpose VLSI. Kestrel is based on B-SYS, a programmable co-processor developed for sequence analysis (Figure 2b) [16, 13]. B-SYS has several shortcomings, in particular when applied to algorithms other than simple edit distance, which Kestrel overcomes. For the design of Kestrel, we examined a variety of sequence analysis algorithms and incorporated hardware within the PE to provide for fast execution of these algorithms while maintaining flexibility.

4: Kestrel Architecture

Kestrel is a linear array of 8-bit, single-instruction stream, multiple-data stream (SIMD) processing elements. Linear arrays have modest I/O considerations compared to other types of arrays because data can only enter and exit the array at the ends. The linear design provides a relatively simple method for scaling the array size up or down: chips can be removed from or added to one end of the array as required.

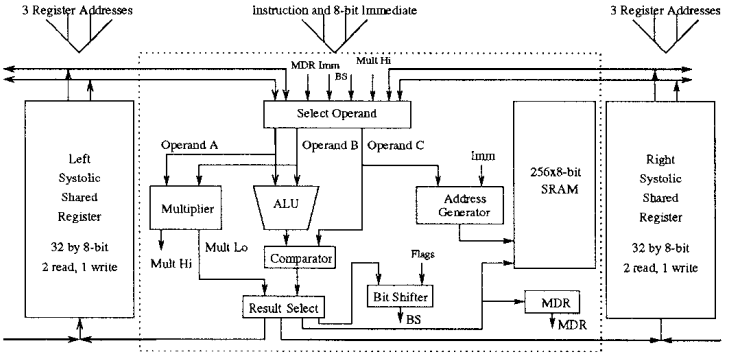

Figure 3. A Kestrel processing element with Shared Systolic Registers.

Kestrel's 8-bit datapath provides good balance between PE density and operand size. All 8-bit operations can be done in a single cycle that will be between 15 and 30 ns (depending on the process and final layout). Since each PE contains its own SRAM, limiting the width of the datapath is necessary if we want to put 64 PEs on a die of a size that will have a reasonable yield and have enough SRAM locations to handle algorithms such as sequence alignment. Fortunately, 8-bit data is sufficient for most sequence algorithms. In cases where larger data sizes are needed, Kestrel is designed to perform multi-precision operations efficiently. The 8-bit datapath also allows for easy interfacing of the array to external staging memory which will hold data to be processed and results generated from running programs.

A block diagram of the Kestrel PE is shown in Figure 3. The main components of the PE are the arithmetic-logic unit (ALU), static random-access memory (SRAM), comparator, bit shifter and multiplier, which are described in the following sections. Masking logic is also present to enable conditional execution. There are three operands for each instruction: Operand A, Operand B, and Operand C. Operand A and operand C are two independently selected registers. Operand B can come from the multiplier, the bit shifter, memory data register (MDR), the same register as Operand C or a globally-issued immediate value. In addition to these sources, Operand B can be the sign extension of Operand C, the MDR or the high byte of the multiplier result. This is useful for signed multi-precision operations where the two operands differ in length — in sequence comparison, the dist() values are commonly of lower precision than the $c_{i,j}$ values.

Depending on the instruction and flags generated in the PE, one of three values is selected as the result. The three possible results are the ALU output, Operand C or MultLo, the low-order byte from the multiplier. The result is always written to a register, and can also be written to SRAM and the bit shifter. Instruction execution takes one clock cycle.

4.1: Systolic Shared Registers

Kestrel employs Systolic Shared Registers (SSRs) for inter-PE communication (Figure 4). The SSRs are composed of 32 8-bit registers. Each SSR has two read ports and one write port, so that instructions can be executed in only a single cycle. Rather than being local to each PE, as register files usually are, SSRs reside between PEs. This allows neighboring PEs to share a register file. SSRs were invented for the B-SYS co-processor and are well-suited to linear arrays, but can be applied to any planar topology [16].

The key idea of SSRs is that communication and computation do not require distinct instructions but occur concurrently. When a result is stored after an instruction is executed, communication automatically takes place. The programmer can naturally think about data streaming through the array as values are computed. SSRs make for an elegant programming model and are a low-overhead

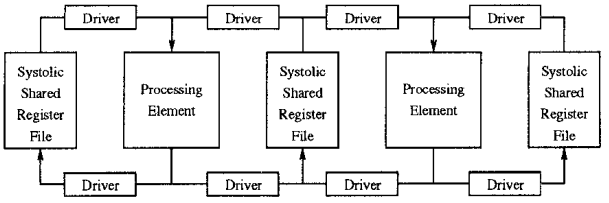

Figure 4. Structure of the Kestrel array.

solution to the inter-PE communication problem–the cost is one bit per register address.

To pass a value from one PE to the next using SSRs, all that is necessary is to store the value in a register file. For example, if each PE is calculating a value to be used by the PE to its right, then each PE stores its result into the register file on its right. During a subsequent instruction, each PE can read from its left register file to get the previously stored value. Because all addresses for these register files are issued globally, adjacent PEs will never write to the same register bank.

To avoid having three operand busses between chips, every chip in a linear SSR machine contains n PEs and $n+1$ SSRs. Thus, at chip boundaries, there are two register banks without an intervening PE. These register banks are coherent: whenever a write occurs, for example, to the left, the value is written both to the leftmost SSR of a chip and off-chip. The rightmost SSR of the adjacent chip then stores the value entering the chip. Inter-chip communication does not involve register reading.

Although similar in form to inter-PE queues, SSRs have several important differences [1, 19]. In queues, data accessed at the ends of the queue. Thus, data must be scheduled so that values appear at the head of the queue at the correct time. This is useful in multiple-instruction stream, multiple-data stream (MIMD) computers where deadlock and contention are key issues. These issues are of no concern in a SIMD machine such as Kestrel. SSRs allow PEs to read and write any location within the register file, providing more flexibility, in both synchronization and value reuse, than queues.

4.2: ALU

Kestrel PEs have an 8-bit arithmetic-logic unit composed of eight identical bit slices, each with a carry-in, a carry-out and a result bit. The carry-in to the least significant slice can come from a bit in the instruction or a latch containing a previous carry-out for multi-precision operations. The carry-out and result from each slice are determined according to the following equations:

$$\text{carry-out} = (\text{propagate AND carry-in}) \text{ OR generate}$$
$$\text{result} = \text{propagate XOR carry-in}$$

Propagate and generate are 4-bit instruction fields. One bit from each field is selected for propagate and generate for each slice according to the corresponding bits from Operand A and Operand B. This design is derived from the OM2 [23].

We chose this type of ALU for two reasons. First, it is programmable. The function of the ALU is determined by the propagate and generate fields. This ALU is capable of 256 functions, including all common logic functions and the standard arithmetic functions of addition and subtraction. The flexible design does have the disadvantage of requiring eight bits to specify the ALU function, and undoubtedly, some of the functions this ALU is capable of will not be used.

Second, the ALU is compact. In the trade-off between functionality and area, this ALU achieves its flexibility without requiring a large amount of area. For sequence analysis, operations will typically be on 8- to 24-bit values. Increasing the datapath to 24 bits would speed up higher precision operations but with a significant impact on PE density. We could have chosen to make a bit-serial design, as well. A bit-serial ALU would be compact, but would have speed limitations. The 8-bit design allows us to achieve good PE density within a chip and a clock speed of between 33

and 66 MHz. This high clock rate offsets the effects of multi-precision computation by maintaining a balance between I/O speed and computation speed.

4.3: SRAM

Each PE contains 256 bytes of static random-access memory. Each has an address generator that supports both local and global addressing. Unlike the SSRs, the SRAM contains a single bidirectional port to conserve area. A value can either be read from the SRAM or written to the SRAM once per cycle. Values read from the SRAM are stored in a memory data register (MDR) that can be accessed in subsequent instructions. The MDR can help to alleviate the potential bottleneck of a single port because a value can be read and stored in the MDR prior to use, and used several times without recomputing the address. Thus, the MDR allows us to schedule the use of the single port.

The need for large-capacity storage arises in sequence alignment, in particular. As characters from a database stream through the array to be compared to a target sequence, the costs of the necessary operations (insertion, deletion, mutation) to align the database sequence with the target sequence are calculated. In more sophisticated models, these costs are character dependent, and local addressing allows cost tables to be indexed using the character currently being examined by a PE. The minimum of these three costs is selected as $c_{i,j}$ and flags indicating which was selected are stored in SRAM. When the cost table is complete, the information in the SRAM can be used to determine the minimum-cost set of operations to align the sequences. Storage-efficient variations of this algorithm based on checkpoints enable alignment of sequences up to length $n = 30\,000$ in 256 memory locations [10].

Although the SSRs could be expanded to handle larger storage requirements, the cost in area and instruction bits would be high. The triple-port regsiters used in Kestrel take twice the area of the SRAM per bit, and since we specify three register addresses in each instruction, at least three bits would have to be added to address larger register files. Since the SRAM is dense and compact, we felt confident making the bulk of the storage a clocked static RAM. The combination of the SSRs and the SRAM provides a memory hierarchy that has high bandwidth (SSRs), large capacity (SRAM) and flexibility.

4.4: Comparator

The comparator compares the output of the ALU with operand C, and the minimum or maximum can be selected as the result of instruction execution. This is done by subtracting operand C from the output of the ALU. The subtraction produces three flags, the borrow-out from the subtraction, the most significant bit (msb) of the subtraction, and the true sign of the subtraction. The true sign is the sign that would be produced had the operands been sign-extended to nine bits, eliminating the possibility of overflow in signed comparisons. Since the machine is SIMD, an overflow bit would be of little use.

The three flags allow for three types of comparison: unsigned comparison (borrow-out), modulo 256 comparison (msb) and signed comparison (true sign). for sequence comparison . As the cost table is generated when comparing two sequences, it is possible for $c_{i,j}$ values to exceed eight bits, which would lead to erroneous results. In certain situations, modulo comparison can be used to avoid this problem [21]. The key in modulo comparison is that local differences in costs are small. With m bits of precision for table entries, if delta costs are restricted to being less than 2^{m-1}, then the msb of subtraction determines the minimum value. Use of modulo comparison enables execution of the affine-gap loop in only 10 instructions. This technique cannot be applied to all cost functions; such cases require multi-precision.

In designing Kestrel, we have paid particular attention to multi-precision operations. The comparator is an example of this. Multi-precision compares are done top-down by byte. Top-down comparison requires fewer cycles to compare two multi-precision values than the standard method of subtracting one value from the other and then selecting the result based on the borrow-out of the subtraction. When comparing two values of length n bytes, $n - 1$ cycles are saved.

Pseudo Code	Kestrel Instructions	Operation Key	
	(1)	Clear stack	(1)
if((X==Y)	ALU== and (2)	Push flag	(2)
&& Z){	ALUtest Z and (6)	Pop	(3)
if((M==N)	ALU== and (2)	Pop, invert msb	(4)
|| P)	ALUtest P and (7)	Invert msb	(5)
{inst0}	inst0 and (5)	AND into msb	(6)
else{inst1}}	inst1 and (4)	OR into msb	(7)
else{inst2}	inst2 and (1)	Replace msb	(8)

Figure 5. Kestrel condition stack operations.

It might appear that the comparator can not start calculating its result until the ALU output has been determined. If true, this would significantly increase the cycle time for an instruction and be a tremendous penalty for instructions that do not use the comparator. This is not the case, however. Each stage of the borrow chain only depends on the corresponding bits from the ALU and operand C. So, as soon as the ALU produces each bit of output, that bit is immediately used by the borrow chain. Thus, the comparator propagates concurrently with the ALU. We estimate the increase to the cycle time compared to a PE without the comparator to be around 15%.

4.5: Bit Shifter

The bit shifter is a loadable shift register that can shift left or right. Various flags from the ALU and the comparator can be shifted into either end of the shift register. Also, the result of instruction execution can be loaded into the register, and the bit shifter contents are available as an operand.

The bit shifter serves two purposes within the PE. First, it can be used for data manipulation. Flags can be shifted into either end, and the least-significant bit (lsb) and msb can be used to select the result of instruction execution. Both these operations are useful in the storage and retrieval of minimization costs in sequence alignment.

The second function is as a one-bit-wide stack machine for evaluating conditionals. Because Kestrel is SIMD, PEs must be turned off to perform conditional computation. The bit shifter aids in the fast evaluation of nested conditionals. Figure 5 shows the available condition stack operations for evaluation of conditionals. Each stack operation causes a mask local to each PE to be set based on the contents of the PEs stack. This mask indicates whether or not a PE is active. All of the stack operations can be overlapped with instruction execution, making conditional processing invisible to execution.

A pseudo-code example of conditional execution and the corresponding stack operations (shown in parentheses) appears in Figure 5. Initially, the stack is cleared, which makes all PEs active. The two if statements can be evaluated with the ALU or comparator and a flag indicating whether or not the condition was true can be pushed into the bit shifter. The else statements can be evaluated by manipulating the conditions placed on the stack during the evaluation of the ifs. Note that an else operation must be done before the code in the else block can be executed. Thus, stack operation (5) is combined with inst0 and stack operation (4) with inst1. Up to eight conditions can be evaluated before the stack becomes full. The contents of the stack can then be stored in SRAM or an SSR and more conditions put on the stack. Even though the generality of this condition mechanism is unnecessary for current sequence comparison, it costs only 4 microcode bits and will allow us to easily adapt to more sophisticated algorithms as they are developed.

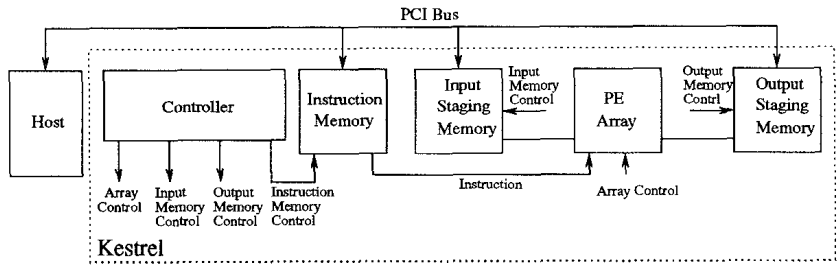

Figure 6. Kestrel system design.

4.6: Multiply-add

The multiplier performs a variety of functions on operand A and operand B to produce a 16-bit result in one cycle. The operand A and operand B values can be independently treated as signed or unsigned, enabling signed and unsigned single and multi-precision multiplies and shifts. An adder in the multiplier can be used to add a previously computed high-order byte to a newly computed low-order byte for multi-precision multiplies.

Although the multiplier is not needed for most sequence analysis applications, it does add to the general programmability of Kestrel. The multiplier provides an easy way of doing multi-bit and multi-word shifts and is useful for multiplicative division methods [4]. The multiplier provides significant speed up over the ALU for these functions with a relatively low design cost and increase in PE area of about 15%. It also allowed us to remove less-intuitive hardware for performing shifts, multiply-add and division steps.

4.7: System Architecture

The Kestrel system will consist of three main elements: the PE array, the array controller and a host interface. The basic system design is shown in Figure 6. The array controller will be responsible for issuing instructions to the PE array and for controlling communication with the host system. Since most systolic programs consist mainly of loops, simple loop control will be provided in the controller. The controller will issue instructions from a program memory of approximately 2 Kbyte instructions. Staging memories of 128 Kbyte to 1 Mbyte will be used to move data in and out of the array. Since most data will be coming from a disk, the memories should be large enough to accommodate one to two tracks of the disk to make accesses efficient. These memories will connect with the host interface and be managed by the controller. As the memories fill up or empty the controller will stop the array and request service from the host. To achieve the necessary speed to match the array, the controller will be a custom VLSI design.

Kestrel instructions are 53 bits long. Approximately half of the instruction is dedicated to specifying operands; the three registers and immediate require 26 bits. The rest of the instruction is dedicated to controlling Kestrel's many functions. The instructions are well encoded but are easy to decode because they have a fixed length and use a fixed-field encoding.

5: Performance

We have designed a simulator for Kestrel that runs on a MasPar computer. The simulator consists of a simple controller model for issuing instructions to the simulated Kestrel array. The size of the array being simulated can be controlled through a command-line switch. The simulator has proved useful for evaluating our design decisions. Our current timing estimates are based on a 33 MHz clock. Trial layouts have indicated this is feasible. The slowest parts of the design are the controller and chip-to-chip communication.

Application	Cell Rate M/s	Kestrel Instr	Instruction penalty without				16-bit Cell×2
			compare	SRAM	bit shift	mult	
Edit Distance	11.0	3	66%				6
Modulo Affine	3.0	11	54%	180%			22
16-bit Affine	1.7	19	95%	100%			12
24-bit Affine	1.1	31	58%	64%			38
24-bit Alignment	0.8	40	45%	∞	26%		56
16-bit multiplication	4.7	7				460%	2
16-bit division	0.9	38	12%			110%	16
FP multiplication	0.4	91	2%		23%	130%	

Table 1. Kestrel cell program times and extra cycles required with loss of each feature

Table 1 shows the applications we have examined so far and the effects of architectural components on running times for the applications. The toy edit distance calculation can be performed at 11 million iterations per second (with a 33 MHz clock), well above disk rates. Affine cost functions able to use modulo costs can be evaluated at disk speeds. The 16- and 24-bit affine cost instruction counts are based on a common variant that detects similarities between subsequences of two sequences [30]. These two methods require about 1 Mbyte/second disk rates. The floating-point multiply results are based on an earlier version of the simulator that did not include the multiplier, and required 208 cycles to perform 32-bit IEEE-format floating-point multiplication.

The table's final column shows the length cell programs would be in if Kestrel were 16 bits wide instead of 8, multiplied by two to reflect the lower PE density — although multiplier area quadruples, local control area would only increase slightly. There are hidden costs to the 16-bit design: 16 additional data pins to each chip, a slower clock speed, a higher I/O rate, and possibly a wider instruction field to enable 16-bit immediates. On all but the multiplication and division algorithms, a bit-serial machine would perform similarly to the 8-bit Kestrel without a comparator (with an 8× slowdown but a somewhat less than 8× increase in PE density, and possibly a higher clock rate if chip-to-chip communication could be sped up).

6: Conclusions

Kestrel strives for the middle ground of providing the speed and price of single-purpose array processors with the simple programmability of general-purpose supercomputers. The Kestrel architecture has been strongly tuned to sequence analysis, in particular its ability to add two numbers, minimize that with a third, and save both the result and the indicator bit in a single cycle. Special instructions such as this one enable Kestrel to keep reasonable pace with both disk speeds and single-purpose sequence analysis co-processors, yet still be programmable. For example, we expect Kestrel to perform 16-bit affine cost sequence comparison three times faster than a 1-million-dollar 16384-PE MasPar MP-2 [26] and 15 times faster than the fastest FPGA-based system, and and about two times slower than BioSCAN's heuristic search method on a typical query [14].

7: Acknowledgements

We gratefully acknowledge the contributions of Elizabeth Avila (FP multiplication), Leslie Grate (architecture discussions), J Alicia Grice (sequence alignment), and Eric Rice (division) to the Kestrel project. Inclusion of the bit-shifter was aided by work with Don Roberts on the MISC architecture [27]. Kestrel is funded by NSF grant MIP-9423985. More information on Kestrel can be found at `http://www.cse.ucsc.edu/research/kestrel/kestrel.html`.

References

[1] Marco Annaratone et al. The Warp computer: Architecture, implementation and performance. *IEEE Transactions on Computers*, 36(12):1523–1537, December 1987.

[2] Patrice Bertin, Didier Roncin, and Jean Vuillemin. Introduction to programmable active memories. Technical Report 3, Digital Paris Research Laboratory, Rueil Malmaison, France, June 1989.

[3] Manjit Borah, Raminder S. Bajwa, Sridhar Hannenhalli, and Mary Jane Irwin. A SIMD solution to the sequence comparison problem on the MGAP. In Peter Capello et al., editors, *ASAP*, pages 336–45, Los Alamitos, CA, August 1994. IEEE CS.

[4] Joseph J. F. Cavanagh. *Digital computer arithmetic.* McGraw-Hill Book Co., New York, 1984.

[5] E. Chow, T. Hunkapiller, J. Peterson, and M. S. Waterman. Biological information signal processor. In Matea Valero et al., editors, *ASAP*, pages 144–160, Los Alamitos, CA, September 1991. IEEE CS.

[6] Compugen Ltd. Biocellerator information package. Obtained from `compugen@datasrv.co.il`, 1994.

[7] Maya Gokhale et al. Building and using a highly parallel programmable logic array. *Computer*, 24(1):81–89, January 1991.

[8] Maya Gokhale et al. Processing in memory: The Terasys massively parallel PIM array. *Computer*, 28(4):23–31, April 1995.

[9] M. Gribskov, R. Lüthy, and D. Eisenberg. Profile analysis. *Methods in Enzymology*, 183:146–159, 1990.

[10] J Alicia Grice, Richard Hughey, and Don Speck. Parallel sequence alignment in limited space. In Christopher Rallings et al., editors, *ISMB*, pages 145–157, Menlo Park, CA, 1995. AAAI/MIT Press.

[11] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, June 1975.

[12] Dzung T. Hoang. Searching genetic databases on Splash 2. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, Los Alamitos, CA, April 1993. IEEE CS.

[13] Richard Hughey. Programming systolic arrays. In Edward Lee and Teresa Meng, editors, *ASAP*, pages 604–618, Los Alamitos, CA, August 1992. IEEE CS.

[14] Richard Hughey. Parallel sequence comparison and alignment. In Peter Capello et al., editors, *ASAP*, pages 137–140, Los Alamitos, CA, July 1995. IEEE CS.

[15] Richard Hughey and Anders Krogh. Hidden Markov models for sequence analysis: Extension and analysis of the basic method. *CABIOS*, 12(2):95–107, 1996.

[16] Richard Hughey and Daniel P. Lopresti. B-SYS: A 470-processor programmable systolic array. In Chuan-lin Wu, editor, *ICPP*, volume 1, pages 580–583, Boca Raton, FL, August 1991. CRC Press.

[17] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler. Hidden Markov models in computational biology: Applications to protein modeling. *JMB*, 235:1501–1531, February 1994.

[18] H. T. Kung. Why systolic architectures? *Computer*, pages 37–46, January 1982.

[19] H. T. Kung. Systolic communication. In K. Bromley, S. Y. Kung, and E. Swartzlander, editors, *First Systolic Arrays*, pages 695–703. IEEE CS, May 1988.

[20] Dominique Lavenier. SAMBA: Systolic accelerators for molecular biological applications. Technical Report 988, IRISA, 35042 Rennes Cedex, France, March 1996.

[21] R. J. Lipton and D. P. Lopresti. Delta transformations to simplify VLSI processor arrays for serial dynamic programming. In Kai Hwang et al., editors, *ICPP*, pages 917–920. CRC Press, August 1986.

[22] Daniel P. Lopresti. P-NAC: A systolic array for comparing nucleic acid sequences. *Computer*, 20(7):98–99, July 1987.

[23] Carver A. Mead and Lynn A. Conway. *Introduction to VLSI Systems.* Addison-Wesley, Reading, MA, 1980.

[24] Eugene W. Myer and Webb Miller. Optimal alignments in linear space. *CABIOS*, 4(1):11–17, 1988.

[25] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *JMB*, 48:443–453, 1970.

[26] John R. Nickolls. The design of the Maspar MP-1: A cost effective massively parallel computer. In *COMPCON Spring 1990*, pages 25–28, Los Alamitos, CA, February 1990. IEEE Computer Society Press.

[27] James D. Roberts. *MISC: A parallel architecture for AI.* PhD thesis, UC, Santa Cruz, CA, 1995.

[28] P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, 26:787–793, 1974.

[29] Raj Singh et al. A scalable systolic multiprocessor system for biosequence similarity analysis. In Lawrence Snyder, editor, *Symp. Integrated Systems*, pages 169–181, Cambridge, MA, April 1993. MIT Press.

[30] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *JMB*, 147:195–197, 1981.