Finding the next computational model: Experience with the UCSC Kestrel

Richard Hughey   Andrea Di Blas
Department of Computer Engineering
University of California, Santa Cruz, CA 95064
{rph, andrea}@soe.ucsc.edu

To appear in Journal of VLSI Signal Processing Systems, 2007.
The original publication is available at www.springerlink.com

June 28, 2007

Abstract

Architects and industry have been searching for the next durable computational model, the next step beyond the standard CPU. Graphics co-processors, though ubiquitous and powerful, can only be effectively used on a limited range of stream-based applications. The UCSC Kestrel parallel processor is part of a continuum of parallel processing architectures, stretching from the application-specific through the application-specialized to the application-unspecific. Kestrel combines an ALU, multiplier, and local memory, with Systolic Shared Registers for seamless merging of communication and computation, and an innovative condition stack for rapid conditionals. The result has been a readily programmable and efficient co-processor for a wide range of applications, including biological sequence analysis, image processing, and irregular problems. Experience with Kestrel indicates that programmable systolic processing, and its natural combination with the Single Instruction-Multiple Data (SIMD) parallel architecture, is the most powerful, flexible, and power-efficient computational model available for large group of applications.

Unlike other approaches that try to displace or replace the standard serial processor, our model recognizes that the expansion in the application landscape and performance requirements simply imply that the most efficient solution is the combination of more than one type of processor. We propose a model in which the CPU and the GPU are complemented by "the third big chip", a massively-parallel SIMD processor.

1 Introduction

Array processors and specialized hardware have long been used to accelerate some of the most computationally demanding algorithms and applications. As the march of technology improves performance and cost, these specialized processors may become ubiquitous in computing systems. Floating-point chips made this transition into the central processing unit (CPU) some time ago, and graphics processing units (GPUs) are working toward this status. Massively-parallel SIMD array processors are a candidate for a similar transition, in support of another wide range of algorithms and applications.

In this paper, we discuss the foundations and architecture of the University of California Santa Cruz (UCSC) Kestrel parallel processor. While the system has proved to be an application-unspecific parallel processor, its design and philosophy is rooted in the domain of biological sequence analysis.

Biological sequence analysis has long been a standard problem for application-specific processing and all other forms of high-performance computing. The underlying algorithms are simple and regular, and the amount of data for analysis is prodigious. University and commercial projects have tackled this problem since the early days of VLSI, and many of these approaches are chronicled within the proceedings of Application-Specific Arrays, Processors and Systems (ASAP) conference and its predecessors.

The first such machine was Lopresti's 1986 Princeton Nucleic Acid Comparator (P-NAC) [27, 28], a single-purpose systolic array that provided great speedup on its algorithm, and set the stage for the many application-specific VLSI and FPGA machines that followed.

The second wave included the use of single-chip (MICSMACS) or single-board (Warp) processors to create short systolic arrays designed for signal processing but suitable for sequence analysis and other applications [11, 2]. This period also introduced three more specialized machines with moderate to extreme numbers of processing elements (PEs) on each chip. BISP (16 PEs/chip) and BioSCAN (892 PEs/chip) were application-specific, each suited to one specific algorithm [4, 36]. This wave also saw the development of P-NAC's successor, the Brown Systolic Array (B-SYS, 47 PEs/chip). B-SYS incorporated full programmability as an *application specialized* processor, while maintaining a simple processing element design [19, 20, 15].

Flexibility and performance continued to increase with new machines from research projects and industry. The introduction of FPGA-based computing [43, 12] led to their use as sequence analysis engines [12, 5, 42, 24]. More general custom VLSI solutions also appeared [41], and the MasPar mini-supercomputer [31] became a common tool for bioinformatics research. The UCSC Kestrel architecture sought to build upon these designs while also preserving the B-SYS system's simple and efficient SIMD programming model and high computational density. The result was a movement from the application-specialized B-SYS machine to the application-unspecific Kestrel machine, able to solve a far broader selection of problems than biological sequence analysis [16, 17, 14, 33, 8, 7].

The following sections describe the architectural and programming features that lead to Kestrel's efficiency, moving from biological sequence analysis, to B-SYS, and then to Kestrel. Many of the features that made Kestrel successful reflect the values of good design: simplicity, regularity, and attention to detail. We conclude with an evaluation of the place of application-*unspecific* SIMD processing within the context of FPGA-based computation, multi-cores, graphics cards, and application-specific processing.

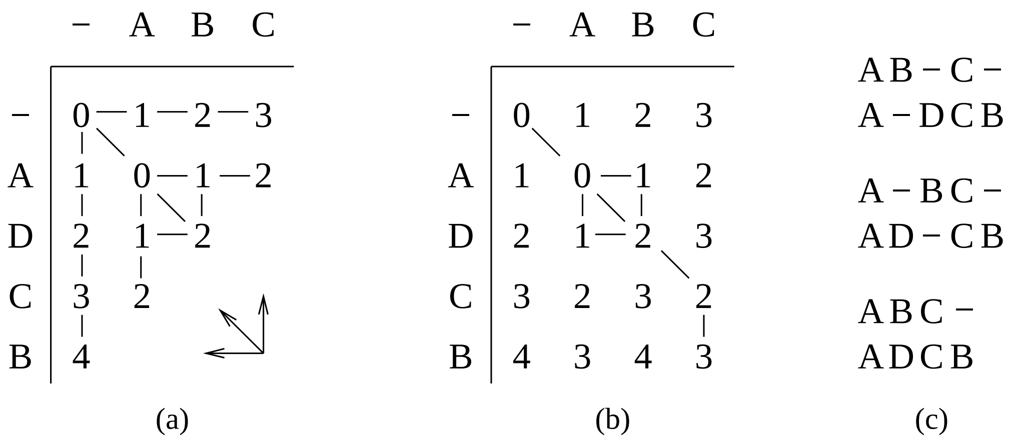


Figure 1: Dynamic programming sequence comparison, including (a) filling in the matrix and storing choices, (b) the complete matrix and best paths, and (c) the three optimal alignments.

2 Biological Sequence Analysis

A core problem of bioinformatics is determining the relationship between molecules such as DNA, RNA, and proteins. Ideally, the biologist wishes to discover the physical relationship between the molecules (do they fold up the same way? are important residues conserved?), but often must rely on just the information relationship.

One of the best ways to discover the information relationship between two sequences $A = a_1 \ldots a_m$ and $B = b_1 \ldots b_n$ is by using one of a family of related dynamic programming algorithms for pairwise sequence comparison. For two sequences, the correspondence is determined with a three-state model. The cost $c_{i,j}$ of aligning $a_1 \ldots a_i$ to $b_1 \ldots b_j$ is calculated by taking the minimum of three alternatives: (1) the two characters correspond to each other, in which case a match cost is added to $c_{i-1,j-1}$; (2) the b_j character is not present in A, in which case a deletion cost is added to $c_{i,j-1}$; or (3) the a_i character is not present in B, in which case an insertion cost is added $c_{i-1,j}$:

$$c_{i,j} \quad = \quad \min \begin{cases} c_{i-1,\,j-1} & + \quad cost(\text{match/mutate } a_i, b_j) \\ c_{i,\,j-1} & + \quad cost(\text{delete}) \\ c_{i-1,\,j} & + \quad cost(\text{insert}). \end{cases}$$

The complete algorithm consists of calculating all of the $c_{i,j}$ values in a two-dimensional dynamic programming matrix starting from $c_{0,0}$ and working toward $c_{m,n}$ (Figure 1a, with an insert or delete cost of 1, match cost of 0, and mutate cost of 2). Each element is calculated based on three adjacent cells, leading to the three data dependencies. If an alignment is being created, an indicator of at least one of the minimizing choices should be saved (the lines in Figure 1a). The elements along any diagonal, such as the last finished one in Figure 1a may be calculated in parallel, since the data dependencies are only on values in the previous two diagonals. Once the calculation is complete, the minimizing choices leading back from the final $c_{m,n}$ score indicate an optimal alignment (the lines in Figure 1b). In this example, there are three optimal alignments, all with a cost of 3 according to this distance measure. The top alignment in Figure 1c has a deletion, insertion, and insertion; the middle has an insertion, deletion, and insertion; and the bottom has a mutation and insertion.

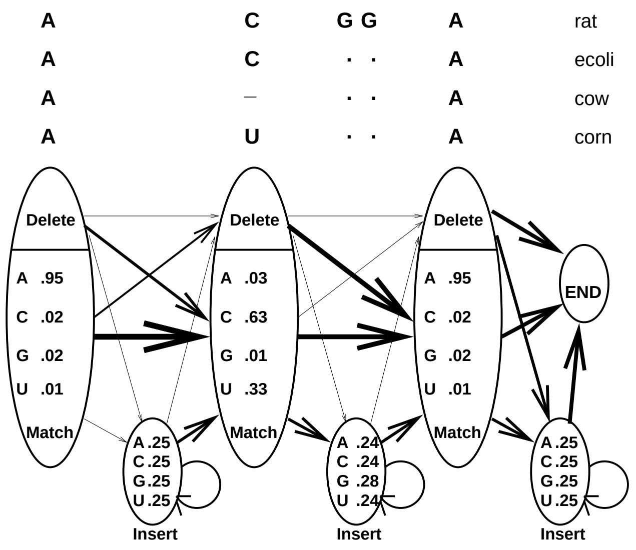


Figure 2: Hidden Markov model and corresponding alignment of three sequences.

In practice, biologists use a more complicated form called the Smith-Waterman algorithm [30, 37], which involves three interleaved recurrence equations as well as gap initiation and continuation costs. The costs are centered around zero, in the example below with negative numbers showing probable correspondence. By thresholding the calculation with 0, and finding the best (most negative) score in the dynamic programming matrix, the highest scoring subsequence-subsequence region between the two sequences may be found:

$$c^M_{i,j} = \min\left(0, \min(c^M_{i-1,j-1}, c^D_{i-1,j-1}, c^I_{i-1,j-1}) + \quad \text{cost}(a_i, b_j)\right)$$

$$c^I_{i,j} = \min\left(c^I_{i-1,j} + c, c^M_{i-1,j} + g\right)$$

$$c^D_{i,j} = \min\left(c^D_{i,j-1} + c, c^D_{i,j-1} + g\right).$$

For the most exacting searches, bioinformatics practitioners use profile hidden Markov models (HMMs) [25, 9], which enable comparison and alignment of a single sequence to a family of sequences. HMMs also use three interleaved recurrences, but with higher precision and more complicated arithmetic.

$$c^M_{i,j} = \min\left(c^M_{i-1,j-1} + g_j^{M\to M}\ c^I_{i-1,j-1} + g_j^{I\to M}\ c^D_{i-1,j-1} + g_j^{D\to M}\right) + \ \text{cost}(a_i, b_j),$$

$$c^I_{i,j} = \min\left(c^M_{i-1,j} \quad + g_j^{M\to I}\ \ c^I_{i-1,j} \quad + g_j^{I\to I}\ \ c^D_{i-1,j} \quad + g_j^{D\to I}\right) \ + \quad \text{cost}(a_i, \phi),$$

$$c^D_{i,j} = \min\left(c^M_{i,j-1} \quad + g_j^{M\to D}\ c^I_{i,j-1} \quad + g_j^{I\to D}\ c^D_{i,j-1} \quad + g_j^{D\to D}\right) + \ \text{cost}(\phi, b_j).$$

A profile HMM (Figure 2) can be thought of as a generative model for a family of aligned sequences. Each state of the HMM corresponds to a column of the alignment, and contains a probability distribution over the amino acids or nucleotides. These probabilities, as well as those for

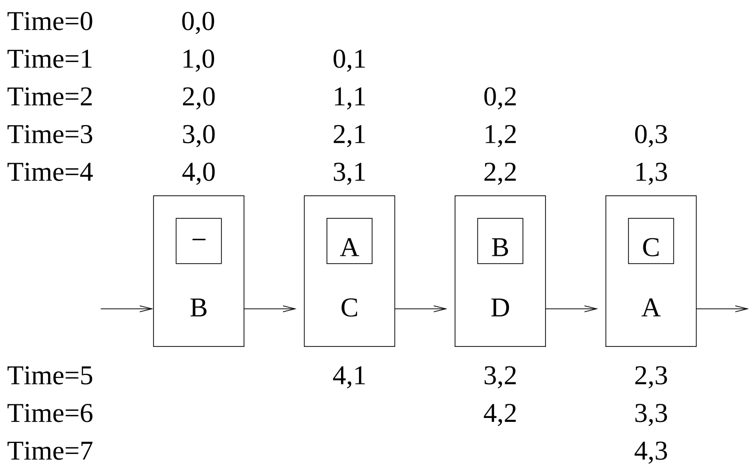


Figure 3: Index points calculated at each time step of the parallel dynamic programming algorithm. The example array, with one fixed sequence and one moving sequence, is shown at time step 4.

insertions and deletions, can be trained from a set of sequences in an iterative process that includes repeatedly evaluating the HMM version of the dynamic programming calculation. In the dynamic programming calculation, the HMM replaces one of the sequences so that, for example, each column of the dynamic programming matrix corresponds to one state in the profile HMM.

An alternate version of this algorithm, the forward algorithm, sums and multiplies probabilities, rather than maximizing and adding scores. This algorithm produces significantly better results for sequence alignment, discrimination, and HMM training [25].

These $O(n^2)$ algorithms have many mappings to linear arrays. For programmable machines, one of the best mappings is to preload one sequence (or the HMM) into the array, and then move the second sequence (or database) through the array from left to right. The dynamic programming matrix is thus calculated along diagonals, with $c_{i,j}$ calculated at step $i + j$ in PE$_i$ using values just calculated in PE$_i$ and the adjacent PE$_{i-1}$ (Figure 3).

Because of the vast size of genomic and protein databases, many universities and companies have worked to accelerate the sequence comparison algorithms [16]. P-NAC, the first application-specific solution, placed 30 PEs on a chip, and implemented the above equation over a four-character (DNA or RNA) alphabet, with an insert or delete cost of 1 [27, 28]. This simplicity enabled fast dynamic programming, critical for analyzing large sequence databases, but the lack of support for Smith-Waterman and other algorithms reduced the utility for biologists.

3 The Brown Systolic Array

The goal of the Brown Systolic Array (B-SYS) was to develop a successor co-processor to P-NAC. Maintaining simplicity similar to that of P-NAC was critical to ensure that many PEs could be placed on a single die, enabling large arrays to be created with only a handful of chips. It was also important to introduce programmability into the system. The family of sequence comparison algorithms is broad, and all or most of the many variations can only be accelerated on a programmable machine.

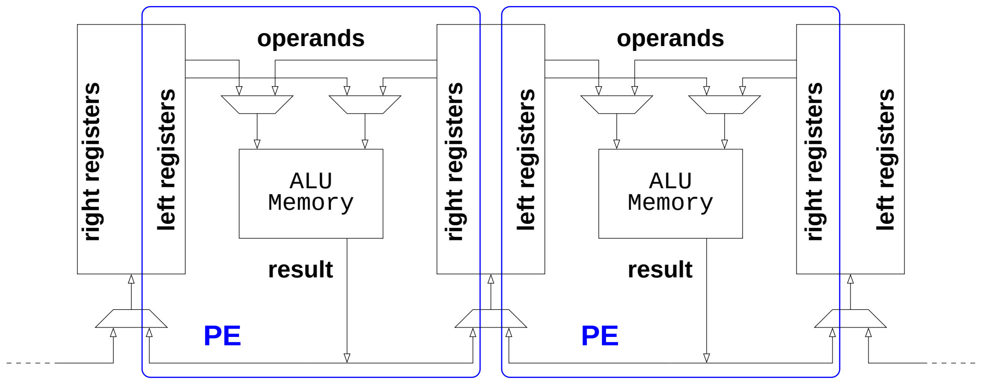


Figure 4: Systolic Shared Registers

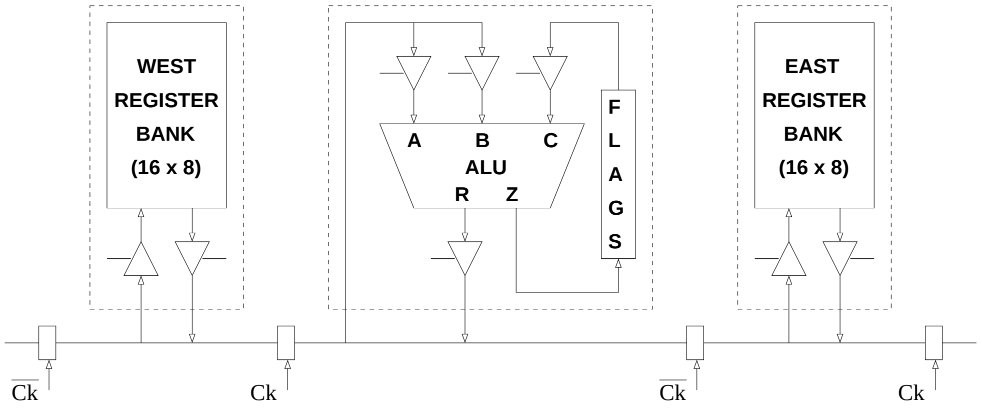


Figure 5: The B-SYS ALU with two flanking Systolic Shared Registers

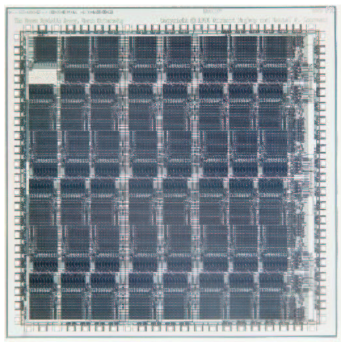


Figure 6: The B-SYS chip with 47 processing elements.

```
  // The first opcode (e.g., xorABC) determines the result
  // (stored in LF) based on two inputs (L2, R2) and the carry.
  // The second opcode (e.g., Zsub) determines the operation of the
  // carry chain, as well as the input and output flag registers.
  // F7 is preset to 0, and F6 is preset to 1.  The next iteration
  // of this loop begins by comparing previous costs stored in L4
  // and R4, placing the result in R2 rather than R4.

  // Compare two previous costs (L2, R2)
xorABC      L2 R2 LF Zsub    F7 F1
  // But do the comparison mod 256
fnA         LF LF LF Zmsb    F1 F1
  // Select the minimum, place in LF
selectABonC L2 R2 LF Zconst  F1 F1
  // Add 1, the insertion or deletion cost, to LF
xorAC       LF LF LF Zadda   F6 F1
  // Compare the characters in L1 and R0, passing the L1
  // character to the right,  feeding a character to the array
fnA         L1 R0 R1 matchAB F7 F2 load(*s++)
  // If the characters matched, use L4 as the cost,
  // otherwise use LF. Store result in R4
selectABonC L4 LF R4 Zconst  F2 F2
```

Figure 7: B-SYS code for calculating edit distance mod-256 (as implemented in hardware by P-NAC).

The natural choice for B-SYS was a linear systolic array, the most efficient architecture for sequence analysis. The linear array can be placed in low pin-count packages and is tremendously scalable. Of course, the cost of that scalability is ever increasing latencies from one end of the array to the other. As the Kestrel project was to demonstrate, the linear array is far more flexible as a computation engine than might be generally thought.

B-SYS introduced the concept of Systolic Shared Registers (SSRs, Figure 4), in which register banks are shared between adjacent PEs. The SSRs enable seamless integration of computation and communication, as a single instruction can specify that values should be read from the left register bank, computed upon, and written to the right register bank. On a SIMD machine, there will be no bank conflicts because every PE is reading the same operand registers and writing the same destination register. The cost of SSRs is one instruction bit per operand and one extra register bank per chip to ensure all operands are on chip.

B-SYS (Figures 5 and 6) consisted of 10 custom chips, each with 47 processing elements implemented with 85,000 transistors [20]. Each processing element had a flexible 8-bit arithmetic-logic unit (ALU) loosely based on the OM-1 [29], 1 mask flag, 7 general-purpose flags, and sixteen 8-bit registers. A 38-bit instruction is broadcast each clock cycle. (The first design presented to ASAP proposed systolic, rather than broadcast, instructions [19].) Due to the lack of an on-board instruction memory and controller, the test system only ran at 250 kHz on a 25 MHz host, although the chips could run 10 times faster.

The SSRs enabled implementation of P-NAC's inner loop in 6 instructions (6 clock cycles) by, for example, shifting a character from left to right at the same time as comparing it to the stored character (Figure 7). With a non-programmable architecture, P-NAC implemented this loop in two clock phases. On a programmable machine without SSRs, two or more additional instructions would be necesary to explicitly move data between processing elements prior to each inner loop calculation. As sequence analysis performance corresponds directly to the length of the inner loop, this was a particularly notable achievement. Even at 250 kHz, the single-board, 470-PE B-SYS system performed sequence comparison at about the same speed as the much larger 16 K PE Thinking Machines CM-2 [20]. At the time, somewhat faster solutions included the first FPGA-based sequence comparison solution on the Splash machine with 32 Xilinx chips [12], and two single-purpose VLSI systems on much larger chips [36, 4].

B-SYS included a unified software environment with both assembly language and a higher-level language, the New Systolic Language (NSL). NSL used a stream programming model for systolic algorithms, joining the I/O characteristics of a stream, such as a sequence, with the parallel stream variables. The language overloaded C++ operators for parallel computation, performed rudimentary code optimization, and managed file I/O between the host and array or simulator [15].

The B-SYS experience provided several lessons:

- Design the full system, not just the chips. The instruction and data bottlenecks between the host machine cost a factor of 10 in performance.
- Know your algorithms. B-SYS was designed for DNA sequence comparison with its four-character alphabet. Protein sequence comparison involves a 20-character alphabet, and requires lookup tables in each PE. Although B-SYS' programmability enabled the clustering of PEs to implement protein analysis, the clustering had considerable overhead, retarding performance.
- Build *application-specialized* processors rather than application-specific. In addition to one dozen sequence comparison variations (with inner loops of 6 to 54 instructions), B-SYS was also quickly programmed for classic algorithms such as sorting and Horner's method, and other problems as well.
- Use simple architectures to enable dense implementations. The great advantage of the SIMD architecture is the lack of local instruction sequencing or decoding. With common register addressing and ALU control, instructions can be latched and decoded once per physical row of PEs within the chip.
- Include more memory per processing element. A recurring theme of first-generation system design. The SSRs were just 40% of B-SYS PE area, with the ALU, flags, and (minimal) local control using the rest. This is an inefficient ratio when grouping is used to expand memory per logical PE.

4 The UCSC Kestrel Parallel Processor

Kestrel grew from three converging experiences: B-SYS, the growing bioinformatics efforts at the University of California Santa Cruz (UCSC), and experience with a MasPar parallel computer [14, 7, 16].

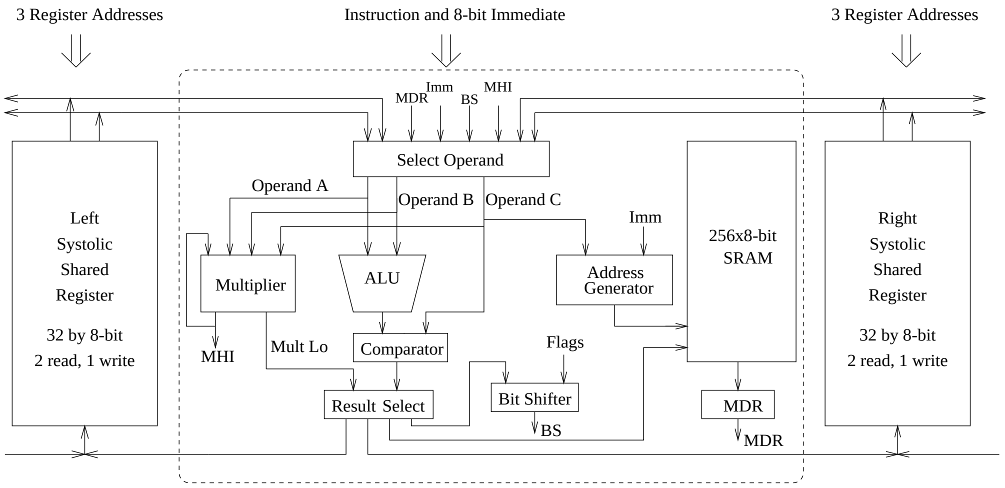


Figure 8: Kestrel processing element architecture.

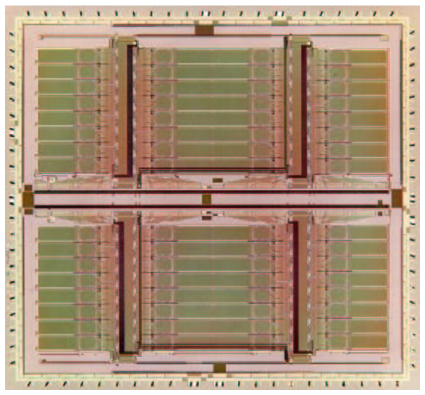


Figure 9: The Kestrel chip with 64 processing elements.

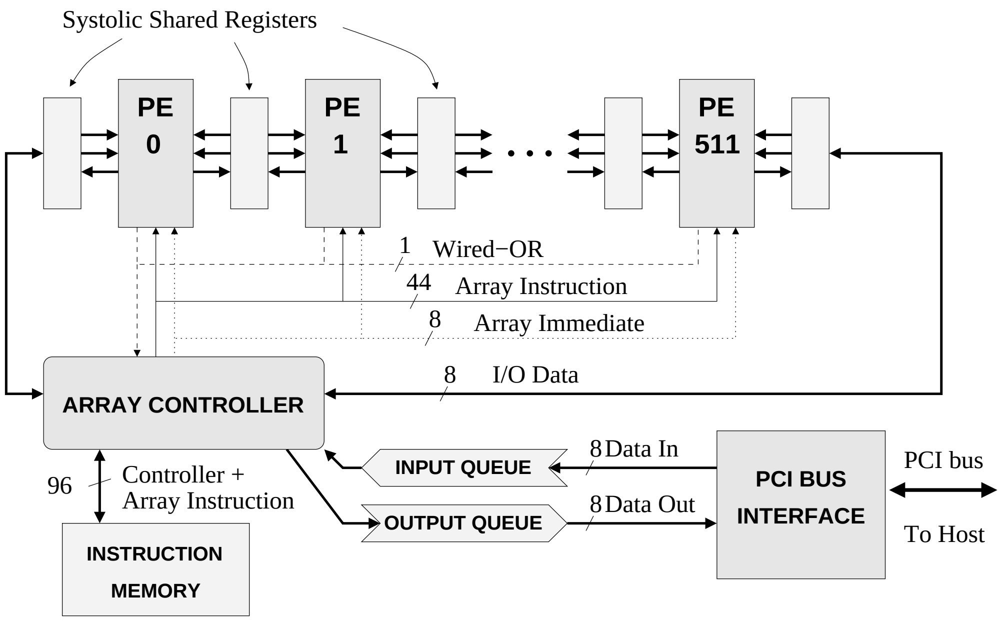


Figure 10: Kestrel board architecture

In the 1990s, UCSC pioneered the use of HMMs for sequence analysis [25]. The technique is now standard throughout bioinformatics, and is a core component of multiple alignment and protein structure prediction algorithms [23]. While scoring against a database of HMMs is similar to performing Smith-Waterman (though slower due to the more complicated algorithm), creating an HMM is a time-consuming iterative process, adding another factor of 50 to 100 [39]. For these reasons, we implemented the HMM algorithms on a MasPar parallel computer [18].

The MasPar MP-2 was a 32-bit SIMD machine with local memory addressing, a mesh connection, and a global router [31]. Instructions were microcoded, with basic operations requiring 4–40, or more, cycles. The system included 32 PEs per chip, and 1K PEs per (large) board, in configurations up to 16K PEs. While the MasPar worked well as a somewhat specialized supercomputer, it was much larger and more complicated than necessary, and in some ways inefficient, for the family of sequence analysis algorithms.

Kestrel grew out of a desire to support bioinformatics algorithms impossible on B-SYS (with its lack of local memory) and more efficiently than the MasPar.

Kestrel (Figures 8, 9 , and 10) maintains the linear Systolic Shared Register (SSR) architecture and the 8-bit word of B-SYS. We determined that 8 bits continued to be an appropriate mix of flexibility for performing 8, 16, 24, and 32-bit sequence analysis problems, and bitwise parallelism.

Conditional processing is a primary limitation of the SIMD architecture. Because the same instruction is broadcast to all processing elements, “if. . . else” clauses require processing element masking. Many SIMD machines (MPP, B-SYS, CM-1, MasPar, Kestrel) have 1-bit mask flag registers to indicate whether or not the PE should execute conditional instructions. While an “else” may be performed in one instruction (negating the flag), other common operations for conditional execution, especially nested conditionals, requires storage in PE memory and additional instructions.

One of the most innovative aspects of Kestrel is each PE’s eight-bit condition stack. The con-

Function	Use
Clear	Erase all conditions, activate all PEs.
Push	Begin an `if` clause, adding a condition to the stack.
AND TOS	Replace the top of stack (TOS, the bit shifter's msb) with the AND of the TOS and another bit. Evaluate the second part of an `if (X | Y)` clause. The AND is used to evaluate OR clauses because bits are asserted low in the bit shifter.
OR TOS	Evaluate the second part of an `if (X & Y)` clause.
Complement TOS	Invert the current condition for an `else` clause.
Pop	Complete a condition, restoring a previous state.
Pop and Complement TOS	Remove one layer of nesting and start an `else` clause, as in `if (X) { if (Y) {}} else`.
Replace TOS	Complete one condition and start another at the same nesting level, as in `if (X) { } if (Y)`.
Store	Save current state. Free bit shifter for other tasks. Can be used with Load, Clear, and Set to process more than 8 nested conditions.
Load	Restore a previous state.

Table 1: Condition stack functions associated with processing nested conditionals [6].

dition stack is an 8-bit hardware stack that also supports parallel reads and writes. The condition stack performs the basic bit operations necessary for conditional processing in parallel with other PE operations, requiring zero additional instructions. Beyond pushing, popping, and complementing bits, it can combine conditions with logical operations and perform such operations as 'pop and compliment', used when exiting a nested 'if' into an 'else' clause (Table 1). The condition stack gives Kestrel the most rapid PE activity switching of, to our knowledge, any SIMD array ever designed.

In addition to the condition stack, Kestrel adds several distinctive features to the B-SYS linear architecture with shared systolic registers (Figure 8):

- Multiplier. Multiplication was required for implementation of the HMM algorithms. The multiplier was key to making Kestrel a general-purpose systolic array, and its multiply-accumulate-accumulate operation more than doubled the speed of 32-bit multiplies.

- Local memory. In addition to the now 32 SSRs, each PE has 256 bytes of locally-addressed memory. The local addressing is critical for protein and HMM sequence analysis, and of course has many other uses.

- ALU/Comparator. Geared toward the sequence analysis applications, Kestrel includes an integrated ALU and comparator, able to perform single-cycle addition and minimization. This feature motivated Kestrel's three-operand design. While important for Smith-Waterman performance, this feature is not a requirement for general-purpose systolic computing.

- Multiprecision operation. Special care was taken in the ALU, comparator, and multiplier to ensure efficient multi-byte operation.

```
;; 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 11: Core assembly code for 16-bit Kestrel Smith-Waterman.

The system architecture (Figure 10) includes an array controller implemented on a field-programmable gate array (FPGA), on-board instruction memory, data input and output queues, a PCI bus interface chip, and clock circuitry (not shown). Every 96-bit Kestrel instruction includes 54 bits of instruction for the Kestrel PEs, and 42 bits for the array controller.

In a parallel processor, it is critical to keep the PE array busy performing useful calculations. Kestrel achieved this goal at several levels. The orthogonal instruction set enables every broadcast instruction to access the memory with indirect addressing, modify the condition stack, perform an ALU/Comparator or Multiplier operation, and, via the SSRs, move data through the array. The single-clock execution of an instruction is well-balanced to the broadcast time of the instruction and decoded register addresses. At the same time, the Kestrel controller may be passing data back and forth to the array, evaluating loop conditions based on a counter or wired-or, or branching.

These many levels of parallelism enable the core Smith-Waterman function on 16-bit numbers to be performed in only 18 instructions or 18 clock cycles (Figure 11). Again, ample use is made of the zero-instruction communication provided by the Systolic Shared Registers. Compared to a 500 MHz UltraSparc-II, Kestrel achieved speedups of 100 for Smith-Waterman, 44 for Viterbi (add/min) HMM scoring, and 8 for Forward (mult/add) HMM scoring [7].

Kestrel also does very well against general-purpose, special-purpose, and FPGA processors. It is difficult to compare machines across technologies and budgets. Newer technologies boast higher clock speeds, and larger budgets lead to larger chips and systems. One way of comparing sequence analysis performance is in terms of performance per transistor [7]. Although this does not correct for clock speed increases between CMOS generations, it does correct for scaling. Kestrel's 1997 0.5μm chips at 20 MHz, half their achievable speed, are 360 times more efficient on protein sequence analysis than the 6-year-earlier MasPar MP-2 (1992, 1.0μm at 12.5 MHz). Kestrel's 20 MHz efficiency is also better than 3 machines from the next two CMOS generations. These include a very large general-purpose VLSI processor (Fuzion 150, 2000, 0.25 μm, 200 MHz [35]), a commercial FPGA sequence analysis machine (DeCypher, 2001, 0.18 μm? [42]), and a commercial sequence analysis ASIC (GeneMatcher2, 2001, 0.13μm? 192 MHz, [32]) [7].

The Kestrel project provided several lessons, some of which echo those learned with the earlier

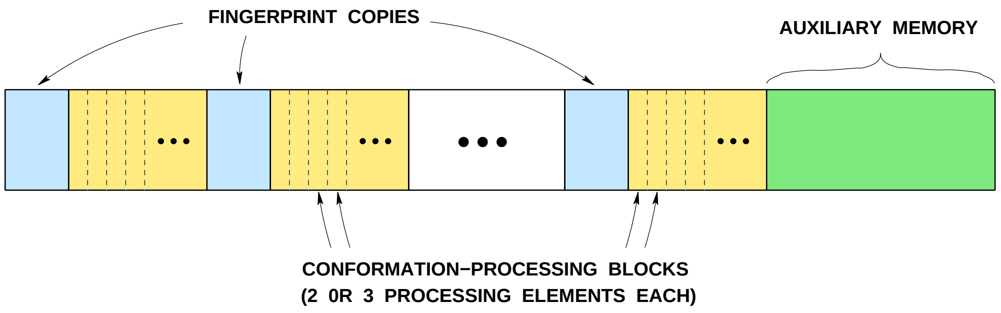

FINGERPRINT COPIES
AUXILIARY MEMORY
CONFORMATION–PROCESSING BLOCKS
(2 0R 3 PROCESSING ELEMENTS EACH)

Figure 12: Grouping processing elements in the chemistry application.

system:

- Spend as much time designing the full system as the chips. The Kestrel 1 controller, now on board with the instruction memory, was quickly prototyped as a single-cycle sequencer. Unfortunately, we did not originally design a pipelined board and controller, which would have enabled the Kestrel chips to function at 40 MHz, twice as fast as the Kestrel 1 system.

- Create new algorithms. One of the most amazing aspects of the Kestrel project has been the variety of algorithms, many not obviously parallelizable on a linear array. While high Kestrel performance was expected on sequence analysis, Kestrel's successful application to graph problems, asynchronous algorithms, floating-point arithmetic, machine learning, and conformation analysis was a surprise.

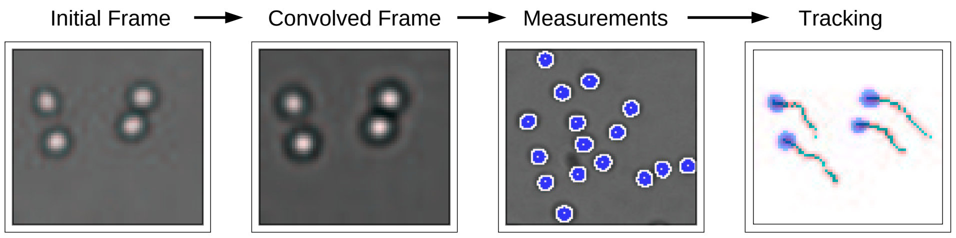
- Compilers are important but hard to do well. For several years, we worked to create a full-fledged compiler for Kestrel. The limited local memory, reducing the ability to spill and retrieve registers, made the implementation difficult. Although able to generate code, we never used it to program the array.

- Include more memory. A recurring theme of second-generation system design as well. In Kestrel's case, the primary memory issues are the need for on-board memory and a potential increase in the local memory. Unlike B-SYS, however, conditional execution and processor grouping is sufficiently efficient so that this is not a major problem.

5 Application-*Unspecific* Processing

The most important feature of Kestrel is its flexibility. We have programmed Kestrel for computational chemistry, protein conformation analysis, neural networks, floating-point arithmetic, high-speed division, and a variety of image processing algorithms [7, 33].

The computational chemistry application is particularly interesting for its use of PE grouping. In this problem, Kestrel analyzes all possible conformations of a small molecule to create a 14 kbit geometric fingerprint [33]. Each fingerprint bit indicates the presence or absence of a particular set of three atom types forming a triangle with specific length ranges between the atoms. The parallel algorithm includes loading conformations into the array, computing pairwise distances within each conformation, and using sets of three pairwise distances and three atom types to determine which



	Time / frame	Frames/s	Speedup
MATLAB	53 s (60% conv.)	0.02	1.0
C	0.96 s (92% conv.)	1.04	55.2
C + Kestrel	0.72 s (90% conv.)	1.38	73.6
C + Kestrel−2	87 ms (19% conv.)	11.50	609

Figure 13: Four steps in our cell-tracking program (shown on a small portion of the actual image) and a summary of the performance with the parallel fraction of the computation, which also includes the bi-linear interpolation.

fingerprint bits to assert. The results for the many conformations are combined, and the final fingerprint is sent back to the host.

In the Kestrel mapping, blocks of three PEs are used to process each conformation. For every eight blocks, we allocated an additional eight PEs for storing local copies of the developing fingerprint (the 8 PEs have a total of 16 kbit of local memory), so that values could be accumulated within the array close to their calculation rather than being shifted out of the array after each completed calculation. Similarly, because of the lack of data memory on the Kestrel board, another 196 of the 512 PEs were used as auxiliary memory, storing atom type and conformation data as groups of conformations are processed (Figure 12). This application highlights that small and simple SIMD PEs can be effectively grouped to tackle large problems.

Most recently, we created a Kestrel application for the object recognition and tracking of red blood cells in microscope images (Figure 13) [22]. The application also enhances the video resolution to be able to accurately measure the cells' parameters needed for the experiment — area, perimeter, and sphericity. Finally, the goal was to monitor each cell's parameters over time to study the cells' reactivity. This was done by tracking all cells across frames in a video stream. Our implementation has a C front-end interfacing with Kestrel for the most data-intensive processing. These include a bi-linear interpolation to increase the resolution of the image, and subsequently performs a bi-dimensional convolution using a cell image as the kernel. Both these operations are computationally- and data-intensive, as well as embarrassingly parallel, and map efficiently on our architecture, while on a standard CPU they stress the cache performance.

Our speedup over the original Matlab-based implementation brought processing time down from several hours to a few tens of seconds for a typical video, as was expected. However, it was still far from real-time processing. We have shown that with our next processor board generation with twice as many of the original 64-PE chips and a pipelined controller enabling a doubling of clock speed, Kestrel 2 (Figure 14), the speedup would enable almost-real-time processing [22].

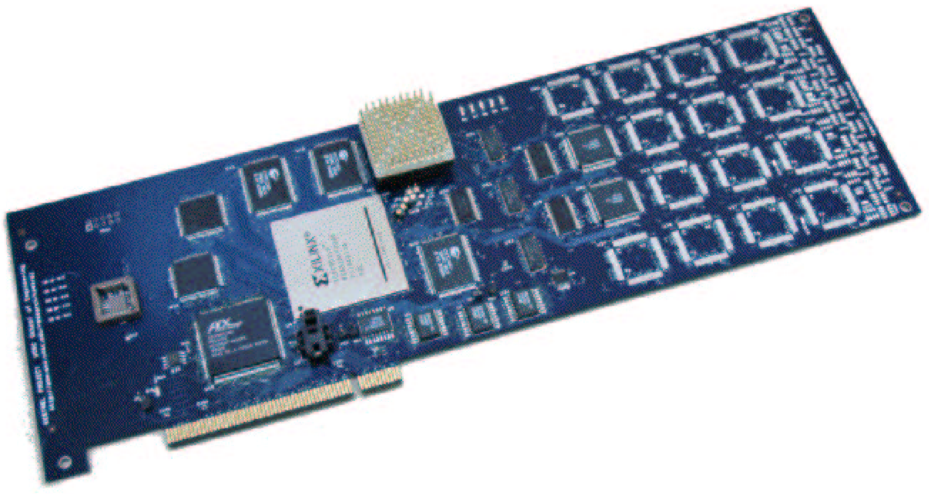


Figure 14: The Kestrel 2 board.

We also developed a paradigm of phased programming for implementing asynchronous algorithms on SIMD arrays [8]. This programming model, the *SIMD Phase Programming Model* (SPPM), provides a clear methodology to implement asynchronous, irregular problems on SIMD architectures, extending the conventional application range of the architecture. SPPM is suitable for assembler or compiler support, though in the latter case, on Kestrel, would run into the same register allocation issues discussed above. This method starts from a sequential implementation of the algorithm to parallelize, and has five steps:

1. *Problem partitioning and data flow definition*, which includes all four steps of Foster's design methodology: Partitioning, communication, agglomeration, and mapping [10].

2. *Control-flow transformation*, in which the control flow of the partitioned program is explicitly modified to implement all flow control structures using only conditional branching (if/else) and unconditional jumps (goto).

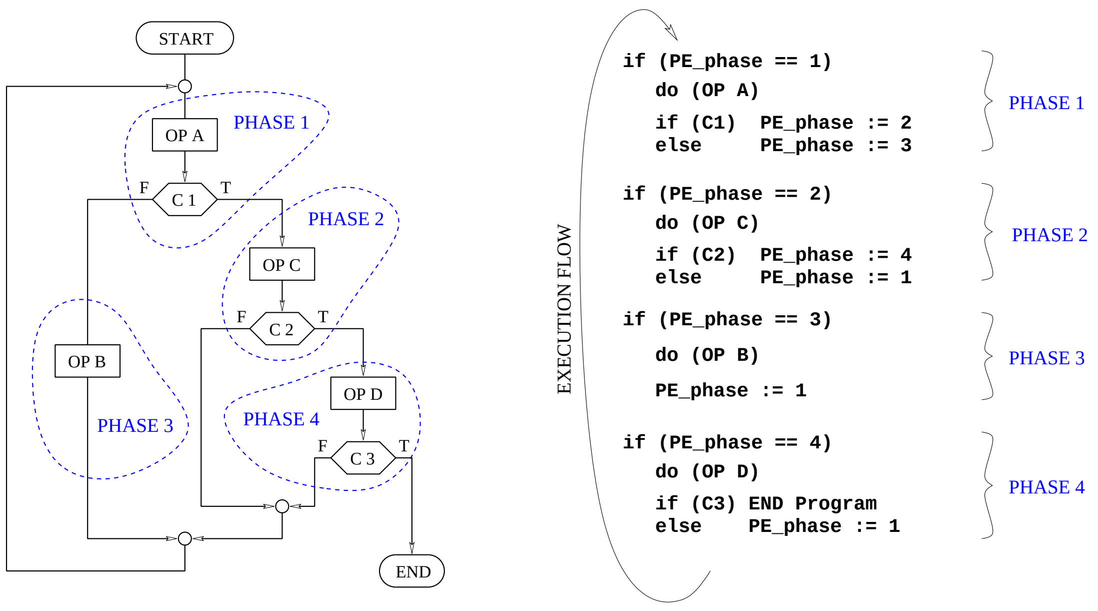
3. *Single-PE sequential program*, in which the program is mapped to the specific parallel architecture, using one of the PEs as a serial processor.

4. *Phase identification and code parallelization*, in which the code is partitioned into *phases*, that will be atomic groups of instructions executed by PEs in the active set only. Phases themselves are designed in such a way that they automatically define the proper active set.

5. *Optimization*, to reduce the parallelism overhead by maximizing the active fraction by dynamically adjusting the schedule in which the phases are broadcast.

Figure 15 illustrates the process in step 4, where phases are identified in the flow-chart reflecting the algorithm produced at step 3, and the code is instrumented to handle parallel execution.

In case studies implemented with this programming model, we have found that the PE efficiency remains fairly high (above 40 or 50%) even for hundreds or thousands of PEs. Of course,



```
if (PE_phase == 1)
   do (OP A)
   if (C1)  PE_phase := 2
   else     PE_phase := 3

if (PE_phase == 2)
   do (OP C)
   if (C2)  PE_phase := 4
   else     PE_phase := 1

if (PE_phase == 3)
   do (OP B)
   PE_phase := 1

if (PE_phase == 4)
   do (OP D)
   if (C3) END Program
   else    PE_phase := 1
```

Figure 15: From a generic flow-chart (left) to the SIMD-parallelized code according to the *SIMD Phase Programming Model*.

this efficiency is strongly dependent on the nature of the problem and on the actual implementation. Overall, our programming model and results reinforce that the SIMD linear array is a powerful and efficient coprocessor with a wide range of applications.

6 The *Third Big Chip*

Currently, most computers are built around two major computational centers: The CPU and the Graphics Processing Unit, GPU. The CPU is responsible for all computations except those related to image-generation, which are performed by the GPU. If one considers the space of all possible applications, image generation is a narrow niche. However, the computational intensity of this particular application and its popularity, such as in video games, has justified the extra component.

Contrary to intuition, the space of popular applications *expands* with the increased computing power available, rather than saturating. Therefore today we see an increased need for computational power to enable new and emerging applications. As an example, the new video compression standards MPEG-4.10 (also known as H.264) or Microsoft's VC-1 require computational power that cannot be found in any ordinary PC or workstation to produce high-definition compression in real time [34, 26]. However, there is a growing demand for this functionality in different spaces, from video-over-IP to high-definition video conferencing.

Of course, both the CPU and the GPU keep improving their performance at an amazing rate, and in the case of the GPU, also the flexibility and programmability. An approach generically called "GPGPU" for General-Purpose GPU studies how to use the impressive computational power of the GPU to solve general problems. This approach, once only the hobby of a few specialized hackers, has been finding more and more interest and support with the GPU manufacturers. Specif-

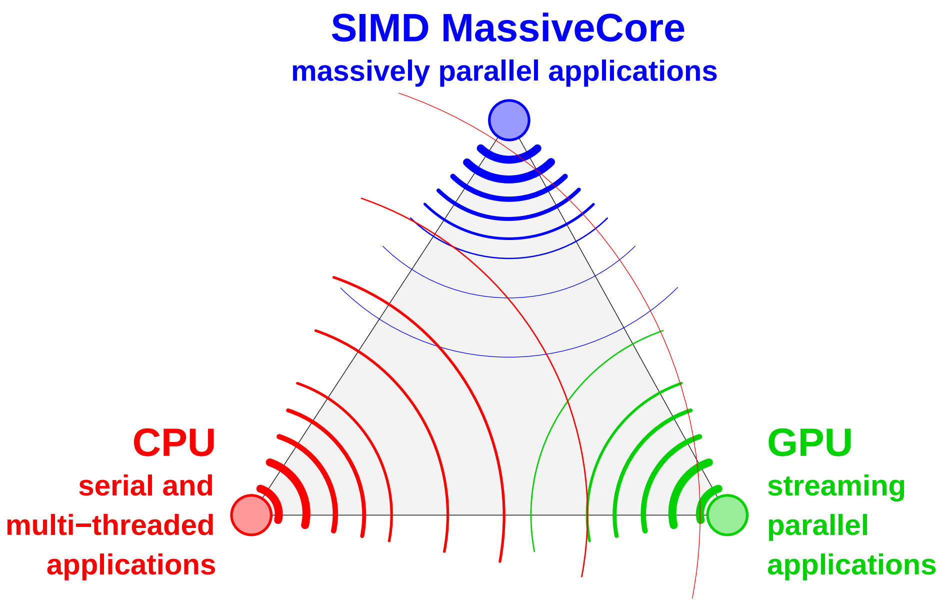

SIMD MassiveCore
massively parallel applications
CPU
serial and multi−threaded applications
GPU
streaming parallel applications

Figure 16: Application domains of the three big chips

ically, nVidia has been very active in this field, and has recently released the most flexible graphics architecture to date, the G80, together with a GPU-specific programming language, CUDA [40].

Even with this added flexibility, it appears that the combination of CPU and GPU does not efficiently cover the space of new massively-parallel applications. The GPU focus on floating-point-intensive streaming applications can lead to inefficiencies and unrealized performance for integer applications, in particular database and data management operations. This is why several companies are now exploring architectural solutions most suitable to become the so-called “third big chip”, to efficiently complement the current two (Fig. 16).

We believe that the massive-core SIMD architecture is the optimal candidate for the job. Its architecture is completely different from that of the CPU and of the GPU, and therefore its performance peaks at completely different coordinates in the application space. The SIMD architecture has a minimal amount of control, making it power-optimal in the best case, which reduces the electrical cost per computation when compared with other solutions even at the same performance level. Also, managing the design complexity of the linear SIMD architecture is straightforward, and this architecture is also the simplest to scale at the software level.

We are currently designing a next generation of our architecture, called the Pelikan processor. In this architecture, we combine the efficient design of RISC CPUs with the best features of linear SIMD arrays based on our experience with Kestrel. We expect the result to be an outstanding machine with exceptional performance overall, as well as per square nm and per Watt.

7 Conclusions

Are programmable systolic arrays a technology that has come and gone? At the time of B-SYS, software-programmable logic was just starting to become viable for computing. B-SYS provided a much more readily programmable environment, and a much higher density of computation. At that time a revolution in computing machines was happening: the availability of high-scale integration that enabled systolic arrays was making possible their extension to more flexible architectures. The merging of SIMD parallel architecture with the systolic structures resulted in machines like Kestrel.

Not exactly as power- and area-efficient as systolic arrays or ASICs, SIMD processors were and still are flexible enough to cover *a range* of applications. While the optimal mapping of general problems on SIMD is still unsolved, we have seen that it is not difficult to map highly parallel problems in a way that makes these architectures competitive. With a single control unit, almost all the power and area are used for datapath and memory. In this sense, they are therefore "optimal". For problems in which the computation flow is data-independent or with small data-dependent conditional branches, these architectures are orders of magnitude faster and more efficient than multi-core or multi-CPU systems. Even when compared with an emerging parallel coprocessor, the video card used for general-purpose processing (GPGPU) [13], SIMD machines still come up winners in many cases. The graphics pipeline produces a tremendous amount of horsepower, but it is much less flexible than a fully-programmable SIMD parallel processor. Moreover, as of now, the programmer has very little control over the actual degree of parallelism in a video card.

Hardware-programmable logic devices (FPGAs) are interesting in that one supposedly achieves ASIC-like performance with CPU-like programmability. However, FPGAs require knowledge of hardware design, and of FPGA-specific design techniques to achieve performance. And even with state-of-the-art FPGA-accelerated computers [38], supported by expert FPGA designers and expert supercomputing applications developers, real FPGA-based systems in practice only achieve modest speedups. For example, one recent design reports achieving a speedup of three compared to microprocessor runtime on a high-performance workstation powered by an add-on processor with two FPGAs for a biomolecular simulation [1]. Other critical factors include power efficiency and silicon efficiency. Roughly speaking, for the same application, an FPGA's computational power and density is about one order of magnitude lower than that of an equivalent ASIC [7], and, due to the larger number of transistors per computation and interconnection overhead, the power consumption can be an order of magnitude higher.

Finally, the industry trend toward multicores confirm, in a way, that SIMD processors have a place in the computational landscape. In fact, the symmetric multicore architecture has shown the same limitations of networks of workstations and in general Multiple Instruction-Multiple Data (MIMD) systems. In most high-performance applications, MIMD systems run in Single Program-Multiple Data (SPMD) mode, with each processor running the same program, wasting a considerable amount of power to replicate the common control flow. While the advantage of having a homogeneous architecture is still considerable, several companies are experimenting with an evolution of this simple multi-core architecture — the *asymmetric* multicore architecture. Perhaps the most famous example today is the IBM Cell [21], which combines a powerful, complex PowerPC core with 8 simpler and more efficient Synergistic Processing Elements. Another example is the architecture by Cradle Technology, that combines two blocks, each with four general-purpose processors and eight DSPs. The idea is clearly to exploit the good serial performance of few complex cores, and the high parallel performance of several simple cores [3]. Taking this approach to its limit generates what we see as the next computational model: a CPU (and possibly a GPU) with a fully-programmable, massive-core SIMD coprocessor.

The experience with Kestrel shows the tremendous efficiency of such a model.

8 Acknowledgements

The authors thank the many contributors to the hardware, software, and algorithms of the Kestrel project: Manju Anand, Elizabeth Avila, Cyrus Bazeghi, Pieris Berreitter, Alexandra Carey, Andrea Di Blas, David Dahle, Kevin Delaney, Brian Feaster, Leslie Grate, J Alicia Grice, Jeffrey Hirschberg, Kevin Karplus, Hansjoerg Keller, Mark Kendrick, David Fulton, Jason Guilford, Andrew Hill, Jennifer Leech, Gabriel Littman, Daniel Littrell, Francisco J. Mesa-Martinez, Justin Meyer, Michael Morrison, Kaushik Narayanun, David Pease, Eric Perlman, Eric Rice, Goeff Ryder, Osama Salem, Angela Schultz, Don Speck, Aaron Tomb, Gang Wang, Ebin Lee Warner, Doug Williams, and Karl Young.

The authors also thank the National Science Foundation, Affymax, and the University of California for funding the Kestrel project.

References

[1] S. R. Alam, P. K. Agarwal, M. C. Smith, J. S. Vetter, and D. Caliga. Using FPGA devices to accelerate biomolecular simulations. *Computer*, 40(3):66–72, Mar. 2007.

[2] M. Annaratone et al. The Warp computer: Architecture, implementation and performance. *IEEE Trans. Comput.*, 36(12):1523–1537, Dec. 1987.

[3] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *32nd Int.Symp. on Computer Architecture (ISCA'05)*, pages 506–517, June 2005.

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

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

[6] D. M. Dahle, J. D. Hirschberg, K. Karplus, H. Keller, E. Rice, D. Speck, D. H. Williams, and R. Hughey. Kestrel: Design of an 8-bit SIMD parallel processor. In R. B. Brown and A. T. Ishii, editors, *Proc. 17th Conf. on Advanced Research in VLSI*, pages 145–162. IEEE CS, Sept. 1997.

[7] A. Di Blas, D. Dahle, M. Diekhans, L. Grate, J. Hirschberg, K. Karplus, H. Keller, M. Kendrick, F. Mesa-Martinez, D. Pease, E. Rice, A. Schultz, D. Speck, and R. Hughey. The UCSC Kestrel parallel processor. *IEEE Trans Parallel and Distributed Systems*, 16(1):80–92, Jan. 2005.

[8] A. Di Blas and R. Hughey. Explicit SIMD programming for asynchronous applications. In E. E. Swartzlander et al., editors, *Proc. Int. Conf. ASAP*, pages 258–267, Los Alamitos, CA, July 2000. IEEE CS.

[9] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Seqeunce Analysis: Probabilistic models of proteins and nucleic acids*. Oxford University Press, Oxford, England, 1998.

[10] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.

[11] P. Frison, D. Lavenier, H. Leverge, and P. Quinton. MICSMACS: A VLSI programmable systolic architecture. In J. McCanny, J. McWhirter, and J. Earl Swartzlander, editors, *Systolic Array Processors*, pages 146–154. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[12] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building and using a highly parallel programmable logic array. *Computer*, 24(1):81–89, Jan. 1991.

[13] GPGPU. General-purpose computation using graphics hardware. http://www.gpgpu.org.

[14] J. D. Hirschberg, R. Hughey, K. Karplus, and D. Speck. Kestrel: A programmable array for sequence analysis. In J. Fortes et al., editors, *Proc. Int. Conf. ASAP*, pages 25–34, Los Alamitos, CA, July 1996. IEEE CS.

[15] R. Hughey. Programming systolic arrays. In E. Lee and T. Meng, editors, *Proc. Int. Conf. ASAP*, pages 604–618, Los Alamitos, CA, Aug. 1992. IEEE CS.

[16] R. Hughey. Parallel sequence comparison and alignment. In P. Capello et al., editors, *Proc. Int. Conf. ASAP*, pages 137–140, Los Alamitos, CA, July 1995. IEEE CS.

[17] R. Hughey. Parallel sequence comparison and alignment. *CABIOS*, 12(6):473–479, 1996.

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

[19] R. Hughey and D. P. Lopresti. Architecure of a programmable systolic array. In K. Bromley, S. Y. Kung, and E. Swartzlander, editors, *Proc. First Int. Conf. Systolic Arrays*, pages 41–50. IEEE CS, May 1988.

[20] R. Hughey and D. P. Lopresti. B-SYS: A 470-processor programmable systolic array. In C. Wu, editor, *Proc. Int. Conf. Parallel Processing*, volume 1, pages 580–583, Boca Raton, FL, Aug. 1991. CRC Press.

[21] IBM. http://www.research.ibm.com/cell/.

[22] C. Ionescu-Zanetti, L. Wang, D. D. Carlo, P. Hung, A. Di Blas, R. Hughey, and L. P. Lee. Alkaline hemolysis fragility is dependent on cell shape: Results from a morphology tracker. *Cytometry Part A*, 65A(2):116–123, April 2005.

[23] K. Karplus, R. Karchin, J. Draper, J. Casper, Y. Mandel-Gutfreund, M. Diekhans, and R. Hughey. Combining local-structure, fold-recognition, and new-fold methods for protein structure prediction. *Proteins: Structure, Function, and Genetics*, 53(S6):491–496, 2003.

[24] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, , and J. Lancaster. Biosequence similarity search on the Mercury system. In *Proc. IEEE 15th Int. Conf. on Application-Specific Systems, Architectures and Processors (ASAP04)*, pages 365–75, 2004.

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

[26] J.-B. Lee and H. Kalva. An efficient algorithm for vc-1 to h.264 video transcoding in progressive compression. *2006 IEEE International Conference on Multimedia and Expo*, pages 53–56, July 2006.

[27] R. J. Lipton and D. Lopresti. A systolic array for rapid string comparison. In H. Fuchs, editor, *1986 Chapel Hill Conference on VLSI*, pages 363–376. Computer Science Press, Rockville, MD, 1985.

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

[29] C. A. Mead and L. A. Conway. *Introduction to VLSI Systems*. Addison-Wesley, Reading, MA, 1980.

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

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

[32] Paracel, Inc. Genematcher2 product literature. http://www.paracel.com, 2001.

[33] E. Rice and R. Hughey. Multiprecision division on an 8-bit processor. In T. Lang, J.-M. Muller, and N. Takagi, editors, *Proc. 13th IEEE Symp. Computer Arithmetic*, pages 74–81. IEEE CS, July 1997.

[34] I. Richardson. *H.264 and MPEG-4 Video Compression — VIdeo Coding for Next-Generation Multimedia*. Wiley, 2003.

[35] B. Schmidt, H. Schroder, and M. Schimmler. Massively parallel solutions for molecular sequence analysis. In *International Parallel and Distributed Processing Symposium*, pages 186–192. IEEE, Apr. 2002.

[36] R. K. Singh, D. L. Hoffman, S. G. Tell, and C. T. White. BioSCAN: a network sharable computational resource for searching biosequence databases. *CABIOS*, 12(3):191–196, 1996.

[37] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.

[38] SRC Computers, Inc. http://www.srccomp.com/.

[39] C. Tarnas and R. Hughey. Reduced space hidden Markov model training. *Bioinformatics*, 14(5):401–406, 1998.

[40] Technical brief. NVIDIA GeForce 8800 GPU architecture overview. `http://www.nvidia.com/object/IO_37100.html`.

[41] T. A. Thanaraj and T. Flores. Assessment of Smith-Waterman sequence search tools implemented in Biocellerator, FDF, and MasPar. Technical report, European Bioinformatics Institute, Wellcome Trust Genome Campus, Hinxton, Cambridge, Feb. 1997. http://www.ebi.ac.uk/industry/Documents/publications/report.pdf.

[42] Time Logic Inc. Decypher II product literature. http://www.timelogic.com, 2002.

[43] J. E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, and P. Boucard. Programmable active memories: reconfigurable systems come of age. *IEEE Trans. VLSI Systems*, 4(1):56–69, 1996.