Softcore Vector Processor for Biosequence Applications

Arpith Chacko Jacob
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri 63130–4899
Email: jarpith@cse.wustl.edu

Brandon Harris
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri 63130–4899
Email: bbh2@cse.wustl.edu

***Abstract*— In this paper we describe the SVP, a Softcore Vector Processor targeted toward Computational Biology and streaming applications. The SVP is a software programmable architecture constructed from predefined hardware building blocks. We leverage the flexibility and power of an FPGA to enhance a streaming vector processor design. Each functional unit includes an instruction controller, parallel processing elements with shared registers, and a memory unit which provides access to both local and streaming data. We target the Smith-Waterman sequence alignment algorithm and report estimated performance numbers based on a Virtex-4 FPGA implementation.**

I. Introduction

The Softcore Vector Processor was developed in response to the need for accelerating a wide range of computational biology applications. The design philosophy of the SVP is based on two important goals: adaptability and high performance.

The SVP is designed to be a software programmable and hardware customizable platform. Instruction based execution allows support for a large number of algorithms in computational biology. However, the fact that different classes of applications require different subsets of hardware resources argues for a customizable hardware design built from primitives. The second goal of the project is to achieve programmability without sacrificing performance. The SVP was designed to perform competitively with full custom solutions available today.

Currently available genome databases are growing exponentially in size [1], making it difficult for software analysis tools to keep up. A number of hardware solutions utilizing special purpose VLSI or reconfigurable hardware such as Field Programmable Gate Arrays (FPGAs) have been proposed to bridge this gap. Systems such as the BISP, and the Kestrel [2], built on custom asics support high performance designs but are expensive to manufacture and cannot be adapted for applications that require differing hardware resources. FPGA solutions tend to be specialized, with support for new applications requiring a laborious re-design of the hardware.

The SVP is a vector processor targetting algorithms that can be easily mapped to a systolic array of processing elements (PEs). Programmable instructions execute on PEs that operate on different units of data achieving fine grained parallelism. Sequence analysis tools commonly use the streaming paradigm, where a large genomic database is streamed through

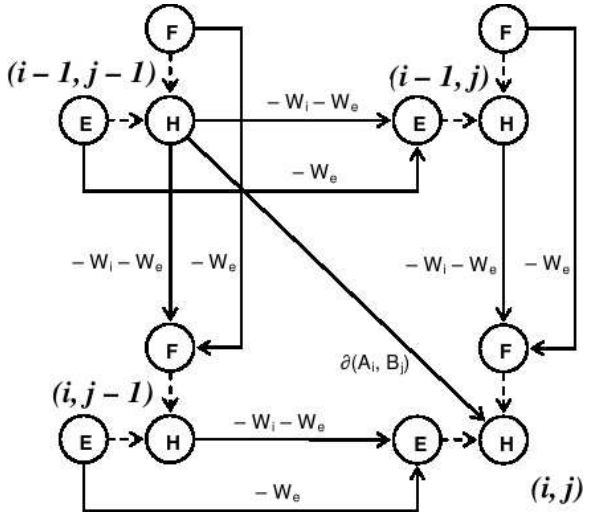

Fig. 2. Data dependencies in the Smith-Waterman algorithm

logical units that process and filter the stream. The SVP is designed to efficiently and seemlessly operate on multiple streams of data without increasing programming complexity. An FPGA is used as the implementation architecture for a number of reasons. FPGA solutions are far less expensive than custom asics; as FPGA devices improve with technology the SVP can be rebuilt on the new target with minimal additional work. The SVP gains its flexibility by being hardware customizable. PEs may be specialized by adding units such as hardware multipliers or a hashing module to support new applications, without having to drastically modify the underlying architecture. Incremental enhancements make it easier to support increasingly complex applications, thus making maximum use of the programmability of FPGAs.

The rest of the paper is organized as follows. Section II introduces the target application: the Smith-Waterman algorithm. Section III describes the SVP architecture in detail with performance numbers reported in Section IV. Section V briefly surveys related work and section VI describes future directions for the SVP. Section VII concludes.

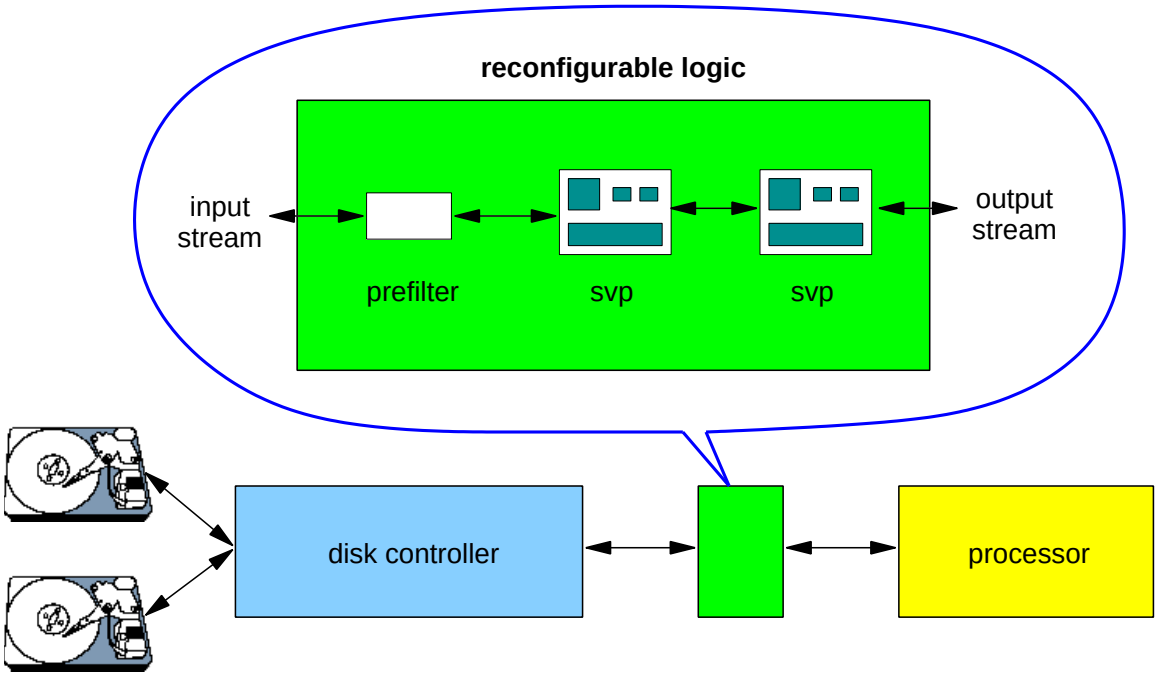

Fig. 1. Mercury Pipeline

II. The Smith-Waterman Algorithm

The Smith-Waterman algorithm [3], [4] is used to compare two sequences for similarity in the hope of detecting common features. The algorithm aligns two sequences by looking at the following mutations: substitutions (replacing one residue with another), insertions and deletions (adding gaps). Given a query sequence Q and a database sequence D, the algorithm scores alignments using a substitution matrix δ that defines the probabilities of mutating from one residue to another, and an affine gap penalty that defines a gap initiation penalty a, and an extension penalty b. The dynamic programming recurrence is computed as follows:

$$E_{i,j} = \max \left\{ \begin{array}{l} E_{i-1,j} - b \\ H_{i-1,j} - a - b \end{array} \right\}$$

$$F_{i,j} = \max \left\{ \begin{array}{l} F_{i,j-1} - b \\ H_{i,j-1} - a - b \end{array} \right\}$$

$$H_{i,j} = \max \left\{ \begin{array}{l} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} + \delta_{Q_i,D_j} \end{array} \right\}$$

with the base case being: $E_{i,j} = F_{i,j} = H_{i,j} = 0$ if $i = 0$ or $j = 0$. The highest value in the H matrix gives the score of an optimal local alignment. The alignment can be retrieved by a linear space divide and conquer strategy described in [5]. We limit our implementation to finding the best score in linear space. The alignment for pairs of sequences that score above a certain threshold may be retrieved by a general purpose processor.

The Smith-Waterman algorithm lends itself readily to fine-grained parallelism [6]. Figure 2 shows the data dependencies of a cell in the H matrix. The computation for the $(i,j)^{th}$ cell depends only on the cell immediately above ($(i, j-1)$), to the left ($(i-1, j)$) and on the previous anti-diagonal ($(i-1, j-1)$). Cells on the same anti-diagonal can be computed in parallel without any data dependencies between them. Computation proceeds in horizontal wavefronts along each anti-diagonal of a fixed number of cells, k, through the entire length of the query sequence. This is repeated along the entire database sequence in steps of k, while keeping track of the best score. The speedup achieved over the sequential version is due to the simultaneous computation of k cells.

To quantify the performance of the algorithm, the measure Millions of Cell Updates per Second (MCUPS) is defined. It represents number of cells that can be computed in the H matrix per second for sufficiently long query and database sequences. This includes the computation of corresponding cells in the E and F matrices along with supporting operations such as table lookups.

III. System Architecture

The SVP is designed to be part of the Mercury system [7] a disk based, high throughput computation architecture. Data from a store is streamed directly into reconfigurable logic at disk access speeds. The reconfigurable logic performs low complexity, highly parallellizable tasks close to the disk to filter the stream to manageable sizes for later more complex stages. Results from the final stage are passed via the I/O bus to a general purpose processor for post-processing. Previous work done on the Mercury system performs text and image searches as well as sequence matching [8], [9]. The Mercury

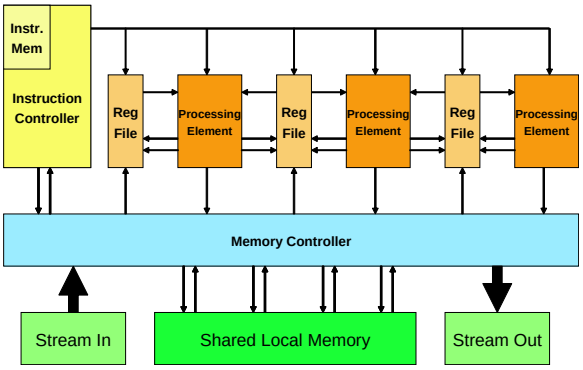

Fig. 3. SVP Functional Unit

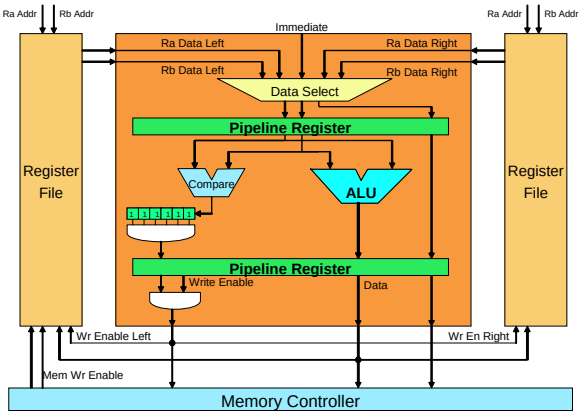

Fig. 4. SVP Processing Element

system is an ideal architecture for streaming applications with pipelined logic blocks manipulating the stream. The SVP is designed to work seemlessly as part of this logic pipeline (Figure 1) with one or more of the softcore processors running more complex functions at the later stages.

We envision the SVP to be a softcore processor that may be customized according to the function being implemented. Besides modifying implementation parameters such as the number of processing elements and registers, or the data width, more significant design changes can be made without drastically affecting the overall architecture. For example, a hashing unit may be added to each processing element, and the memory controller modified to operate on bit vectors to implement a BloomFilter [10] unit. This approach exploits the programmability of FPGAs to a greater extent than most current FPGA designs that target a single algorithm. When describing the SVP architecture in the following sections, we mention the design parameters that is specific to our Smith-Waterman unit.

A. Functional Unit

The basic organization of the SVP consists of a functional unit (FU) that performs a well defined computation on the input stream and sends the filtered or transformed data via the output stream to the following unit. The block diagram of an FU is shown in figure 3. It consists of a user programmable instruction controller (IC) that broadcasts instructions to a linear array of processing elements (PEs). These two units together support the Single Instruction stream Multiple Data stream (SIMD) program organization. A memory controller (MC) provides shared and/or local memory access to the PEs and the IC. The MC plays a vital role in providing access to the input and output data streams.

B. Instruction Controller

The Instruction Controller is a simple three-stage pipelined processor. The Fetch stage loads instructions from programmable instruction memory before decoding it in the Decode stage. The SVP is progammed using two classes of instructions: those that operate on data in the PEs and those meant primarily to affect program flow, which is executed on the IC. Decoding for both classes of instructions is done in the Decode stage, with the decoded signals for the former being broadcast directly to the PEs. The decoded IC instructions pass through a simplified ALU stage that supports basic arithmetic operations. During execution of IC instructions NOP codes are broadcast to the PEs.

The IC contains a register file of user accessible registers that are distinct from the PE. Register zero holds a constant value of zero and is used primarily as an operand for immediate instructions. Register one termed the Memory Register, is used by the memory controller to store data words requested from memory. Hence all memory read operations must be followed by a read from this register. This is done to simplify design of the register file. All remaining registers are used as general purpose registers.

Load/Store instructions interface the IC to the memory unit and is used to read user instructions and control parameters from an external source. Program flow is controlled by unconditional and conditional instructions executed on the IC. An atomic loop instruction is implemented to simulate *C for loops*.

The concept of Instruction Register broadcast is introduced in this section. An observation made during the analysis of common sequence analysis algorithms was that a large number of registers in the PEs were used to store base addresses of tables in memory. Though individual processing elements index different locations in the table, they are offset by a register that holds the same base address across the PEs. When implementing Instruction Register broadcast, base addresses of tables or other parameters that are constant across PEs are instead stored in the IC registers. The value of an instruction register is then broadcast as an immediate value operand to all PEs. This represents a huge saving in terms of logic area as the value need only be stored once in the IC instead of on all the PEs. In our Smith-Waterman implementation, Instruction Register Broadcast reduced the number of registers required on PEs by 40%.

Our softcore SVP instantiation to implement Smith-Waterman uses 32 16-bit registers including the two special purpose registers. The register file includes two read ports and

two write ports: one to write to the memory register, and the other to all general purpose registers.

C. Processing Element

Processing Elements are independently operating core computational units of the vector processor. Register files placed alongside PEs provide program storage. The PE is a simple two-stage pipelined execution unit that executes instructions broadcast to it on data available in its registers. The single cycle ALU stage performs common logical operations, bit shifting as well as addition and subtraction. Arithmetic operations use saturated arithmetic, where the result is clamped to all ones or zeroes in case of an overlow or underflow respectively.

The design philosophy behind the PE was to build a resource efficient, specialized unit that implemented common operations performed by target applications to improve performance. For example, since sequence analysis algorithms routinely perform compare-assign operations, the ALU implements single cycle MAX and MIN commands. Another major goal was to design a core unit with a basic set of features that could be expanded incrementally. The current design uses signed arithmetic and does not include a multipler unit. Support for these features may be added when they are critical to an application's performance. The data path implemented is 16 bits long, which is adequate for most targetted applications.

1) Shared Registers: Register files are unique in the sense that each is shared by two processing elements, placed adjacent to it. In addition to providing storage space, it facilitates communication between PEs. The targetted sequence analysis applications commonly require close communication between adjacent PEs, for example to stream sequences through the array. It is important to not only provide single cycle access to the two PE register banks, but also to make it as seemless and intuitive as possible to the programmer. The size of register address fields in all instructions have been increased by a bit to indicate the appropriate bank. The first stage of the PE is a register select MUX that reads the value of the register from the selected bank. Since all PEs execute the same instruction in any clock period, read/write conflicts between adjacent PEs are automatically avoided.

There are as many register files as there are PEs, all accomodated on a single chip. Special register zero holds a constant value of zero, while register two is the Memory Register used as described in the IC. Register one holds the processing element identification number (PE ID), which is a unique 16-bit integer constant varying from zero to one less than the number of processing elements. The PE ID register enforces an ordering on the PEs and is used commonly to index unique data locations in memory, in spite of them running the same instruction. Two read and write ports are provided at each register file. Our implementation provides an additional 13 16-bit general purpose registers.

2) Conditional Execution: One of the consequences of broadcasting instructions to all PEs is that there is no way to limit execution to a subset. When streaming data through the linear array for example, the new data value has to be introduced at the first PE while propogating old values to higher ordered PEs. One solution is to introduce conditional bits attached to every instruction. However this introduces overhead that is not justified due to the low percentage of conditional instructions.

We use special conditional instructions *bmset* and *bmend* to signify blocks of code that are to be executed on a subset of PEs. A special 8-bit mask register in each PE hold its execution state. *bmset* compares two operands on a PE and performs a bit shift left and set operation on the mask register if the condition is satisfied. The logical *and* of all bits in the mask register is used to switch the PE on and off, i.e: the write back phase on a PE is disabled if the mask register is not all ones. *bmand* is used to further constrain the subset of PEs selected while *bmor* increases it, by and-ing or or-ing respectively the result of the comparison operation with the lsb of the mask register. *bmelse* inverts the lsb of the mask register and is used to select the difference subset of PEs. Up to eight nested conditional blocks may be executed with the current setup.

D. Memory Controller

Memory organization is crucial to the performance of the target application. In line with our design philosophy, we define the memory interface between the PEs and the Memory Controller, while leaving the implementation details dependent on the target application.

Smith-Waterman performs a large number of table lookups in the inner loop, while memory writes are less frequent and confined to the final PE. Our implementation of Smith-Waterman also calls for a large amount of data such as the score table and query sequence to be shared between PEs. Hence, memory is organized in distributed fashion on the SVP.

We make use of high speed block RAM storage available on FPGAs. Block RAMS on Xilinx Vertex series FPGAs have two independent ports, allowing servicing of requests from two PEs simultaneously. Block RAM replication is necessary to service read requests from all PEs and the IC simultaneously in a single clock. As a consequence, memory write operations require multiple clock cycles to be satisfied. A finite state machine registers write requests from PEs and sends two per clock to all replicated copies of the memory, stalling the PEs and IC during this time. It optimizes for the special case where writes come only from the first and the last PE, satisfying them in a single clock cycle. The provision of a stall signal frees the programmer from the details of the implementation of the memory controller. All memory read and write requests from the PE have the same latency of five clock cycles. Hence with the current design, all memory read and write operations in the inner loop of the Smith-Waterman computation are satisfied in a single clock, providing maximum performance.

Our implementation provides 2048 words of shared memory storage space. It is also possible to easily switch between a distributed organization and a local memory organization with a single instruction. In the latter case, the FSM directly routes read and write requests from a PE to the corresponding

TABLE I

SVP SAMPLE SOURCE CODE

```
1:  _query_loop:
2:  subir       %r8, %r3, %ir10
3:  nop
4:  nop
5:  max         %r4, %r4, %r8
6:  add         %r3, %r19, %r0
7:
8:  bmseti      %r1 EQ PE_NUM_ELEMENTS - 1
9:  icaddi      %ir15, %ir8, PE_NUM_ELEMENTS - 1
10: nop
11: nop
12: ldir        %r2, %r0 (%ir15)
13: nop
14: nop
15: nop
16: nop
17: addi        %r3, %r2, 0
18: bmend
19:
20: ld          %r2, %r0 (0xFFFF)
21: icaddi      %ir7, %ir7, 1
22: icaddi      %ir9, %ir9, 1
23: icloop      %ir4, %ir5, _query_loop
```

block RAM. The block RAM is now divided between two PEs providing each with 1024 words of local storage.

1) Stream Organization: An important function of the memory controller is to provide stream processing capabilities to the PEs. Communication with the software interface as well as external stages in the pipeline is done via data streams through the memory controller. In addition to streaming data (in our case the database sequence), instructions to program the instruction controller, as well as tables stored in memory are sent by the software interface via the input stream. Results of computation, such as the Smith-Waterman scores are sent via the output stream.

To handle both streams, FIFOs are memory mapped in the address space of the SVP memory controller. On receiving a read request at the stream address, the finite state machine reads data words from the input FIFO, one for each PE. Similarly a write request at the address is mapped to the output FIFO. This provides the programmer seemless access to streams using the same load/store instructions used to access shared memory on chip. Latency for stream access remains the same as that of shared memory access. However, a problem arises if the input FIFO is empty on a read, or the output FIFO is full on a write. We solve this problem by stalling the PEs and IC thus presenting a constant five cycle clock latency to the programmer.

The memory controller may be also modified to support multiple streams as well as other data structures such as stacks. Memory mapped stream processing provides immense flexibility and is the key to placing a vector processor in a streaming pipeline.

E. Instruction Set Architecture

The SVP instruction set architecture is based on the MIPS ISA, easing the transition to SIMD programming. Instructions are divided into two classes: those executed on the IC, and those on the PEs. The IC instructions typically control program flow or modify the IC register file. These include an add/subtract, conditional and unconditional branches as well as memory load/store instructions. Instructions on PEs operate on data elements and include add/subtract, logical, bit shift, conditional execution and memory load/store instructions.

Instructions are 34 bits long and are encoded in well defined fields thus simplifying the decoding logic on the IC. In keeping with the design philosophy, the ISA is designed to be customizable. We define a core instruction set that supports basic functionality described above. Over half the opcode space is provided for use as an extended instruction set to support functionality specific to a class of applications. Extended instructions implemented include MAX, MIN and LOOP instructions.

Table I shows sample SIMD code that executes on the SVP. Instructions that execute on the IC are prefixed by *ic* and operate on IC registers specified by %irN. Line 2 is an example of Instruction Register Broadcast, which is used to send a constant value as an immediate. The first source operand on line 6 selects register 3 on the register file to the right of a PE, while the second operand selects register 0 on the register file to the left of a PE. The code block on lines 9-17 is conditionally executed on the final PE, specified by the bmset instruction. Line 20 is an example of a read from an input stream mapped to memory location 0xFFFF. Finally, the loop instruction on line 23 decrements its counter register %ir4 by a step value in %ir5 and conditionally jumps to the target if the counter is not zero.

TABLE II

SMITH-WATERMAN SPEEDUP

System	Frequency	MCUPS	Speedup
P4	1.8 GHz	15	1
SVP16	150 MHz	52	3.47
SVP32	150 MHz	103	6.87
SVP64	125 MHz	167	11.13
SVP128	120 MHz	302	20.13
SVP128	150 MHz	378	25.20

TABLE III

HARDWARE RESOURCE USAGE

PEs	Frequency (Mhz)	Area	Block RAMS
16	150	13%	22
32	150	22%	38
64	125	41%	70
128	120	80%	134

IV. PERFORMANCE

We have implemented the SVP in VHDL verifying functional correctness. In this section we compare the performance of the SVP with varying numbers of PEs: 32, 64 and 128, estimated to run at 150 Mhz, 125 Mhz and 120 Mhz respectively on a single chip. We believe with hand placement and

TABLE IV
COMPARATIVE PERFORMANCE

System	Freq	PEs/Chip	MCUPS/PE	Chips	MCUPS/Chip	Cost ($1000)	$/MCUP
GeneMatcher2	192 MHz	192	5.21	16	1000	69[a]	4.31
SVP128	150 MHz	128	2.95	1	378	5	13.23
SVP128	120 MHz	128	2.36	1	302	5	16.56
SVP64	125 MHz	64	2.61	1	167	5	29.94
SVP32	150 MHz	32	3.22	1	103	5	48.54
Kestrel	20 MHz	64	0.78	8	50	25[a]	62.50
Fuzion 150	200 MHz	1536	1.63	1	2500	?	?

routing the SVP128 can be synthesized to run at 150 Mhz. We have implemented a parallel version of the Smith-Waterman algorithm and run it on the SVP16 simulation on Modelsim to verify correctness. We extrapolate performance numbers based on this to higher configurations.

Table II shows the speedup achieved by the SVP over a Pentium 4 1.8Ghz workstation running a 32-bit integer implementation of the Smith-Waterman algorithm. For most sequence analysis algorithms 16-bit data words are sufficient, which considerably reduces the size of the SVP with a corresponding increase in performance. Our lowest configuration using 32 processing elements runs over 3 times as fast as the Pentium 4. The speedup increases linearly with the number of processing elements with an expected speed of 378 MCUPS on the SVP128 running at 150Mhz. This configuration provides a greater than 25-fold speedup over the sequential version.

Table IV compares the performance of the SVP against other systems that perform sequence analysis as described in [2]. The Kestrel and Fuzion 150 architectures are software programmable parallel processors built on custom ASICs. The Paracel GeneMatcher2 is an FPGA based custom solution to accelerating Smith-Waterman. The SVP provides the best performance per PE compared to the other programmable architectures. The per chip performance numbers are between two and seven times better than the Kestrel system. This is explained partially due to the fact that the Kestrel was designed and manufactured using a $0.5\mu m$ process.

The SVP is a cost-effective solution with a price to performance ratio that is beaten only by the GeneMatcher2 system. However, the SVP can be programmed to exploit fine-grained parallelism in a large class of applications thus proving to be far more cost-effective. The SVP outperforms other programmable processors in terms of performance and cost, and as far as we are aware is the only system that allows hardware customizability.

We synthesized the SVP architecture on a Xilinx Vertex 4 VLX200 part and report results in table III. We use between 18% and 80% of LUTs available on the chip and between 22 and 134 block RAMs for the different configurations. Block RAM usage in our design includes one per PE in the memory controller, two for the IC data memory, two for the instruction memory and two for FIFOs to buffer stream data. We were able to achieve a clock speed of 120Mhz for the 128 PE configuration but believe this can be further increased to 150Mhz with hand placement and routing. Further, the 32 and 64 PE configuration models may be built on a smaller size part such as the Vertex 2 decreasing the manufacturing cost of the overall system.

V. RELATED WORK

Sequence analysis algorithms, in particular Smith-Waterman, has been the target of a large number of systems that exploit its inherent fine-grained parallelism to speedup sequence comparison. A good survey is in [2], [11], [12] and divides such systems into five main categories. The first category of systems are ASIC based solutions built to accelerate a single application. They are inflexible and cannot adapt to changes in algorithms. The second category includes custom FPGA based solutions again targetting a single application. While technically being a programmable device, it requires considerable hardware expertise and design engineering to support new applications. The third category of devices are software programmable SIMD arrays built on custom ASICS, such as the kestrel. A user with far less hardware expertise can map new applications into the microcode of the system. However, a major drawback is the inability to run more complex applications that require hardware resources not present at manufacturing time. Category four systems include clusters and supercomputers, while category five systems include modern high-performance uniprocessors. While the latter two classes are easy to program, they perform less well than the other classes.

We build on the work done at UCSC on the Kestrel Parallel Processor using the concept of Processing Elements, Shared Registers and Conditional Execution. The SVP is unique in the sense that it tries to bridge the gap between the second and third categories. Being a software programmable vector processor, it is flexible enough to run multiple applications and adapt to new algorithms. By targetting an FPGA device we retain hardware flexibility, so an advanced user can add resources to support more complex applications. The SVP's stream processing capabilities makes it an excellent choice as a stage in a logic pipeline. The hardware pipeline of BLAST [8] presents an excellent opportunity to incorporate the SVP. The SVP running a slightly modified version of Smith-Waterman (gapped or ungapped extension) can be easily added to the latter stages of the BLAST pipeline to filter data.

VI. Future Work

Future work on the SVP will focus on supporting new applications such as HMMer and other systolic array algorithms. We intend to focus on streaming applications such as BLAST, using the SVP as a logic unit in the pipeline. Support for new applications will add new hardware primitives and require ISA expansion.

A new avenue for improving the computational capabilities of the PEs is to more effectively use the resources available on modern FPGAs. These include hardware multipliers, and DSP elements that can perform high-speed, multi-cycle multiplications at little cost in terms of area usage. Adding these units as part of the single cycle ALU stage of the PE will greatly increase the class of applications supported.

VII. Conclusion

The SVP was designed to be a high-performance programmable system. We believe its stream processing capabilities and its ease of programming will allow us to run a large class of applications. The design choice of targetting FPGAs and the corresponding hardware customizability afforded will enable us to incrementally add to the compute capabilities, while still maintaining comparable performance with the state of the art.

Acknowledgment

The authors would like to thank our instructor Dr. Young Cho for valuable suggestions offered during the course of the project. We would like to thank our advisors Dr. Jeremy Buhler and Dr. Roger Chamberlain for their insight. The authors appreciate the help provided by Joseph Lancaster with the synthesis tools.

References

[1] "Growth of genbank: National center for biological information." `http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html`.

[2] A. D. Blas, D. Dahle, M. Diekhans, L. Grate, J. Hirschberg, K. Karplus, H. Keller, M. Kendrick, F. J. Mesa-Martinez, D. Pease, E. Rice, A. Schultz, D. Speck, and R. Hughey, "The kestrel parallel processor," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 1, 2005.

[3] T. F. Smith and M. S. Waterman, "Identifi cation of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[4] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of Molecular Biology*, vol. 162, no. 3, pp. 705–708, 1982.

[5] D. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, vol. 18, no. 6, pp. 341–343, 1975.

[6] A. C. Jacob and S. Sanyal, "Whole genome comparison using commodity workstations," 2003.

[7] R. D. Chamberlain, R. K. Cytron, M. A. Franklin, and R. S. Indeck, "The mercury system: Exploiting truly fast hardware for data search," *Proc. of Int'l Workshop on Storage Network Architecture and Parallel I/Os*, pp. 65–72, September 2003.

[8] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, and J. Lancaster, "Biosequence similarity search on the mercury system," *Proc. of the IEEE 15th Int'l Conf. on Application-Specific Systems, Architectures and Processors*, pp. 365–375, September 2004.

[9] J. Lancaster, J. Buhler, and R. D. Chamberlain, "Acceleration of ungapped extension in mercury blast," *Proc. of 7th Workshop on Media and Streaming Processors*, November 2005.

[10] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[11] R. Hughey, "Parallel sequence comparison and alignment," *Proc. Int. Conf. Application-Specific Array Processors, IEEE Computer Society Press*, July 1995.

[12] R. Hughey, "Parallel hardware for sequence comparison and alignment," *CABIOS*, vol. 12, no. 6, pp. 473–479, 1996.