VENICE: A Compact Vector Processor for FPGA Applications

Aaron Severance, Guy Lemieux
Department of ECE, UBC
Vancouver, Canada
aaronsev@ece.ubc.ca, lemieux@ece.ubc.ca

***Abstract*—This paper presents VENICE, a new soft vector processor (SVP) for FPGA applications. VENICE differs from previous SVPs in that it was designed for maximum throughput with a small number (1 to 4) of ALUs. By increasing clock-speed and eliminating bottlenecks in ALU utilization, VENICE can achieve over 2x better performance-per-logic block than VEGAS, the previous best SVP. VENICE is also simpler to program, uses standard C pointers into a scratchpad memory rather than vector registers, and has no penalty for unaligned data.**

***Keywords*-vector; SIMD; soft processors; scratchpad memory; FPGA;**

I. Introduction

FPGAs offer low power operation and great performance potential through massive amounts parallelism. Although FPGAs are designed for maximum flexibility to implement any logic circuit, they offer little structure to guide logic designers into building parallel compute engines. As a result, designing a compute accelerator takes more than simple HDL skill, it also takes a deep understanding of how to design compute engines.

Instead of application-specific compute engines, soft processors implemented in FPGA logic may be used to perform computation. Soft processors have different design constraints than fixed processors [6]; they must rely on the FPGA's embedded block RAM for register files or caches and embedded DSP blocks for complex operations such as shifting and multiplication. Operations such as multiplexing are relatively more resource intensive on FPGAs, while addition is relatively cheap thanks to built in carry chains. Soft processors allow more flexibility than fixed processors, as parameters can be changed for every instantiation, varying cache size, adding/removing complex instructions, etc. While clockspeeds are typically limited to a few hundred megahertz, they have abundant on-chip block RAM that can be used in parallel to achieve high memory bandwidth.

This work builds a soft vector processor (SVP), named VENICE, which can be used to accelerate a wide range of tasks that fit the SIMD programming model. VENICE is smaller and faster than all previously published SVPs. Overall, it achieves a superior area-delay product that is roughly $2\times$ better performance per logic block (speedup per ALM) than the previous best, VEGAS [1]. It is also $5.2\times$ better than Altera's fastest Nios II/f processor. As a result, less area is needed to achieve a fixed level of performance, reducing device cost or allowing more room to be left for other application logic.

The key contributions of this work are:

- Removal of vector address register file (area)
- Use of 2D and 3D vectors (performance)
- Operations on unaligned vectors (performance)
- New vector conditional implementation (area)
- Optimized multiplier implementation (area)

Programming VENICE requires little specialized knowledge, utilizing C langauge with simple extensions for data parallel computation. Changes to algorithms require a simple recompile taking a few seconds rather than several minutes or hours for FPGA synthesis. The removal of the vector address register file, streamlining of instructions, and lack of performance penalty for unaligned instructions make VENICE easier to program than previous SVPs.

Previous work describes a compiler for VENICE based on Microsoft Accelerator [4]. By contrast, this paper describes the design, architecture, native programming interface, and native programming results of VENICE.

II. Prior Work

Vector processing has been applied in supercomputers on scientific and engineering workloads for decades. It exploits the data-level parallelism readily available in scientific and engineering applications by performing the same operation over all elements in a vector or matrix. It is also well-suited for image processing.

A. Vector Processing Overview

Classical vector execution originated with the CRAY-1 [5]. The CRAY-1 is a load/store architecture with a fixed number of vector registers (8) of a fixed maximum size (64 elements of 64b each). By streaming data into a pipelined FU, the CRAY-1 produces 1 result per 80MHz cycle. Additional parallelism is obtained by chaining FU's together. However, chaining requires complex register files with multiple read and write ports. Alternatively, several parallel ALUs can operate in lockstep SIMD mode to execute the same instruction. Since each ALU writes back to its own independent partition of the vector register file, multiple write ports are not required [2].

Modern microprocessors are augmented with SIMD processing instructions to accelerate data-parallel workloads. These typically operate on short, fixed-length vectors (e.g., only 128b, or four 32b words) in a fixed number of data registers. Significant overhead comes from instructions to load/pack/unpack these short vectors.

There are two key distinguishing traits of vector processors that set them apart from SIMD processing instructions. First is the use of the vector length (*VL*) control register, which can be changed at run-time to process arbitrary-length vectors up to a certain maximum vector length (*MVL*). Second is the use of complex addressing modes, such as walking through memory in strides to columnwise traverse a 2D array.

B. Soft Vector Architectures

Early soft vector architectures are based on the VIRAM [2] embedded vector processor. VIPERS [8] and VESPA [7] were both modeled closely on VIRAM, with a fixed number of vector data registers of configurable *MVL*.

VEGAS [1] followed VIPERS, but uses an 8-entry vector *address* register file that points into a large vector scratchpad memory rather than vector data registers. The scratchpad can be partitioned into any number of vectors of any length. Furthermore, the ALUs can be fractured to support subword arithmetic. This means a fixed-width vector engine can operate on more elements if they are halfword or byte sizes. Also, it makes more effective use of on-chip memory, since smaller operands are not expanded to fill an entire word as done with prior architectures.

III. VENICE Architecture

VENICE is a new SVP architecture that improves in several areas on VEGAS. A block diagram of the VENICE (Vector Extensions to NIOS Implemented Compactly and Elegantly) architecture is shown in Figure 1. Similar to previous SVPs, VENICE requires a scalar core as the control processor; in this case, a Nios II/f executes all control flow instructions and issues instructions to the VENICE vector core as one or two tandem Nios custom instructions. In our work, Nios is configured to use a 4kB instruction cache and a 4kB data cache. Unfortunately, Nios lacks support for hardware cache coherence, so the programmer must sometimes explicitly flush data from the cache to ensure correctness.

The VENICE vector engine implements a wide, double-clocked scratchpad memory which holds all vector data. Operating *concurrently* with the vector ALUs and the Nios core, a DMA engine transfers data between the scratchpad and main memory. The DMA engine also has a 1-entry control queue, allowing the next transfer to be queued.

It is very important for the DMA engine to operate concurrently with the vector ALUs, because this hides the memory latency of prefetching the next set of data elements.

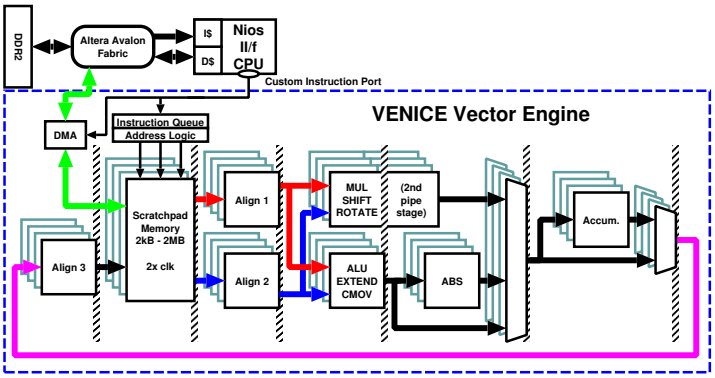

Figure 1. VENICE Architecture (vertical gray bars are pipeline stages)

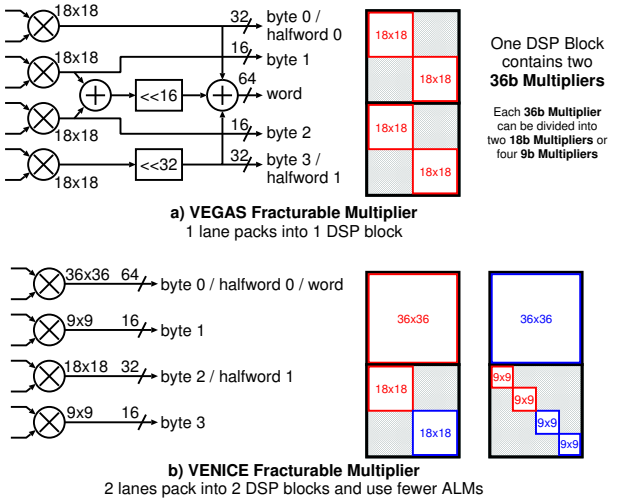

Figure 2. Fracturable Multiplier Styles

All of our benchmarks implement some form of double-buffering, allowing us to hide most of the memory latency.

All vector ALU operations are performed memory-to-memory on data stored in the scratchpad. The configurable number of vector lanes (32-bit vector ALUs) provides scalable data-level parallelism on top of the instruction and memory-level parallelism achieved by running the scalar core, vector core, and DMA engine concurrently. Each 32-bit ALU supports subword SIMD operations on halfwords or bytes, thus doubling or quadrupling the data parallelism available with these smaller data types, respectively.

IV. VENICE Implementation

Below, we will describe each of the key improvements made to VENICE. As a result of these and other optimizations, the design was pipelined to reach 200MHz+ (roughly 50–100% higher than previous SVPs).

A. Removal of Vector Address Register File

One major difference from VEGAS is removal of the vector address register file. Instead of storing vector addresses

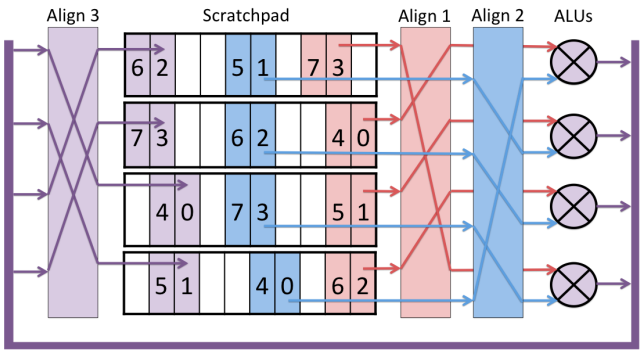

Figure 3. Example of Misaligned Operation

in a separate register file, VENICE relies upon the Nios scalar register file to contain the vector addresses. Each vector instruction is emitted with the contents of 3 scalar registers (two source addresses and one destination address in the scratchpad) as well as immediate bits specifying the opcode and other information. Nios instructions can run concurrently with vector operations, calculating addresses for the next vector instruction while the current one executes.

One drawback of this approach is that increased instruction issue bandwidth is required between the Nios and VENICE vector engine. The NIOS II/f custom instruction interface can only access 2 register values per cycle. Hence, 2 custom instructions are needed to issue 3 required addresses to the vector engine.

B. 2D and 3D Vector Instructions

To achieve high ALU utilization on problems with short but regular vectors, VENICE implements 2D and 3D vector instructions. A 1D vector is an instruction applied to a configurable number of elements, which can be thought of as columns in a matrix. The 2D vector extends this to repeat the 1D vector operation across a certain number of rows. In between each row, each operand (2 source addresses and 1 destination address) will be incremented by a unique configurable stride value, allowing for selection of arbitrary submatrices. When executing, the address logic inside the vector core dispatches a separate vector instruction for each row, and in parallel adds the strides for each operand to determine the address of the next row. As a result, VENICE can issue up to 1 row per cycle. This has a direct extension to 3D instructions for operations on 3D data, or to process multiple 2D matrices with a single instruction.

C. Operations on Unaligned Vectors

When input and/or output vectors are not aligned, data needs to be shuffled between lanes. For example, Figure 3 shows how VENICE uses 3 alignment networks to shuffle unaligned data. This is especially useful when doing convolution, as one operand's starting address is continuously changing and seldom aligned. VENICE suffers no penalty for unaligned operations, in contrast with previous SVPs that required extra instructions to be inserted for alignment.

Note that only two alignment networks are required to align the 2 input operands and 1 output operand; the extra one is in place to allow for future work when operand data sizes mismatch. It could be removed from the current design for a nominal area reduction.

D. Vector Conditional Operations

VIRAM based SVPs traditionally have a separate register file of flag registes that can be used to implement predication and data-conditional operations. VEGAS does not have fixed length vector registers, so it uses a separate, configurable length, flag scratchpad.

In VENICE a single vector flag is written alongside each byte, utilizing the 9th bit in BRAMs that have an extra data bit for parity or optional storage (such as the M9K in modern Altera FPGAs). Conditional move operations that read the flag bits as one of the input operands are used to implement simple predication. The flag and result can be used to check out-of-range results, less-than/greater-than/equal-to, or extended precision (64-bit) arithmetic. Instead of supporting separate flag registers and operations, flags can be stored at different locations and combined using logic operations.

E. Multiplier Implementation

Figure 2 shows VENICE's method of implementing fracturable multipliers versus the partial products method used in VEGAS. The VENICE method does not require additional adders or multiplexers on the inputs, so has lower ALM usage and delay. The only drawback of the VENICE organization is that the multipliers used by a single lane cannot be packed into a single DSP Block. Instead, two lane's worth of multipliers must be packed into two DSP Blocks. Hence, V1 may use an additional half of a DSP Block, but larger designs use the same amount as VEGAS.

V. Native Programming Interface

VIPERS and VESPA both required assembly level programming, with the programmer manually dealing with register allocation and usage. VEGAS introduced more user friendly C macros to dispatch vector assembly instructions, but its vectors were still indexed by an address register file that had to be filled and spilled manually by the programmer. The native VENICE application programming interface (API) also uses C macros to generate inline assembly in C. However, the removal of a vector address register file means that instructions can directly address the scratchpad using C pointers.

The sample code in Figure 4 adds three vectors together. Each macro dispatches one or more vector assembly instructions to the vector engine. Depending upon the operation, these may be placed in the vector instruction queue, or the DMA transfer queue, or executed immediately. A macro that

```
#include "vector.h"

int vadd3(int *A, int *B, int *C, int *D, int length)
{
  // if A,B,C are dynamically modified,
  // then flush them from the data cache here

  const int num_bytes = length * sizeof(int);

  // alloc space in scratchpad, DMA from A to va
  int *va = (int *)vector_malloc( num_bytes );
  vector_dma_to_vector( va, A, num_bytes );

  // alloc and DMA transfer, in one simple call
  int *vb = (int *)vector_malloc_and_dmacpy( B, num_bytes );
  int *vc = (int *)vector_malloc_and_dmacpy( C, num_bytes );

  // setup vector length, wait for DMA
  vector_set_vl( length );
  vector_wait_for_dma();   // ensure DMA done

  vector( VVW, VADD, vb, va, vb );
  vector( VVW, VADD, vc, vb, vc );

  // transfer results from vc to D
  vector_instr_sync(); // ensure instructions done
  vector_dma_to_host( D, vc, num_bytes );
  vector_wait_for_dma();

  vector_free();
}
```

Figure 4. VENICE Native API to Add 3 Vectors

emits a queued operation may return immediately before the operation is finished. Some macros are used to synchronize the vector core, scalar core, and DMA engine.

The VENICE programming model uses a few basic steps:

1) Allocation of memory in scratchpad
2) Optionally flush data in data cache
3) DMA transfer data from main memory to scratchpad
4) Setup for vector instructions (e.g., the vector length)
5) Perform vector operations
6) DMA transfer resulting data back to main memory
7) Deallocate memory from scratchpad

The basic instruction format is `vector(MODE, FUNC, DEST, SRCA, SCRB)`, where the values of MODE and FUNC must be predefined symbols, while the values of DEST and SRCB must be scratchpad pointers and SRCA can be a scalar value or scratchpad pointer.

For example, to add two unsigned byte vectors located in the scratchpad by address pointers `va` and `vb`, increment the pointer `va`, and then store the result at address pointer `vc`, the required macro would be `vector( VVBU, VADD, vc, va++, vb);`.

Space can be allocated in vector scratchpad memory using the special `vector_malloc(num_bytes)` which returns an aligned pointer. The `vector_free()` call simply frees all previous scrachpad allocations, since the common case is to utilize the scratchpad for one kernel/function after which it can be reused for the next kernel/function. DMA transfers and instruction synchronization are handled by macros as well.

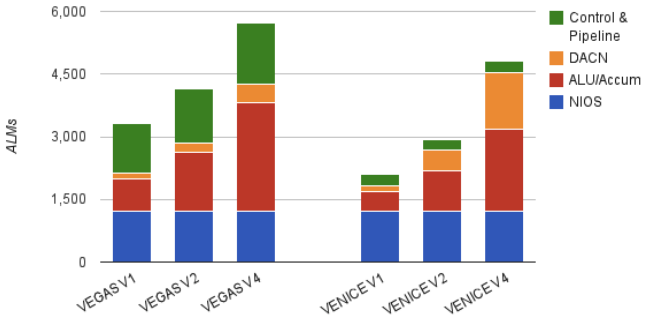

Figure 5. Area Savings Breakdown (ALMs)

VI. Results

All soft processor results in this paper are measured by running the benchmarks on an Altera DE4-530 development system using Quartus II version 11.0. All software self-verifies itself against a sequential C solution.

A. Area and Clock Frequency

The overall compilation results for VENICE are compared with VEGAS in Table I. In this table, the overall Stratix IV-530 device capacity is shown for ALMs, DSP Blocks, and M9K memory blocks. It is important to note that an Altera DSP Block is a compound element consisting of two 36b multipliers. Alternatively, each 36b multiplier can be statically configured as two independent 18b multipliers or four independent 9b multipliers. Altera literature usually quotes device capacity using a special mode that achieves 8 18b multipliers per DSP block.

Table I gives area and clock frequency results for VEGAS and VENICE, including their Nios II/f scalar processor. VENICE uses fewer ALMs, and fewer M9Ks, than VEGAS across the board. With one vector lane VENICE is using 0.5 more DSP Blocks than VEGAS, but there is room for one 18b and two 9b multipliers to be packed into partially used DSP blocks. The clock frequency achieved with VENICE is also 50% higher than VEGAS.

Figure 5 gives a more detailed area breakdown. VENICE has consistently lower area than VEGAS for everything but the alignment network (when using multiple lanes). Precise area values for the V1 configuration are shown in Table II. The savings in each area is primarily due to:

- Fracturable ALU savings due to the new parallel multiplier, which uses fewer adders and requires less input and output multiplexing.
- Control/Pipeline savings due to removal of the vector address register file.
- DMA savings due to not needing to align operands during transfer since there is no penalty for unaligned operations in VENICE.
- Alignment network savings due to being optimized for a small number of lanes, though having multiple

Table I
RESOURCE USAGE COMPARISON

Device or CPU	VEGAS				VENICE			
	ALMs	DSP Blocks	M9Ks	F_{max}	ALMs	DSP Blocks	M9Ks	F_{max}
Stratix IV EP4SGX230	91,200	161	1,235	–	91,200	161	1,235	–
Stratix IV EP4SGX530	212,480	128	1,280	–	212,480	128	1,280	–
Nios II/f	1,223	1	14	283	1,223	1	14	283
Nios II/f + V1 (8kB)	3,831	2	35	131	2,529	2.5	27	206
Nios II/f + V2 (16kB)	4,881	3	49	131	3,387	3	40	203
Nios II/f + V4 (32kB)	6,976	5	77	130	5,096	5	66	190

Table II
AREA SAVINGS BREAKDOWN (ALMS)

	VEGAS	VENICE	Savings
Fracturable ALU	771	471	300
Control/Pipeline	1200	538	662
DMA	501	181	320
Alignment (V1)	136	116	20
Alignment (V4)	448	855	-407

alignment networks means scaling worse than VEGAS when going to a large number of lanes.

B. Benchmark Performance

The characteristics of nine application kernels are reported in Table III. The input, output, and internal data types for each kernel are shown. The overall input data set size is also shown, along with the size of a filter window in the Taps column. Some of these kernels come from EEMBC, others are from VIRAM, and others are written by the authors.

The Nios II/f processor was run at 283MHz with a 200MHz Avalon interconnect and 200MHz DDR2 controller (i.e., at the limit of the DDR2-800 SODIMM). The VENICE V1 and V2 configurations were run synchronously at 200MHz for everything, including the Nios II/f, VENICE engine, Avalon interconnect, and DDR2 controller, while the V4 configuration was run with everything at 190MHz.

Results are in units of millions of output elements computed per second, except matrix multiply (which is not linear in computations per output element), for which we report performance as millions of multiply-accumulates per second (MAC/S). The peak performance of VENICE V4 at 190MHz is 760 million MAC/S, so our matrix multiply code is running at 78% of peak performance, whereas the Nios II/f (assuming it can do one multiply or add per cycle) only achieves 8.3% utilization.

C. Speedup versus Area

Figure 6 demonstrates the speedup versus area increase from VEGAS to VENICE, normalized to the Nios II/f. Speedup results are a geometric mean of the execution times for nine benchmark programs. Area results account for ALMs only. VENICE dominates VEGAS in both area and speed, achieving with just 4 ALUs a geomean speedup of 20.6× Nios at an area overhead of 4.0×.

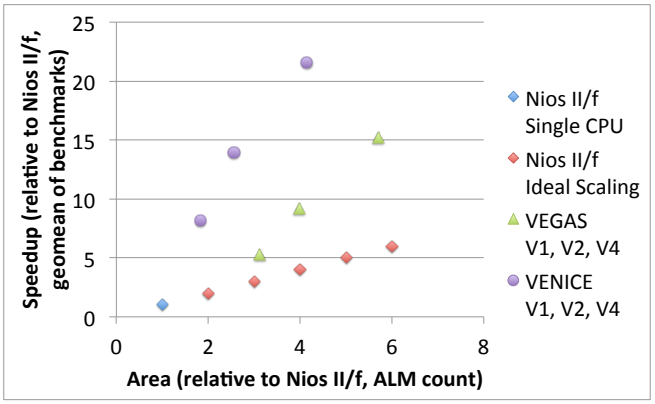

Figure 6. Speedup (geomean of 9 Benchmarks) vs Area Scaling

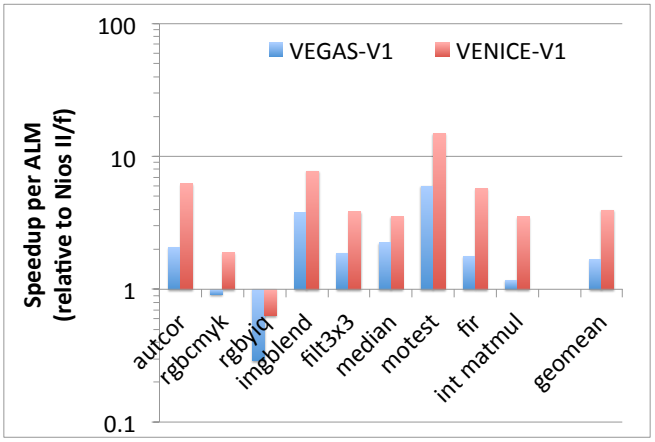

Figure 7. Computational Density with V1 SVPs

Figure 7 compares the computational density (speedup divided by area) for VEGAS and VENICE using a small V1 configuration. Simple benchmarks such as `rgbcmyk`, `rgbyiq`, `imgblend` and `median` achieve the smallest performance increase over VEGAS. These benchmarks have large vector lengths and no misaligned vectors; the speedup comes mostly from clock speed increase. Convolution benchmarks like `fir` and `autocor` benefit from the lack of misalignment penalty. 2D vectors accelerate `autocor`, `motest`, and `fir`. On `matmul`, using 3D vectors and the accumulators achieves 3.2× the performance of VEGAS.

Table III
BENCHMARK PERFORMANCE AND PROPERTIES

Benchmark	Performance (Millions of elem. per second)				Speedup			Data Type		Benchmark Properties		
	Nios II/f	V1	V2	V4	V1	V2	V4	In/Out	Intermed.	Data Set Size	Taps	Origin
autocor	0.46	5.94	11.11	18.94	12.9	24.2	41.2	halfword	word	1024	16	EEMBC
rgbcmyk	4.56	17.68	21.41	22.72	3.9	4.7	5.0	byte		896×606		EEMBC
rgbyiq	5.20	6.74	11.09	15.61	1.3	2.1	3.0	byte	word	896×606		EEMBC
imgblend	4.83	77.63	145.57	251.18	16.1	30.1	52.0	halfword		320×240		VIRAM
filt3x3	2.11	16.82	26.95	36.42	8.0	12.7	17.2	byte	halfword	320×240	3×3	VIRAM
median	0.10	0.74	1.45	2.69	7.3	14.4	26.6	byte		128×21	5×5	custom
motest	0.09	2.37	4.18	6.29	27.4	48.2	72.4	byte		32×32	16×16	custom
fir	3.32	20.11	34.95	41.67	6.1	10.5	12.5	halfword		4096	16	custom
matmul	11.7	148.20	322.22	593.75	12.6	27.4	50.6	word		1024×1024		custom
				Geomean	7.95	13.8	20.6					

For one application, `rgbyiq`, the computational density falls below 1.0 on VENICE, meaning Nios II/f is better. This is because the area overhead of 1.8× exceeds the speedup of 1.3×. The limited speedup is due to a combination of memory access patterns (r,g,b triplets) and wide intermediate data (32b) to prevent overflows. However, on average, VENICE-V1 offers 3.8× greater computational density than Nios II/f, and 2.3× greater density than VEGAS-V1. For V4 VENICE (not shown), the computational density is 5.2× that of Nios II/f and 1.9× that of V4 VEGAS.

VII. CONCLUSIONS

This work has shown that an optimized soft vector processor can provide significant speedups on data parallel workloads. With a single ALU, VENICE achieves 7.95× the performance of Nios II/f. With just 4 ALUs, speedups over 70× were demonstrated. From V1 to V4 VENICE has 3.8× to 5.2× better performance per logic block than Nios II/f. The VENICE soft vector processor is also both smaller and faster than VEGAS, offering roughly 2× the performance per logic block while using fewer block RAMs and the same number of DSP blocks.

During our experiments, we discovered some limitations that should be addressed by future work. Proper gather/scatter support is needed for rearranging data (e.g., de-interleaving r,g,b triplets). These currently must be done with 2D instructions operating on one element per cycle and bottleneck certain benchmarks, such as `rgbyiq`.

The use of 2D and 3D instructions allows for high ALU utilization even with small vector lengths. Further speedups require either extensive rearranging of data with scatter/gather type operations or extending past the vector paradigm. VENICE has been conceived as a building block in a vector/thread hybrid solution [3]. Future work will attempt to simplify the programming of such a system and integrate the components to reduce the overhead of connecting multiple VENICE processors.

VIII. ACKNOWLEDGMENTS

The authors would like to thank NSERC, and VectorBlox Computing, Inc. for funding and Altera for donating DE4-530 development boards.

REFERENCES

[1] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. Lemieux. VEGAS: Soft vector processor with scratchpad memory. In *FPGA*, pages 15–24, 2011.

[2] C. Kozyrakis and D. Patterson. Scalable, vector processors for embedded systems. *Micro, IEEE*, 23(6):36 – 45, nov.-dec. 2003.

[3] R. Krashinsky, C. Batten, M.Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. In *ISCA*, June 2004.

[4] Z. Liu, A. Severance, S. Singh, and G. G. Lemieux. Accelerator compiler for the venice vector processor. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA '12, pages 229–232, New York, NY, USA, 2012. ACM.

[5] R. M. Russel. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.

[6] P. Yiannacouras, J. Rose, and J. G. Steffan. The microarchitecture of fpga-based soft processors. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '05, pages 202–212, New York, NY, USA, 2005. ACM.

[7] P. Yiannacouras, J. G. Steffan, and J. Rose. VESPA: portable, scalable, and flexible FPGA-based vector processors. In *CASES*, pages 61–70. ACM, 2008.

[8] J. Yu, C. Eagleston, C. H. Chou, M. Perreault, and G. Lemieux. Vector processing as a soft processor accelerator. *ACM TRETS*, 2(2):1–34, 2009.