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 clockspeed and eliminating bottlenecks in ALU utilization, VENICE can achieve over 2x better performance-per-logic block than VEGAS, the previous best SVP. While VENICE can scale to a large number of ALUs, a multiprocessor system of smaller VENICE SVPs is shown to scale better for benchmarks with limited inner-loop parallelism. VENICE is also simpler to program, as its instructions use standard C pointers into a scratchpad memory rather than vector registers.**

***Index Terms*—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 compute engine design.

To harness the parallelism of FPGAs, users often build custom datapath accelerators that can quickly execute a single application kernel. For complex applications, multiple such accelerators are likely to be required. To assist this process, there is a growing trend towards using C-to-hardware synthesis tools to build such accelerators. However, this approach suffers from significant drawbacks. Every kernel requires its own accelerator, and every C software change introduces a new place-and-route iteration. Hence, even simple changes can lead to dramatic shifts in area usage and clock frequency. All of these factors make design closure with custom datapath accelerators very difficult.

Instead of application-specific accelerators, 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. Alternatively, it enables larger multiprocessors to be built using VENICE, resulting in greater computational throughput for the fixed area of a specific device.

The key contributions of this work are:

- Removal of vector address register file (area)
- Use of 2D and 3D vectors (performance)
- Absolute value and accumulate stages (flexibility)
- Operations on unaligned vectors (performance)
- New vector conditional implementation (area)
- Streamlined instruction set (area)

Programming VENICE requires little specialized knowledge, utilizing the C programming 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. In particular, the removal of the vector address register file, streamlining of instructions, and new conditional support makes VENICE easier to program than VEGAS. In addition, VENICE does not suffer a performance penalty when using unaligned operands, freeing programmers from worrying about the placement of data in memory.

Previous work describes a compiler for VENICE based on Microsoft Accelerator [5]. By contrast, this paper describes the design, architecture, native programming interface, and native programming results of VENICE. A shorter version of this paper was presented at the CARL 2012 Workshop. This version has an additional example, expanded background and architecture descriptions, and DCT multiprocessor case study.

II. Background and Related 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 [6] by sending a stream of values into a pipelined functional unit (FU) at a rate of one element per clock cycle. Here, an FU is

978-1-4673-2845-6/12/$31.00 2012 Crown

a fixed-function unit, such as an adder or multiplier, not a full-blown ALU. In the CRAY-1, parallelism is obtained through pipelining the FU, allowing high clock rates. It also overlaps execution of different FUs by vector chaining, where the output of one FU is passed directly to the input of another FU which is executing a subsequent vector instruction. Chaining requires complex register files with one write port per FU to support writeback each cycle, and one or two read ports per FU to supply operands each cycle.

Alternatively, several parallel ALUs can operate in lockstep SIMD mode to execute the same vector instruction, reducing the time to process a long vector. In this mode, multiple ALUs each write back to their own independent partition of the vector register file, so parallelism is achieved without the multiple write ports required by chaining [3].

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). Significant overhead comes from instructions to load/pack/unpack these short vectors and looping (incrementing, comparisons, and branches are still necessary). Intel's Sandy Bridge processors implement AVX, the latest version of SSE, with a maximum vector length of 256b (8x32b words). In contrast, the Intel Many Integrated Core (MIC) microarchitecture Xeon Phi supports 512b vector lengths (16x32b words).

There are two key 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 instead of complex pack/unpack instructions. For example, strided access simplifies columnwise traversal of a 2D array.

B. Soft Vector Architectures

The VIPERS soft vector architecture [10], [11] demonstrated that programmers can explore the area-performance tradeoffs of data-parallel workloads without any hardware design expertise. Based on results from three benchmark kernels, VIPERS provides a scalable speedup of 3–30× over the scalar Nios II processor. Additionally, a speedup factor of 3–5× can be achieved by unrolling the vector assembly code. VIPERS uses a Nios II-compatible multithreaded processor called UT-IIe [2], but control flow execution is hindered by the multithreaded pipeline. The UT-IIe is also cacheless; it contains a small on-chip instruction memory and accesses all data through vector read/write crossbars to fast on-chip memory. VIPERS instructions are largely based on VIRAM [3].

The VESPA soft vector architecture [7], [8] is a MIPS-compatible scalar core with a VIRAM [3] compatible vector coprocessor. The original VESPA at 16 lanes can acheive an average speedup of 6.3× over six EEMBC benchmarks. Furthermore, an improved VESPA achieves higher performance by adding support for vector chaining with a banked register file and heterogeneous vector lanes [9]. Over the 9 benchmarks tested, the improved VESPA averages a speedup of 10× at 16 lanes and 14× at 32 lanes. The MIPS core uses a 4kB instruction cache, and shares a data cache with the vector coprocessor. Smaller (1- or 2-lane) vector coprocessors use an 8kB data cache, while larger ones use 32kB.

The VEGAS soft vector architecture [1] uses a Nios II/f with a soft vector accelerator. Each vector instruction is encoded into the free bits of a Nios custom instruction. Instead of vector data registers, VEGAS uses an 8-entry vector address register file that points into a large, banked scratchpad memory. 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 the scratchpad memory, since smaller operands are not expanded to fill an entire word as done with prior architectures.

Both VIPERS and VESPA also share a similar memory-system design. They use vector load and vector store instructions to transfer blocks of data, with optional strides, from main memory to a vector data register file. Separate read and write crossbars shuffle data during loads and stores.

In contrast, VEGAS does not support vector load and store instructions. Instead, it uses DMA block-read and block-write commands to copy between the scratchpad and main memory. All vector alignment / byte shuffling is done at runtime in the writeback pipeline stages by passing vector results through a shuffle network. Except for lengthening the pipeline, there is no run-time penalty if the destination vector is misaligned. However, if vector source operands are misaligned, VEGAS must first copy one operand to a temporary location in the scratchpad. This extra copy operation roughly cuts performance in half.

III. VENICE Architecture

A block diagram of the VENICE (Vector Extensions to NIOS Implemented Compactly and Elegantly) architecture is shown in Figure 1. Similar to previous soft vector processors (SVPs), VENICE requires a scalar core as the control processor; in this case, a Nios II/f executes all control flow 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.

VENICE vector instructions are encoded into one or two tandem Nios custom instructions and placed inline with regular instructions. These custom instructions are written into a 4-entry vector instruction queue. Typically, this allows the Nios core to continue executing its next instruction. However, it will stall if the FIFO is full, or if the custom instruction synchronizes by explicitly waiting for a result.

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

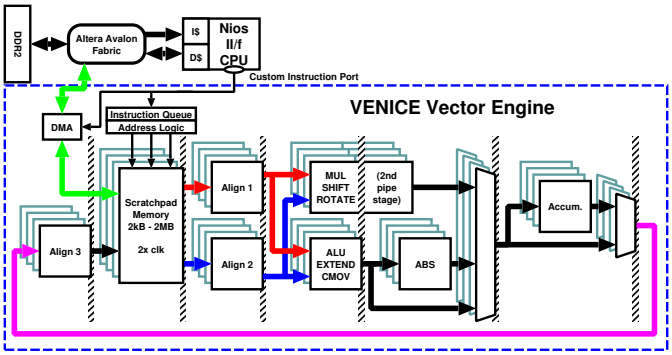

Fig. 1. VENICE Architecture (vertical gray bars are pipeline stages)

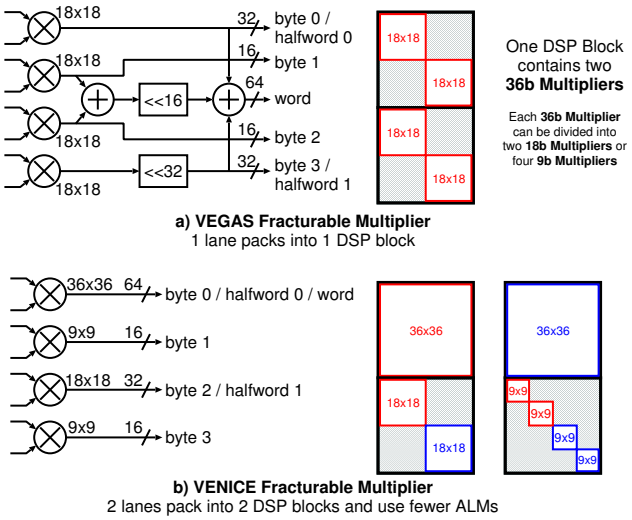

Fig. 2. Fracturable Multiplier Styles

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. 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 additional parallelism. Each 32-bit ALU supports subword SIMD operations on halfwords or bytes, thus doubling or quadrupling the parallelism 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). This also allows the SVP to run synchronously at the same full clock rate as the Nios II/f, simplifies control, adds the predictablity of a fully synchronous design, and reduces resource usage.

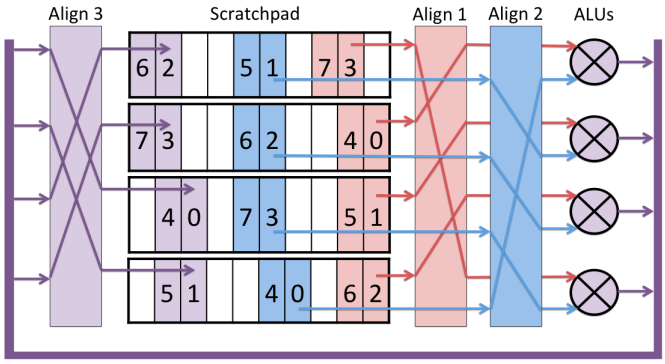

Fig. 3. Example of Misaligned Operation

A. Removal of Vector Address Register File

One major difference from VEGAS is removal of the vector address register file (VARF). In VEGAS, the vector instructions emitted from the scalar core include two source register numbers and one destination register number. They are indices into the VEGAS 8-entry VARF. The result is a rather large structure built from ALMs and FFs.

Instead, VENICE relies upon the Nios register file to contain the vector addresses. Instead of emitting register numbers, the vector instruction is emitted with two source addresses and one destination address; these addresses directly index the scratchpad. Nios instructions can run concurrently with vector operations, calculating addresses for the next vector instruction while the current one executes.

In VEGAS, tracking and spilling values in the VARF is a cumbersome task. In VENICE, many ALMs are saved, and there is no need to do this extra work. The lack of VARF also makes the job of writing a compiler for VENICE much easier.

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

Some benchmarks are limited by instruction dispatch rate, especially when short vectors are used. This makes it difficult to achieve high vector ALU utilization.

To get around this limitation, 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, which can 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. Each of the two source operands is aligned into a canonical position such that element 0 is moved to the top position. The result is then re-aligned, moving element 0 into the target position. This is especially useful when doing convolution, as one operand's starting address is continuously changing and seldom aligned.

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 to save area.

The older VEGAS processor uses a single alignment network because this is the only component that grows superlinearly with the number of vector lanes. In VENICE, where the number of vector lanes is typically small, this is not an issue. However, a single alignment network introduces a performance penalty when input operands are not aligned: a copy operation needs to be inserted, halfing performance.

D. New Vector Conditional Operations

VENICE takes a novel approach to data-conditional operations. Vector flags for compare instructions and arithmetic overflow are written alongside each byte, using the 9th bit in BRAMs that have an extra data bit for parity or optional storage (such as the M9K in Altera Stratix devices). This reduces BRAM usage, but does not allow for a VIRAM-style register file of flags. Instead, conditional move operations that read the flag bits as one of the input operands are used to implement simple predication.

The stored flag value depends upon the operation: unsigned addition/subtraction stores the carry-out/borrow, while signed addition/subtraction stores the overflow. The flag and MSB result bits can thus be used to check out-of-range results, less-than/greater-than (using the negative-result flag), or extended precision (64-bit) arithmetic.

E. Streamlined Instruction Set

VENICE implements a simple instruction set, consisting of 24 basic operations versus over 50 instructions in VEGAS. VENICE also has several mode bits to indicate more complex operations. Because two Nios custom instructions are required to dispatch a VENICE instruction, there are enough usable bits to encode the ISA while allowing any combination of options. Absolute value, 2D/3D instructions, and accumulation reduction can all be combined with any instruction and each other. This allows for complex instructions to be built that perform multiple operations between reading and writing to the scratchpad.

F. FPGA Architecture-Specific Optimizations

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 in Stratix IV FPGAs.

The only drawback of the new multiplier organization is that a single lane's multipliers cannot be packed into a single Stratix IV DSP Block due to configuration limitations. However, two lanes worth of multipliers may be packed into two DSP Blocks. Hence, V1 may use an additional half DSP Block, but larger designs use the same amount as VEGAS.

The multiplier (which also performs shift/rotate) requires two cycles operational latency at high freqencies. This leaves an extra cycle for processing simple (non-multiplier) ALU operations which can complete in a single cycle. A general absolute value stage was added after the integer ALU, allowing operations such as absolute difference in a single instruction. When followed by VENICE's reduction accumulators, operations such as sum-of-absolute-differences (used for motion estimation) and multiply-accumulate instructions (for matrix multiply) can be implemented in a single instruction.

V. Native Programming Interface

The native VENICE application programming interface (API) is similar to inline assembly in C. However, the C macros are used to simplify programming and make VENICE instructions look like C functions without any run time overhead. 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 emits a queued operation may return immediately before the operation is finished. Some macros are used to restore synchrony and explicitly wait until the vector engine or DMA engine is finished.

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, VD, VA, VB)`, where the values of `MODE` and `FUNC` must be predefined symbols, while the values of VD and VB must be scratchpad pointers and VA 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);`.

```
#include "vector.h"

int main()
{
    const int length = 8;
    int A[length] = {1,2,3,4,5,6,7,8};
    int B[length] = {10,20,30,40,50,60,70,80};
    int C[length] = {100,200,300,400,500,600,700,800};
    int D[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();
}
```

Fig. 4. VENICE Native API to Add 3 Vectors

```
int     num_taps, num_samples;
int16_t *v_output, *v_coeffs, *v_input;

// Set up 2D vector parameters:
vector_set_vl( num_taps  );             // inner loop count
vector_set_2D( num_samples,             // outer loop count
               1*sizeof(int16_t),               // dest gap
               (-num_taps  )*sizeof(int16_t),   // srcA gap
               (-num_taps+1)*sizeof(int16_t) ); // srcB gap

// The instruction below repeats a 1D dot-product (ie,
// 1D accumulate) operation once for every output sample.
// After each 1D dot product, it adds the gap values (set
// by vector_set_2D) to each of the three pointers.
vector_acc_2D( VVH, VMULLO, v_output, v_coeffs, v_input );
```

Fig. 5. FIR Kernel Using 2D Vector Instructions

In the example, the MODE specifier of `VVBU` refers to a 'vector-vector' operation (`VV`) on byte-size data (`B`) that is unsigned (`U`). The vector-vector part can instead be scalar-vector (`SV`), where the first source operand is a scalar value provided by Nios instead of an address. These may be combined with data sizes of bytes (`B`), halfwords (`H`) and words (`W`). A signed operation is designated by (`S`) or by simply omitting the unsigned specifier (`U`). For example, computing a vector of signed halfwords in `vresult` by subtracting a vector `vinput` from the scalar variable k would be written as `vector( SVH, VSUB, vresult, k, vinput)`.

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 scratchpad 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.

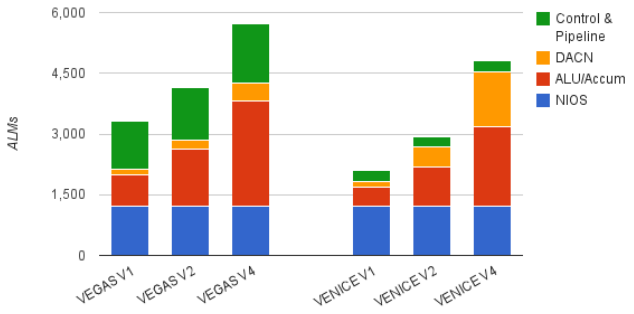

Fig. 6. Area Savings Breakdown (ALMs)

Figure 5 gives an example of using 2D vector instructions: a kernel that performs a FIR filter with 16-bit input and output data. This function can be performed in a single VENICE instruction using the reduction accumulators. Without 2D instructions, the scalar processor must repeat a 1D vector instruction (of `num_taps` length) using a `for()` loop of length `num_samples`. This requires the scalar processor to increment the source and destination addresses and count the number of iterations.

With 2D instructions, the loop counting and operand address incrementing is performed in hardware. The `vector_set_2D()` call specifies the number of iterations (first parameter), and three gaps for the destination and two source operands, respectively. The gaps are a distance, measured in bytes, between the end of the row and the start of the next row. Hence, a gap of 0 implies a packed 2D matrix, while a negative gap produces a sliding window effect as required by the FIR filter. The gap for operand A, the FIR coefficients, produces a stationary window. The destination gap is 2 bytes because each 1D operation is accumulated, producing a result row which contains just 1 element (i.e., the dot product).

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 18b multipliers or four 9b multipliers. Altera

TABLE I
RESOURCE USAGE COMPARISON

Device or CPU	VEGAS				VENICE			
	ALMs	DSP Blocks	M9Ks	F_{max}	ALMs	DSP Blocks	M9Ks	F_{max}
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

literature usually quotes device capacity as the number of 18b multipliers, which is 8 times the number of DSP Blocks.

Table I also gives area and clock frequency results for several VEGAS and VENICE configurations, each of which includes a Nios II/f base processor. The V1, V2, and V4 notation indicates one, two and four 32b vector lanes, respectively. The (8kb), (16kB), and (32kB) notation indicates the scratchpad sizes used in each configuration. VENICE uses fewer ALMs, and fewer M9Ks, than VEGAS across the board. Except for the smallest V1 case, VEGAS and VENICE use the same number of DSP Blocks. In the V1 case, VENICE is using 2.5 DSP Blocks, but these are not filled to capacity; in addition to the 0.5 DSP block being available, there is also room for one 18b and two 9b multipliers. The clock frequency achieved with VENICE is also 50% higher than VEGAS. Except for the V1 DSP block anomaly, VENICE completely dominates VEGAS in area and clock frequency.

Figure 6 gives a more detailed area breakdown of VEGAS and VENICE. VENICE has consistently lower area 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 is primarily due to the new parallel multiplier, which uses fewer adders and requires less input and output multiplexing. Additional savings was obtained by streamlining the ALU operations. Note that this savings is per lane.
- Control/Pipeline savings is primarily due to removal of the 8-entry vector address register file. Since each operand must support an auto-increment mode, VEGAS implemented these in ALMs and FFs.
- The VEGAS DMA engine includes an alignment network, which allows data to be loaded into an aligned position and avoid the misalignment performance penalty. This is not necessary in VENICE, since it runs full-speed on unaligned data.
- The VEGAS alignment network was designed to scale to a large number of lanes, while the VENICE alignment network was optimized for only a few lanes.

B. Benchmark Performance

The characteristics of nine application kernels are reported in Table III. The input and output data types for each kernel are shown, including whether a higher precision is used during intermediate calculations, in the Data Type column. The overall input data set size is also shown, along with the size

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

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. Note that the motest application only computes the set of SAD values, it does not find the lowest SAD location.

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.

The performance of these kernels is shown in Table III. The results are in units of millions of elements computed per second. For the first eight kernels, an element is in units of the output data type. This works because the amount of compute work is linear to the number of output bytes. For example, the motest kernel computes 0.09 million SAD calculations per second, which each calculation results in one byte of output. Matrix multiply, however, does not require linear computations per output element. As a result, for that kernel only, 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.

C. Speedup versus Area

The VENICE processor is designed to offer higher performance and use less area than VEGAS. Figure 7 demonstrates this with a speedup versus area plot. The speedup and area axes are normalized to the Nios II/f results. Speedup results are a geometric mean of the execution times for nine benchmark programs. Area results account for ALMs only.

Results for a single Nios II/f processor are presented at the <1.0,1.0> position on the graph. Diamond-shaped data points extrapolate the performance and area if multiple Nios processor instances are created (with no interconnect overhead) and run perfectly in parallel. Triangular-shaped data

TABLE III
BENCHMARK PERFORMANCE AND PROPERTIES

Benchmark	Performance (Millions of elem. per second) Nios II/f	V1	V2	V4	Speedup V1	V2	V4	Data Type In/Out	Intermed.	Benchmark Properties 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					

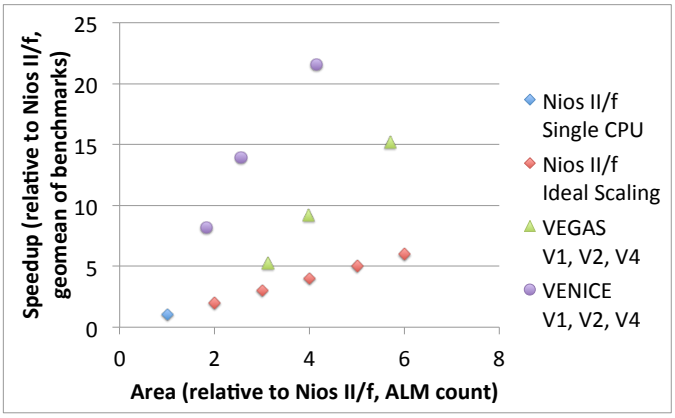

Fig. 7. Speedup (geomean of 9 Benchmarks) vs Area Scaling

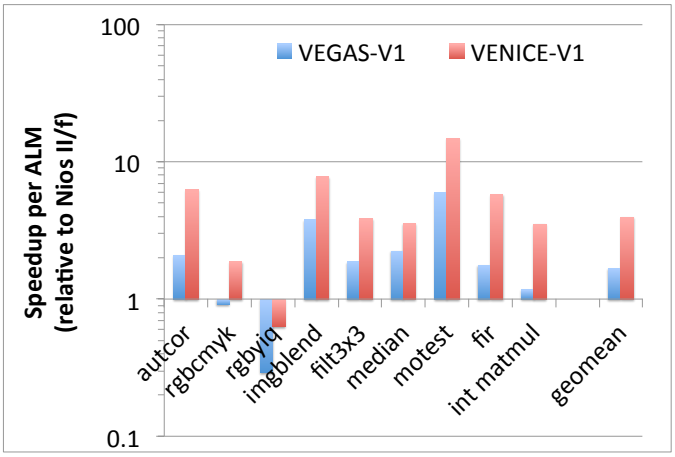

Fig. 8. Computational Density with V1 SVPs

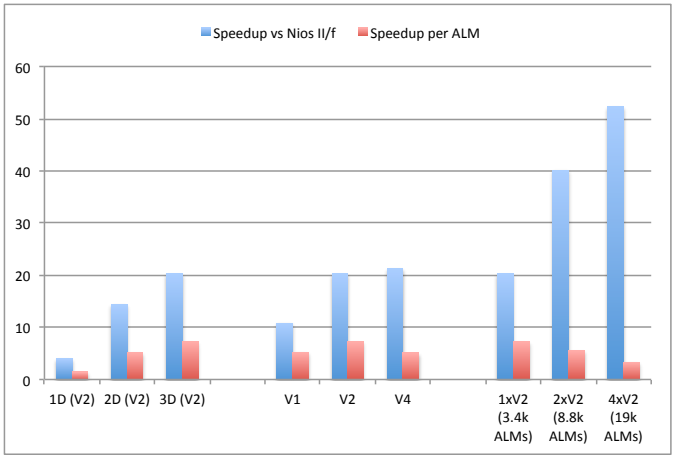

Fig. 9. 16-bit 4x4 DCT varying 2D/3D Dispatch, SVP Width, and # of SVPs

points for V1, V2, and V4 configurations of VEGAS clearly outperform Nios II/f, achieving a maximum speedup of 17.2× at an area overhead of 5.7×. Circular-shaped data points for similar configurations of VENICE show it dominates VEGAS in both area and speed, achieving a maximum speedup of 20.6× at an area overhead of 4.0×.

Speedup divided by area produces a metric of performance per unit area. This can also be considered a measure of the computational density of an FPGA, which has a fixed total area. Figure 8 compares the computational density 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, and so the speedup comes mostly from clock speed increase. Convolution benchmarks like fir and autocor benefit from the lack of misalignment penalty on VENICE. The 2D vectors accelerate autocor, motest, and fir. On matmul, using 3D vectors and the accumulators achieves 3.2× the performance of VEGAS.

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.

Comparing V4 configuration results (not shown), the computational density of VENICE is 5.2×, while VEGAS is 2.7× that of Nios.

D. Case Study: DCT

VENICE was designed to exploit vector parallelism, even when vectors are short. By remaining small, VENICE can be efficiently deployed in multiprocessor systems to efficiently exploit other forms of parallelism (eg, thread-level) on top of the vector-level parallelism.

In this section, we use VENICE to perform a 4x4 DCT with 16-bit elements on a total of 8192 different matrices. Each DCT is implemented using two matrix multiplies followed by shifts for normalization. Vectors are short, limited to four halfwords for the matrix multiply.

In Figure 9, the first set of bars shows the benefit of using 2D and 3D vector operations with a V2 VENICE configuration. In the 1D case, VENICE is issue-rate limited and gets poor speedup. To compute one element in the output matrix (i.e., compute the dot product of one row with a transposed row), the Nios processor must compute the addresses and issue two custom instructions. Still, it manages to get a speedup of 4.6× the Nios II/f. With 2D instructions, an entire row in the output matrix can be computed with a single VENICE instruction, giving a speedup of 3.6× over the 1D case. With 3D instructions, it can compute the entire output matrix with one instruction, running 1.4× faster than the 2D case. With 3D instructions, one DCT takes 43.9 cycles on average. The theoretical minimum is 40 cycles, giving 91% ALU utilization.

The second set of bars shows performance scaling from one to four vector lanes. A V1 VENICE can achieve a speedup of 10.8× Nios II/f, and the benchmark scales well to V2 (1.87× faster than V1). However, V4 is only 1.05× faster than V2. During the matrix multiply step, the maximum vector length is 4 halfwords, so it does not benefit from more than 2 lanes; only the normalization step benefits.

The final set of bars shows the results of a simple multiprocessor VENICE system, consisting of one, two, and four V2 VENICE processors. This system was constructed by connecting together several Nios II/f processors, with VENICE accelerators, using the default Altera Avalon fabric. Scaling from one to two processors is 1.98× faster. The next doubling to four processors achieves a speedup of only 1.31×. This is lower than expected, because we are not yet bandwidth-limited (the memory bandwidth limit is 8 cycles per DCT, but we are achieving only 17.0 cycles per DCT at this point). Further investigation is required.

The set of red bars in the figure indicates speedup per ALM. The multiprocessors use a lot of additional ALMs in the Avalon fabric. This greatly deteriorates the speedup-per-ALM advantage, and suggests that Avalon is not the most efficient way to build a VENICE multiprocessor.

VII. Conclusions

This work has shown that an optimized soft vector processor can provide significant speedups on data parallel workloads. Speedups over 70× a Nios II/f were demonstrated, with 3.8× to 5.2× better performance per logic block from V1 to V4.

The VENICE soft vector processor is also both smaller and faster than VEGAS, offering over 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 scatter/gather support is needed for rearranging data (e.g., de-interleaving r,g,b triplets). These are currently done with 2D instructions operating on one element per cycle and bottleneck certain benchmarks, such as `rgbyiq`. Data conversion for benchmarks that need higher internal precision is also done with 2D instructions; future work will explore options for converting between types in the DMA engine or the vector pipeline to eliminate this bottleneck.

The use of 2D and 3D instructions allows for high ALU utilization (91% in DCT) 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 [4], and multiprocessor VENICE results show that it can be used as such. 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 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] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown. A multithreaded soft processor for SoPC area reduction. In *FCCM*, pages 131–142, 2006.

[3] C. Kozyrakis. *Scalable Vector Media Processors for Embedded Systems*. PhD thesis, University of California at Berkeley, May 2002. Technical Report UCB-CSD-02-1183.

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

[5] 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.

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

[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] P. Yiannacouras, J. G. Steffan, and J. Rose. Data parallel FPGA workloads: Software versus hardware. In *FPL*, pages 51–58, Progue, Czech Republic, 2009.

[9] P. Yiannacouras, J. G. Steffan, and J. Rose. Fine-grain performance scaling of soft vector processors. In *CASES*, pages 97–106. ACM, 2009.

[10] 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.

[11] J. Yu, G. Lemieux, and C. Eagleston. Vector processing as a soft-core CPU accelerator. In *FPGA*, pages 222–232, Monterey, California, USA, 2008.