VENICE: A Compact Vector Processor for FPGA Applications

Aaron Severance Guy Lemieux
University of British Columbia, Vancouver, Canada

VENICE (Vector Extensions to NIOS Implemented Compactly and Elegantly) is a SVP (soft vector processor) intended to accelerate computationally intensive applications implemented on an FPGA. SVPs are exclusively for FPGAs, targeted at the productivity gap between writing custom hardware in an HDL and writing software for a soft processor in FPGA-based applications. They provide the convenience of software programming and software compile times, and yet they can achieve over 200x speedup compared to a scalar soft processor [1]. Unfortunately, a large speedup requires very wide SVPs running on highly regular data-parallel applications. In contrast, VENICE is designed to be highly efficient with shorter vectors, optimized to be a building block in systems that support multiple hybrid forms of parallelism. It is partly inspired by the vector-thread architecture [2], but designed entirely for an FPGA environment.

In our earlier SVP work, VEGAS, we noticed significant speedups (up to 20x on motion estimation) using just a single vector lane (one 32-bit ALU). This was obtained by eliminating inner loop overhead. Traditionally, this overhead is primarily counting and conditional branches, but in VEGAS the load & store instructions are also eliminated using memory-to-memory vector instructions operating upon a private, on-chip scratchpad. We also noticed significant improvement by fracturing each 32b ALU into multiple 16b or 8b ALUs, the same technique used in most SIMD instruction set extensions. As a result, most applications obtain best area-delay product (best throughput per unit of area) using a relatively small number of vector lanes. For VEGAS, this was often 2 to 8 vector lanes of 32b-ALUs, aka V2 to V8.

VENICE aims to provide more SVP instances per chip, improving the ability to exploit task- or thread-level parallelism. VENICE has both modified the architecture and optimized the implementation of the VEGAS SVP for improved throughput, by both increasing performance and lowering area overhead. Three key properties inherited from VEGAS (not present in other SVPs [3]) are (i) the scratchpad memory, (ii) the fracturable 32b ALUs, and (iii) the DMA programming model. Instead of a cache, the scratchpad is an addressable vector-data store, wherein all vector instructions operate memory-to-memory. It is highly efficient for on-chip FPGA storage because it can be easily 2X overclocked for more r/w ports, and 8b data does not need to be sign-extended to 32b as in traditional register files. The fracturable ALUs provide better throughput on small operands, and the DMA programming model is similar to CUDA/OpenCL models which explicitly copy blocks of data from host memory before operating. VEGAS and VENICE use asynchronous, double-buffered DMA to hide off-chip memory latency by overlap with computation.

VEGAS is designed for large widths: V32 beats an Intel Core2 at integer matrix multiply, and we have scaled up to V128. In contrast, VENICE presumes small vectors and is optimized for up to V4 only. This results in a number of architectural changes to save area. In particular, the dedicated vector address register file of VEGAS is deleted. Instead, the regular scalar register file entries provide the addresses. Multi-cycle vector operations give the scalar processor time to increment or fetch new vector addresses in parallel. This ISA change makes programming VENICE even easier than VEGAS, as programmers can now directly specify regular C pointers as operands for each vector instruction, eliminating the need to allocate, set, and use vector address registers as operands. VENICE also conveniently generalizes a vector into a 2D submatrix, where the global <vector-length> register is instead replaced with a <row-length, row-stride, row-count> triplet; each of the 2 source operands and destination operand have unique row strides.

VENICE also contains implementation-aware optimizations. The VEGAS ALU used extra ALMs to conserve FPGA multiplier blocks; VENICE does the opposite because ALMs are more preciously needed by application logic. The pipeline depth was lengthened and rebalanced, so it now matches the NIOS clock speed; this saves clock-domain crossing FIFOs and reduces communication latency between scalar and vector cores. Overall, VENICE uses ~20% fewer on-chip memory blocks. Since V4 is the maximum size, VENICE uses 3 separate interconnection networks (ICN) for performance to align 2 sources and 1 destination; in contrast, VEGAS shares 1 large ICN, thus it runs slower on unaligned vector operations.

Together, these improvements make VENICE a much better building block for a multiprocessor-based system to support thread- and task-level parallelism. A single-lane (V1) VENICE offers 2.5x better area-delay product than V1-VEGAS and 4.7x better than Nios II/f alone. In addition to future area and speed improvements, we are working on using OpenCL for exposing thread-level parallelism.

REFERENCES

[1] C. Chou, A. Severance et al, "VEGAS: Soft Vector Processor with Scratchpad Memory", FPGA2011.
[2] R. Krashinsky, K. Asanovic et al, "The Vector-Thread Architecture", ISCA2004.
[3] P. Yiannacouras, J.G. Steffan, J. Rose, "VESPA: Portable, Scalable and Flexible FPGA-based Vector Processors", CASES 2008.

VENICE Overview

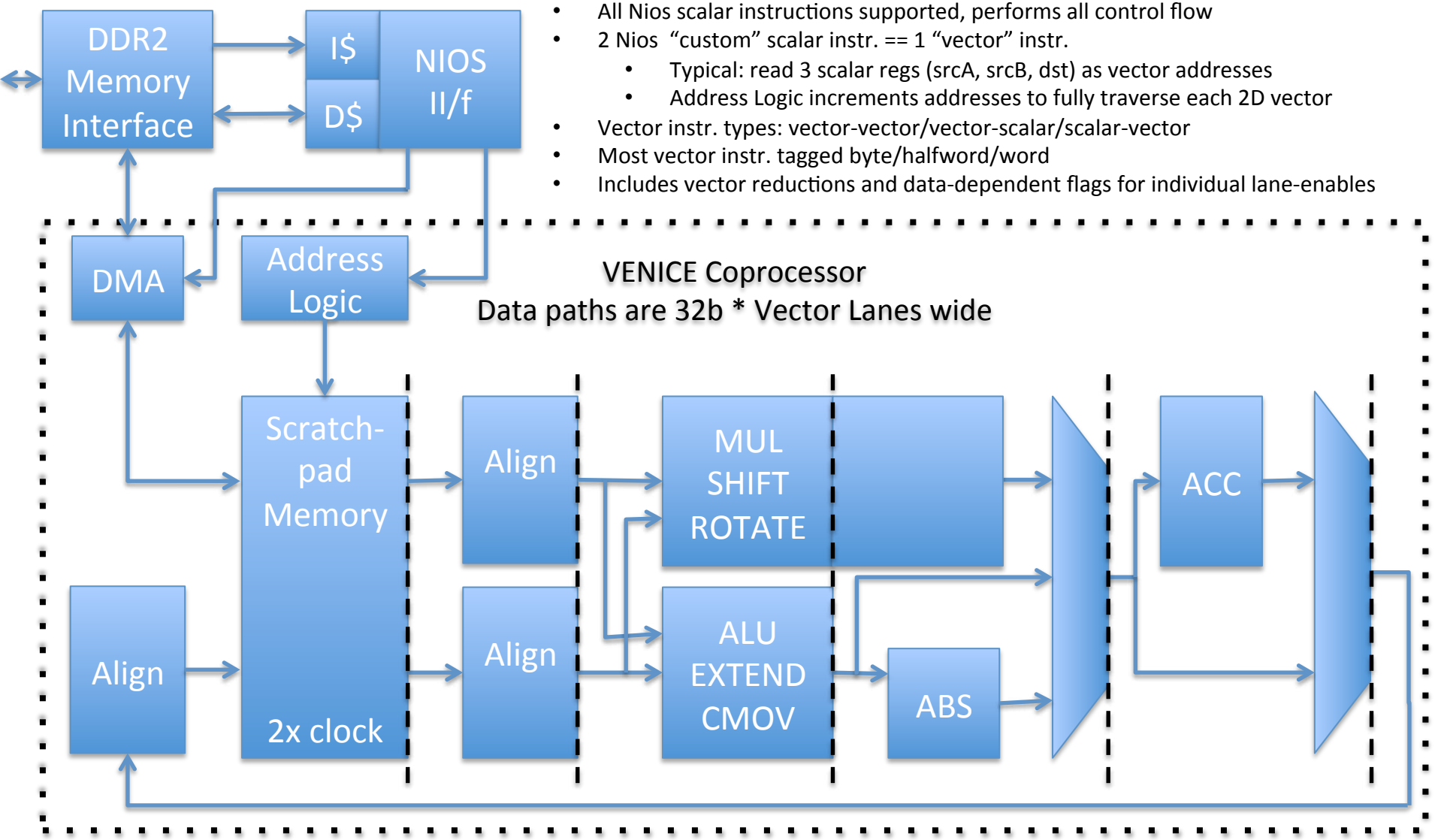
Vector Extensions to NIOS Implemented Compactly and Elegantly

- Soft Vector Processor (SVP) for use on FPGAs
  - Builds on previous VEGAS coprocessor
    - Scratchpad & DMA reduces data redundancy & block RAM needed
    - Intra-lane SIMD accelerates byte/halfword instructions
  - Optimized for Small Area, High Performance up to 4 Vector Lanes
    - Architectural changes: remove vector address register file, add more ICNs
    - Changed ISA encoding, major changes to ISA dispatch (single clock domain)
    - Aimed at future work on task-level parallelism and hybrid SIMT/vector
  - Simplified programming
    - C pointers used to index into scratchpad
    - 2D vectors for multimedia/linear algebra/etc.
- Approximate comparison to ARM/Intel…
  - 40nm ARM A9 with NEON SIMD, dual-core, 6.7mm^2, up to 2.0GHz
    - Peak multiply-accumulate throughput = 2 cores * 4 NEON ops/core * 2.0GHz = 16 GOP/s
  - 40nm, Stratix IV (~600mm^2 die), VENICE-V1/V2/V4 approx. 12.7/9.4/7.7mm^2 per lane, up to 200MHz
    - Largest FPGA fits up to 47/32/19 VENICE-V1/V2/V4 cores, giving 47/64/78 total vector lanes
    - Peak multiply-accumulate throughput = 78 lanes * 200MHz = 15.6 GOP/s
    - Our integer matrix multiply code is simple and achieves 78% of peak throughput on one VENICE-V4 core
  - 65nm, Stratix III, 100MHz VEGAS-V32 is 30% faster than 2.66GHz Intel Core2 (single core, not using SSE/MKL) at integer matrix multiply (IMM) using 4096x4096 matrices (192MB working set)
    - Intel version of IMM was tuned and cache-aware (tiled, loop reordered, etc)
    - VENICE-V4 has half the area-delay product of VEGAS-V4 on IMM
    - A multiprocessor-based VENICE-V4 should vastly outperform a single VEGAS-V32

VENICE Architecture

Single instruction stream:

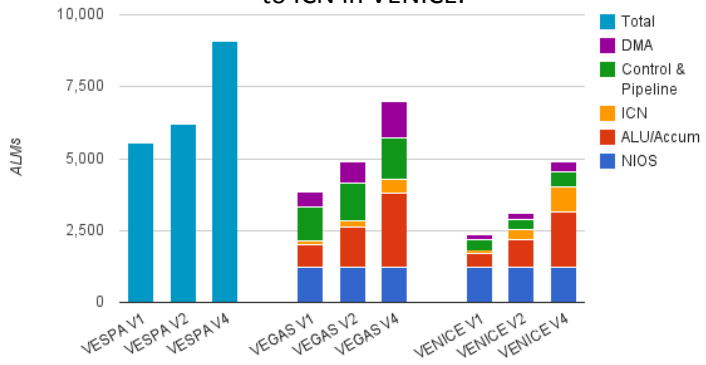
- All Nios scalar instructions supported, performs all control flow
- 2 Nios "custom" scalar instr. == 1 "vector" instr.
  - Typical: read 3 scalar regs (srcA, srcB, dst) as vector addresses
  - Address Logic increments addresses to fully traverse each 2D vector
- Vector instr. types: vector-vector/vector-scalar/scalar-vector
- Most vector instr. tagged byte/halfword/word
- Includes vector reductions and data-dependent flags for individual lane-enables



Area and Speed Comparisons

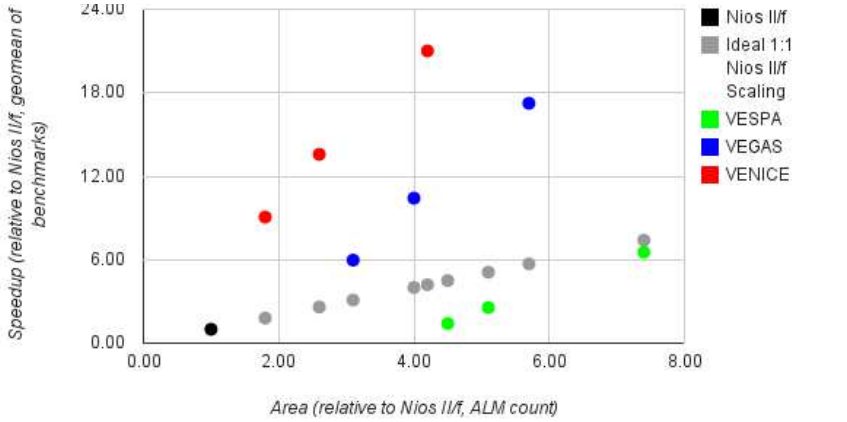
VENICE area greatly reduced.

Note: some DMA area in VEGAS moved to ICN in VENICE.



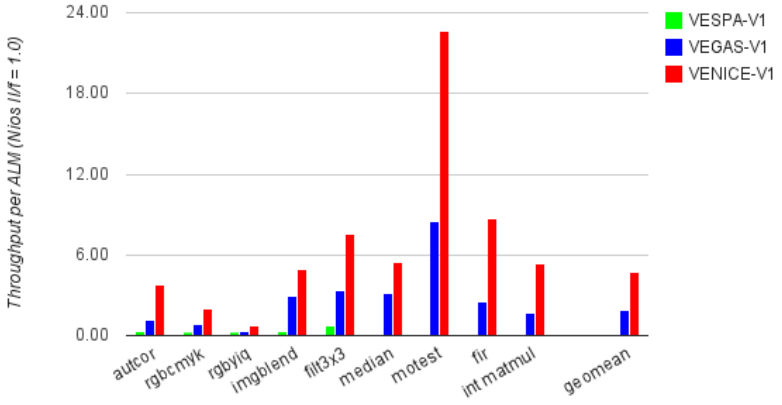
VENICE is faster and uses less area than VEGAS or VESPA (for each of V1, V2 and V4).

Note: speedup here is geomean of only autcor, rgbcmyk, rgbyiq, imgblend, filt3x3.



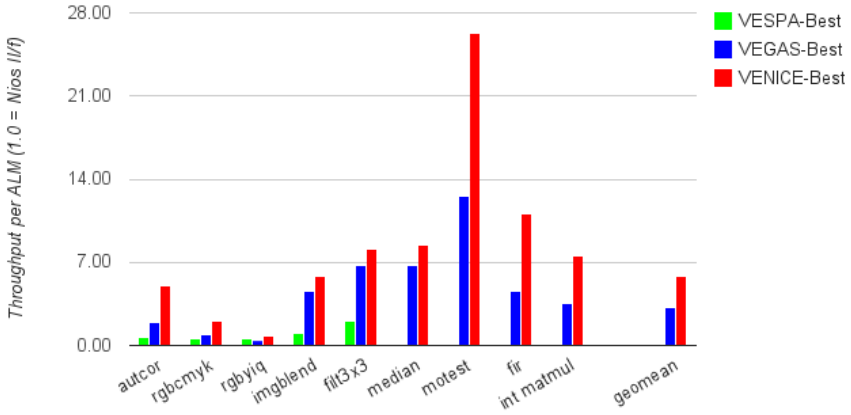
VENICE with V1 (single lane) 4.7x better area-delay product over Nios II/f

Note: area is based on ALM count only.



Best VENICE 5.8x better A*D than Nios II/f

"Best" = best A*D of V1/V2/V4 on per-application basis



Code Sample – Blocked and Transposed FIR

```
// Common starting code. The code below is single-buffered, but invariant code is placed inside while() loop for double-buffered version.
void vector_fir( input_type *input, output_type *output, input_type *coeffs, int sample_size, int num_taps )
{
    //5 vectors size chunk_size+scratchpad+padding, round to 1/8th vector memory
    int chunk_size    = (VEGAS_VECTOR_MEMORY_SIZE >> 3)/OUTPUT_WIDTH;

    input_type  *sample_on_vpu = (input_type  *)vegas_malloc( (         chunk_size+num_taps)*INPUT_WIDTH  );
    output_type *mult          = (output_type *)vegas_malloc( (         chunk_size+num_taps)*OUTPUT_WIDTH );
    output_type *dest_on_vpu   = (output_type *)vegas_malloc( (num_taps+chunk_size+num_taps)*OUTPUT_WIDTH );
    dest_on_vpu  += num_taps;

    int j, chunk_start = 0;
    while( chunk_start < sample_size ) {
        vegas_dma_to_vector( sample_on_vpu, input+chunk_start, (chunk_size+num_taps)*INPUT_WIDTH );
        vegas_wait_for_dma();
```

```
        // VEGAS code (old architecture, old style)
        vegas_set( VCTRL, VL, chunk_size+num_taps ); // sets vector length
        vegas_set( VADDR, V1, sample_on_vpu );
        vegas_set( VADDR, V2, mult );
        vegas_set( VADDR, V4, dest_on_vpu-1 );
        vegas_set( VADDR, V5, dest_on_vpu );
        vegas_set( VINCR, V4, 0-OUTPUT_WIDTH ); // autoincr amount
        vegas_set( VINCR, V5, 0-OUTPUT_WIDTH );
        vegas_vsh( VMULLOU, V5INC, V1, coeffs[0] );
        for( j = 1; j < num_taps; j++ ) {
            vegas_vsh( VMULLOU, V2   , V1,    coeffs[j] );
            vegas_vvh( VADDU,   V5INC, V4INC, V2        );
        }
```

```
// VENICE code (new architecture, new style)
output_type *temp_dest = dest_on_vpu;
venice_set_vl( chunk_size+num_taps ); // sets vector length

venice( SVHU, VMULLO, temp_dest, coeffs[0], sample_on_vpu );

for( j = 1; j < num_taps; j++ ) {
    temp_dest--;
    venice( SVHU, VMULLO, mult,  coeffs[j], sample_on_vpu );
    venice( VVHU, VADD, temp_dest, temp_dest, mult );
}
```

```
// Common tail code

        vegas_wait_for_dma();
        vegas_instr_sync();

        vegas_dma_to_host( output+chunk_start, dest_on_vpu, chunk_size*OUTPUT_WIDTH );
        chunk_start += chunk_size;
    }

    vegas_free();
    vegas_wait_for_dma();
}
```