VEGAS: Soft Vector Processor with Scratchpad Memory

Christopher H. Chou
cchou@ece.ubc.ca

Aaron Severance
aaronsev@ece.ubc.ca

Alex D. Brant
alexb@ece.ubc.ca

Zhiduo Liu
zhiduol@ece.ubc.ca

Saurabh Sant
ssant@ece.ubc.ca

Guy Lemieux
lemieux@ece.ubc.ca

Dept. of Elec. and Comp. Eng.
Univ. of British Columbia
Vancouver, Canada

ABSTRACT

This paper presents VEGAS, a new soft vector architecture, in which the vector processor reads and writes directly to a scratchpad memory instead of a vector register file. The scratchpad memory is a more efficient storage medium than a vector register file, allowing up to 9× more data elements to fit into on-chip memory. In addition, the use of fracturable ALUs in VEGAS allow efficient processing of bytes, halfwords and words in the same processor instance, providing up to 4× the operations compared to existing fixed-width soft vector ALUs. Benchmarks show the new VEGAS architecture is 10× to 208× faster than Nios II and has 1.7× to 3.1× better area-delay product than previous vector work, achieving much higher throughput per unit area. To put this performance in perspective, VEGAS is faster than a leading-edge Intel processor at integer matrix multiply. To ease programming effort and provide full debug support, VEGAS uses a C macro API that outputs vector instructions as standard NIOS II/f custom instructions.

Categories and Subject Descriptors

C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: Array and vector processors; C.3 [**Special-purpose and Application-based Systems**]: Real-time and Embedded systems

General Terms

Design, Experimentation, Measurement, Performance

Keywords

vector, SIMD, soft processors, scratchpad memory, FPGA

1. INTRODUCTION

FPGA-based embedded systems often use a soft processor for control purposes, but they use RTL blocks for performance-critical processing. Transferring some of these processing-intensive tasks onto a soft processor offers productivity, cost, and time-to-market advantages by reducing the amount of RTL design. In particular, soft processors allow algorithms to be easily modified without changing the FPGA bitstream, which could otherwise lead to convergence issues such as timing closure.

Unfortunately, traditional soft processors are too slow for most processing-intensive tasks. However, vector processing is known to accelerate data-parallel tasks. The VIRAM architecture [9] demonstrated that embedded tasks such as the EEMBC benchmark suite [1] can be accelerated with vectors. Embedded vector architectures SODA [12] and Ardbeg [18] were developed for low-power wireless applications which are also rich in data parallelism. VIRAM, SODA and Ardbeg were all developed for ASIC implementation, but the VESPA [20] and VIPERS [24, 23] processors demonstrate that soft vector architectures can be implemented efficiently and offer significant speedups on an FPGA as well.

This paper develops a new soft vector processor architecture called VEGAS. The two key distinguishing features of VEGAS are a cacheless scratchpad memory and the ability to fracture a 32-bit ALU into two 16-bit or four 8-bit ALUs at run-time. Combined, these two features make VEGAS more efficient with limited on-chip memory resources, allowing up to 9× more vector data elements to be stored on-chip and 4× more ALU engines. Benchmark results demonstrate that VEGAS offers up to 68% smaller area-delay product than VIPERS and VESPA, meaning it provides up to 3.1× the performance per unit area.

Scratchpads enable several performance enhancements. Instead of using traditional RISC-like vector load/store instructions, direct memory-memory operations are executed using the scratchpad. Source and destination operands are specified by vector address registers, each of which holds a scalar representing the starting address of the vector. To reduce loop overhead, VEGAS supports very large vector lengths, up to the full size of the scratchpad. Auto-increment of the address registers make it efficient to iterate through very long vectors or 2D arrays in a blocked fashion. Auto-increment allows very compact loops to achieve the same 3–5× performance advantage as the loop-unrolled examples in VIPERS, without the code bloat. Compact loops are also easier to program than unrolled loops. Finally, the scratchpad and external memory can tranfer data asynchronously with double-buffering to hide memory latency.

To conserve on-chip memory resources, VEGAS is cacheless and double-clocks the scratchpad, providing four

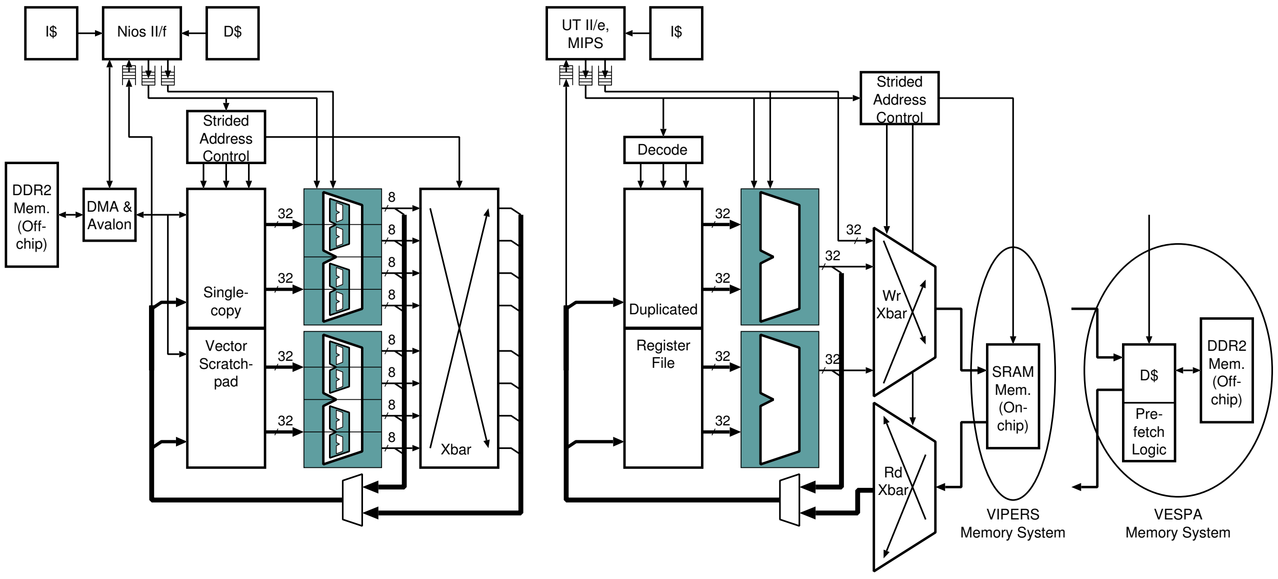

Figure 1: VEGAS (left) compared to VIPERS and VESPA (right)

read/write ports per cycle. Two read ports and one write port are used for vector instruction execution, while the fourth read/write port is used as a dedicated DMA channel between the scratchpad and off-chip memory. Efficient use of on-chip memory reduces the need to spill vector data to off-chip memory. This leads to significant performance advantages in some applications.

The VEGAS soft vector processor is attached to a standard Nios II/f. We encode each vector instruction as a single Nios II custom instruction by fitting the vector instruction bits into the available fields. Vector instructions are dispatched by Nios to a vector instruction queue. Since many vector instructions are multi-cycle, this allows Nios II to run ahead and asynchronously execute scalar code, such as loop bounds checking and control flow instructions. The Nios also issues scratchpad DMA transfer requests asynchronously as custom instructions. This means DMA transfers, vector instructions, and Nios instructions can all execute concurrently. A few special synchronization instructions cause Nios to wait, such as reading a scalar result from the vector scratchpad. The use of a standard Nios II/f gives users full debug capability using the Altera IDE as well.

A common concern for soft vector processors is compiler support. VESPA and VIPERS require hand-written or inline assembly code, translating vector instructions with a modified GNU assembler (gasm). Researchers have investigated the autovectorizing capability of gcc, but have not yet used it successfully [21]. Instead of an autovectorizing compiler, VEGAS uses C macros exclusively to emit Nios custom instructions on demand without modifying gcc or gasm. The macros are more readable, and the system is much simpler to program because the user doesn't need to track the scalar (control flow) values as register numbers. Instead, users track only vector scratchpad addresses stored in the vector address registers, and initiate DMA transfers explicitly. We also provide some convenience routines to simplify allocating and deallocating scratchpad memory. The macros are easier to program than pure assembly, and still gives explicit control over the hardware for maximum performance.

2. 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.

2.1 Vector Processing Overview

Classically, execution of a vector instruction is done by sending a stream of values into a pipelined ALU at a rate of one element per clock cycle. Parallelism is obtained through pipelining, allowing high clock issue rates. Additional parallelism is obtained by vector chaining, where the output of one ALU is passed directly to the input of another ALU which is executing a separate vector instruction. Chaining requires complex register files with multiple write ports to support writeback by several ALUs each clock cycle, and multiple read ports to feed several ALUs each cycle.

Alternatively, several ALUs can operate in lockstep SIMD mode to execute the same vector instruction, thus shortening 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 can be achieved without the multiple write ports required by chaining.

Modern microprocessors are augmented with SIMD processing instructions to accelerate data-parallel workloads. These operate on short, fixed-length vectors (e.g., only four 32-bit words). Significant overhead comes from instructions to load/pack/unpack these short vectors and looping.

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

2.2 Soft Vector Architectures

The VIPERS soft vector architecture [23, 24] 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. Moreover, an additional speedup factor of 3–5× can be achieved by fully unrolling the vector assembly code. VIPERS uses a Nios II-compatible multithreaded processor called UT-IIe [7], but control flow execution is hindered by the multithreaded pipeline. The UT-IIe is also cacheless; it contains a small, fast on-chip instruction memory and accesses all data through the vector read/write crossbars to fast, on-chip memory. VIPERS instructions are largely based on VIRAM [9].

The VESPA soft vector architecture [20, 21] is a MIPS-compatible scalar core with a VIRAM [9] compatible vector coprocessor. The original VESPA at 16 lanes can acheive an average speedup of 6.3× over six EEMBC benchmarks. Furthermore, VESPA demonstrated improved performance by adding support for vector chaining with a banked register file and heterogeneous vector lanes [22]. 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.

Both VIPERS and VESPA offer a wide range of configurability. For example, the parallel vector lanes can be specified at FPGA compile-time to be 8, 16 or 32 bits wide. However, when mixed-width data is required, the vector engine must be built to the widest data. Therefore, when processing smaller data, load instructions will zero-extend or sign-extend to the full width, and store instructions will truncate the upper bits. Since the vector register file must store all data (even byte-sized data) at the widest width, VIPERS and VESPA can be very inefficient: byte-wide data is stored in the on-chip main memory or data cache, then expanded to word-wide inside the registerfile. On top of that, to implement dual read ports, VIPERS and VESPA duplicate the vector register file. Hence, a single byte of data may occupy up to 9 bytes of on-chip storage.

Both VIPERS and VESPA also share a similar design for striding through memory. The vector register file is connected to an on-chip memory (VIPERS) or on-chip data cache (VESPA) through *separate* read and write crossbars. These crossbars are used when striding through memory during vector loads and stores; they must shuffle bytes/halfwords/words from their byte-offset in memory into word size at a new byte-offset in the vector register file. The size of the crossbars are constrained on one end by the overall width of the vector register file, and on the other side by the overall width of the on-chip memory/cache. As the vector processor is scaled to contain more lanes, one end of the crossbars increases in size while the other end is fixed by the memory width. To quickly load a large register file, the on-chip memory/cache width must be similarly increased to match. The area to implement these crossbars is significant, and grows as the product of the two widths.

2.3 Scratchpad Memory

Many recent embedded processors in academia and industry supoprt private scratchpad memories. The CELL processor [16, 5, 6] from IBM, which is designed for streaming multimedia computations, features 8 synergistic processor elements (SPEs), each operating on fixed-width (128-bit) vectors with private SRAM scratchpads which are filled using DMA operations. The Signal-processing On-Demand Architecture (SODA) [11, 12] for 3G wireless protocols has a global scratchpad memory and local scratchpad memory for each of its 4 SIMD processors. ARM, Nios II and MicroBlaze processors also support both cache and scratchpad memory, so users can tune the system design based on the needs of a specific application.

As an architectural feature, researchers have also investigated ways to automatically utilize scratchpad memories. For a joint cache and scratchpad system, [14] presents a scheme for partitioning the scalar and array variables to minimize cache misses. In [3], a technique for static data allocation to heterogeneous memory units at compile-time is presented. Dynamic allocation approaches are also discussed in [8, 13].

2.4 Address Registers

Address registers have long been used in processors for indirect accessing of memories, where the effective address of an operand is computed based on the address register content and the addressing mode. Indirect access of memory via address registers can also be found in vector processors. The Cray-1 [17] uses eight dedicated address registers for memory accesses. The Torrent-0 [2] supports unit-stride (with auto-increment) and strided memory accesses by computing the effective address from its scalar registers, and indexed memory accesses by computing the effective address from its vector registers. The VIRAM [9, 10] also supports unit-stride, strided, and indexed accesses, but the base addresses and stride values are stored in a separate register file to comply with the MIPS ISA. In all of these cases, the address registers are dedicated for use with load or store instructions.

The register pointer architecture (RPA) [15] proposed using register pointers to indirectly access a larger register file without modifying the instruction set architecture. It also demonstrated that by changing the register pointer contents dynamically, the need for loop unrolling can be reduced. The same technique is exploited by the vector address registers in VEGAS.

3. VEGAS ARCHITECTURE

Similar to VESPA and VIPERS, the new VEGAS architecture offers scalable performance and area. Without any RTL design effort, users can configure the high-level design parameters to tune the VEGAS soft processor for a given application, providing productivity, cost, and time-to-market advantages. Moreover, VEGAS achieves better performance per unit area over the existing soft vector designs from its major design features:

1. Decoupled vector architecture,
2. Vector scratchpad memory,
3. Address registers,
4. Fracturable ALUs,
5. Data alignment crossbar network (DACN), and
6. Simplified C-macro programming.

The subsequent sections provide details on each feature.

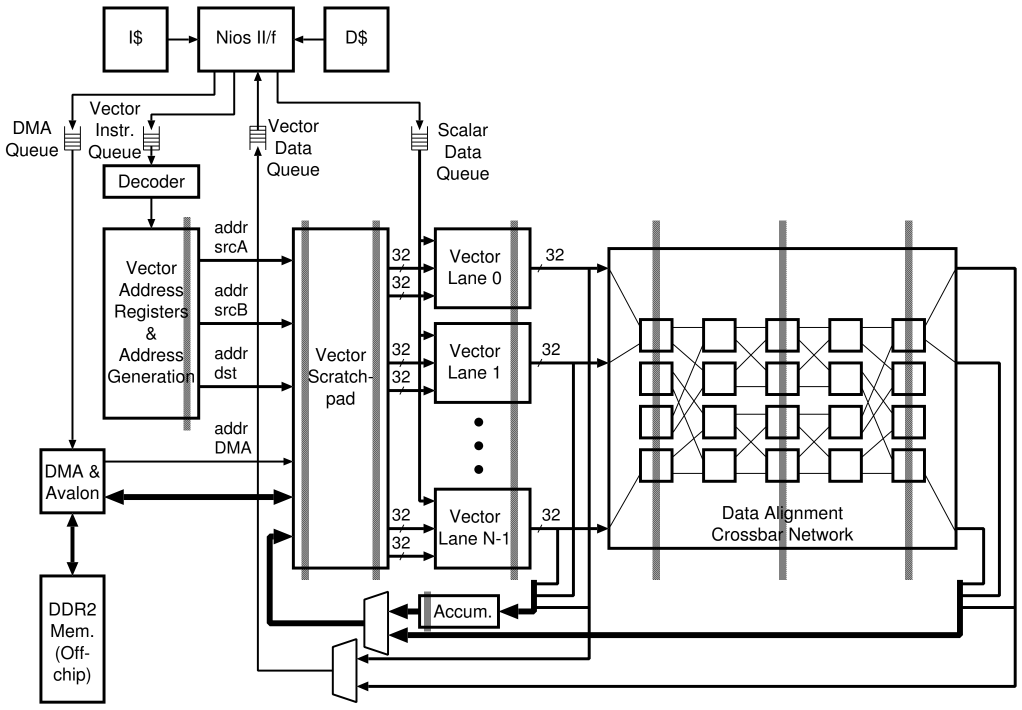

Figure 2: VEGAS Architecture (gray vertical bars indicate pipeline registers)

3.1 Decoupled Vector Architecture

VEGAS evolved from three key observations concerning VESPA and VIPERS: Nios II is small and fast and has plenty of debug support; the soft vector processors run about half the speed of Nios II/f; and vector data register files are huge while on-chip memory capacity in FPGAs is strictly limited. We wanted to design a vector processor that could scale to fill the entire FPGA, e.g. use all of the FPGA multipliers while still leaving memory blocks left over, or use all memory efficiently for applications with large datasets. We also looked at the loop-unrolled code examples presented in VIPERS and thought there must be a way to capture the performance without unrolling.

Figure 1 provides a high-level comparison of VEGAS, VIPERS and VESPA. A more detailed view of VEGAS is provided in Figure 2. VEGAS consists of a standard Nios II/f processor, a vector core, a DDR2 controller plus external memory, and a DMA engine. All of these blocks are capable of running concurrently.

The vector core does not access the Nios II instruction memory directly; vector and DMA instructions are implemented as custom instructions in the regular Nios II instruction stream. When 'executed' by Nios, these instructions are typically deposited into an appropriate queue for later asynchronous execution by the other core; if the target queue is full, the Nios II processor stalls until there is space. There are two separate 16-entry queues: one for vector instructions, and another for DMA block-transfer operations. Since vector and DMA operations usually take multiple cycles, this allows Nios II to run ahead to enqueue several instructions. This usually allows the overhead of control flow instructions to be hidden.

Some vector instructions include a scalar operand, which is deposited into a separate scalar data queue. Likewise, some vector instructions demand a response word from the vector core, such as reading a scalar value from the scratchpad memory or querying the current DMA queue length. Currently, instructions that demand a response block the Nios II, resulting in a flushed vector instruction queue. In the future, we plan to implement a non-blocking version, where the response is deposited into a vector data queue to be picked up by a later 'read vector queue' instruction. This pipelining avoids instruction queue flushes and hides the latency of crossing clock domain boundaries twice (two directions), but there is a risk of deadlock if the vector data queue fills up.

The DMA engine processes block transfers by issuing read or write operations between the DDR2 controller and the fourth port of the scratchpad memory; data movement between the scratchpad and DDR2 controller uses Altera's Avalon system fabric.

In our implementation, the Nios II/f uses a 4kB instruction cache and a 4kB data cache. The Nios II, Avalon fabric, and DMA engine run at 200MHz, and the vector core runs at 100MHz. The clock ratios need not be strictly 2:1, as dual-clock FIFOs are used to cross clock domain boundaries. Timing analysis reports indicate that VEGAS can run up to 130MHz. VIPERS and VESPA achieve similar clock rates, but they lock the scalar processor clock to the vector clock. To avoid ambiguity, we report performance results by measuring wall-clock time and reporting this elapsed time in terms of 100MHz cycles.

VEGAS encodes register values, auto-increment, and operand size in the upper 16 bits of the Nios II custom instruction. It uses 6 bits of the 8-bit N field for encoding the function to be performed, allowing up to 4 VEGAS vector cores to share a Nios II, or allowing VEGAS to coexist with other coprocessors.

3.2 Vector Scratchpad Memory

Instead of a traditional vector register file, VEGAS uses a scratchpad memory to store the working set of vector data. Eight vector address registers are used to access the scratchpad memory. Data read from vector scratchpad memory are sent directly to the vector lanes for processing, and the results are written back into the same memory. This direct coupling of the scratchpad memory and vector lanes is very efficient when the vector data operands are aligned to the lane structure. When vectors are not aligned, a data alignment crossbar network is used to correct the situation. The scratchpad enhances the VEGAS design in terms of performance, memory efficiency, and flexibility.

3.2.1 Performance

The traditional vector register file requires explicit load/store operations to transfer the data from/to memory before any vector operations can be performed. These data transfers can be time consuming. A 3–5× speedup over the original vector code is possible by fitting the primary working set into the 64-entry vector register file. As demonstrated in the VIPERS median filter example [23], this was only possible by fully unrolling two nested loops, leading to 225 unrolled iterations.

In VEGAS, data vectors are stored in the scratchpad memory and accessed via address registers. Most vector instructions specify 3 address registers: 2 source, and 1 destination. Each register specifies the starting location of a vector in the scratchpad. The number of elements in the vector is determined by a separate dedicated vector length register, VL.

When data needs to be loaded from the external DDR2 memory, the DMA engine will transfer the data into the vector scratchpad memory. This is done in parallel with vector operations, usually in a double-buffered fashion, so memory transfer latency is often completely hidden from the vector core. The programmer must explicitly ensure the DMA transfer is complete by polling or blocking before issuing any dependant vector instructions.

Performance gained by the elimination of load/store operations is maximized when the vector sources and destination reside in aligned locations and the working set fits entirely in the scratchpad memory. If the working set does not fit, vectors must be spilled to memory, degrading performance. Also, if the vectors are not aligned, additional data movement using the data alignment crossbar network (Section 3.5) is necessary to temporarily restore alignment. If unaligned accesses are predictable and occur more than once, an explicit vector move instruction to bring the data into alignment helps restore performance.

3.2.2 Memory Efficiency

In typical soft processor implementations, the register file is duplicated to provide dual read ports. However, for a vector processor, the vector data store is much larger, making the cost of duplication extremely high in terms of memory usage. In the VEGAS architecture, we take advantage of the high speed memory that is readily available in modern FPGAs and operate the memory at twice the clock frequency of the vector processor to provide the dual read ports. Therefore, the VEGAS vector processor needs just one set of data to reside in the on-chip memory, resulting in improved storage efficiency.

Moreover, instead of zero/sign-extending the vector elements to match the vector lane width, VEGAS stores vector data in the scratchpad memory at their natural length of 8 bits, 16 bits, or 32 bits. This maximizes the utilization of limited on-chip memory. As these smaller data elements are fetched directly from the scratchpad, fracturable ALUs are used to operate on elements with varying sizes. Details on the fracturable ALUs are presented in Section 3.4.

3.2.3 Flexibility

The scratchpad memory is also a more flexible form of storage than traditional vector register files. A typical vector register file is fixed in size, limiting both the number of vectors and the maximum vector length. In VEGAS, there is greater flexibility in dividing the scratchpad up into different vector lengths and the number of vectors stored. The maximum vector length is limited only by the availablity of scratchpad memory. Although there are only eight address registers, reloading an address register is far faster than reloading an entire vector of data. Hence, the number of vectors resident in the scratchpad is not artificially limited by the instruction set architecture bit encoding. This flexibility allows the use of long vectors, or many vectors, depending upon the needs of the application.

It should be noted that using long vectors is encouraged: it requires fewer iterations to compute a given task, lowers loop overhead, and prepares an application for scalable execution on wider vector processor instances. The configurability of the FPGA allows the scratchpad memory to be scaled up to provide as much storage space as allowed by the device without changing the instruction set. Since there is little waste, it is easy for users to predict the total storage capacity of VEGAS for each FPGA device. If the required scratchpad memory size does not fit in a particular FPGA device, there is the option to migrate to a device with higher capacity. The flexibility of the vector scratchpad memory allows the VEGAS architecture to tackle many different applications.

3.3 Address Registers

VEGAS contains 8 vector address registers. Each of these registers points an address in the scratchpad which holds the beginning of a vector (element 0). We found 8 address registers to be sufficient for all of our benchmarks. If more than 8 vectors are needed, a new value can be written to an existing address register very quickly (one clock cycle).

In addition, address registers have an auto-increment feature. Encoded in each instruction is a bit for each source and the destination operand which indicates whether the register should be incremented after use. The increment amount is determined by a set of 8 increment registers, one for each address register. If the same register specifier appears in multiple operand fields, it is only incremented once after the instruction completes.

The use of auto-increment helps lower loop overhead in tight vector code. The ability to specify arbitrary increment amounts is useful because the amount of increment can vary: for example, it may equal 1 element, turning the vector into a shift register, or it may equal VL elements to iterate over a long sequence of data VL elements at a time, or it may equal the row length of a 2-dimensional array to traverse the array columnwise. Although the incremented address can be computed by Nios II, this adds loop overhead (e.g., several address registers are incremented in a tight loop).

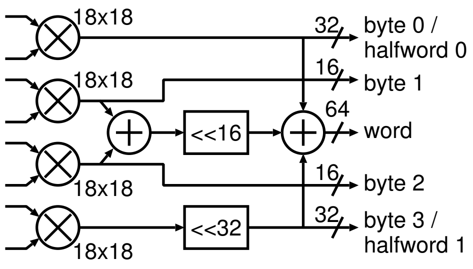

Figure 3: Fracturable Multiplier

When used with auto-increment, it is possible to obtain the equivalent performance of fully unrolled code without unrolling the loop. Our rolled-up median filter kernel is shown in Figure 6. In comparison, the original VIPERS code examples unrolled these nested loops and used 25 vector registers to hold the entire filter window. While this avoided all loads and stores in the innermost loop, the 225 loop iterations had to be *fully* unrolled (not partially) because the vector data can only be addressed by static register names. In VEGAS, unrolling is not necessary. The entire filter window is stored in the scratchpad, and data is addressed dynamically by address register. In the example, the address register V2 is auto-incremented to the next vector of data in each inner loop iteration.

The VIPERS unrolling approach is limited by the number of iterations required, the number of vector registers in the ISA, the size of the vector register file, and the amount of instruction memory available. In contrast, using address registers with auto-increment, VEGAS can automatically cycle through the data in the scratchpad with a loop without any unrolling or reloading of the address register. Only scratchpad capacity limits the length and number of vectors that can be stored. The added efficiency of natively storing bytes or halfwords without extending them to 32-bits in the scratchpad magnifies this advantage.

3.4 Fracturable ALUs

To execute on input data of different sizes, each lane must be subdivided into sub-lanes of appropriate width. This requires a *fracturable* ALU which can be subdivided to operate on four bytes, two halfwords, or one word for every 32 bits of data. Furthermore, each instruction must specify the operand size so the ALUs can be configured to an appropriate width. With fracturing, a 4-lane VEGAS processor offers the performance potential of a 16-lane VIPERS/VESPA processor when operating on byte-size vector elements.

Fracturing works together with the scratchpad to help it preserve limited on-chip memory. In VIPERS, an implementation capable of operating on all three data sizes would require each lane to have a 32-bit register file and 32-bit ALU, so both the storage efficiency and processing power is wasted when operating on vectors of halfwords or bytes. In VEGAS, the fracturable ALUs allow vectors to be stored at their natural lengths, so memory efficiency remains high.

The fracturable ALUs require a fracturable adder and fracturable multiplier. A fracturable adder is four 8-bit adders which can dynamically cascade their carry chains for wider operation. While the interrupted carry chain does cause some area overhead, the implementation is simple. In contrast, a fracturable multiplier is more complex and must

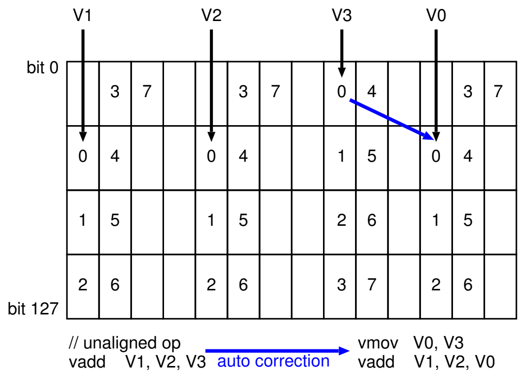

Figure 4: Example of Misaligned VEGAS Operation

support both signed and unsigned modes. In Stratix III, the embedded multiplier blocks cannot be dynamically resized or reconfigured for signed/unsigned operation. In VEGAS, we designed a fracturable multiplier around four 18×18 hardware multiplier blocks which fit nicely into a single DSP block plus additional logic. The basic design is shown in Figure 3. With careful input multiplexing, as well as output selection, this design becomes a signed/unsigned fracturable multiplier. With additional input selection logic, these multipliers are turned into fracturable shifters as well.

The ALUs support absolute-difference and sum-of-absolute-difference instructions. Also, we have an accumulate function like VIPERS, but our implementation is significantly different. Instead of instantiating more DSP blocks and using the built-in accumulators, we build our own fracturable accumulators in logic placed near the write-back port of the scratchpad.

3.5 Data Alignment Crossbar Network

Since the scratchpad memory couples directly to the vector lanes in VEGAS as shown in Figure 2, all vectors involved in a single vector instruction (two sources and one destination) must reside in aligned locations. When vectors are not aligned, the input to the vector lanes is mismatched and would produce an incorrect result. When misalignment occurs, an extra instruction is necessary to move the data into alignment as shown in Figure 4.

To help lower the learning curve in software development, misalignment detection and auto-correction logic is implemented in VEGAS. If two source vectors are misaligned, a vector move instruction is automatically inserted into the pipeline. The automatic-move makes use of the data alignment crossbar network to align the one operand, and stores the result at a reserved location determined by vector address register V0. Then the original instruction is re-issued, using V0 to replace the misaligned source. If the destination is also misaligned, the result is re-aligned on the fly prior to writing back the result in the correct location.

Although the auto-correction is convenient, it represents lost performance. To help tune software, performance counters are implemented for each of three misalignment cases that occur while running the vectorized code. The user can retrieve the counter values via control registers, and utilize this information to optimize their application.

In addition to alignment, variations of the move instruction called scatter and gather provide strided access through

```
#include "vegas.h"

int dotprod( int *v1, int *v2,
             int const1, int const2, int vec_len)
{
  int result;
  int *vegas_v1,*vegas_v2,*vegas_result;

  //Allocate two vectors in scratchpad
  vegas_v1 = vegas_malloc( vec_len*sizeof(int) );
  vegas_v2 = vegas_malloc( vec_len*sizeof(int) );
  vegas_result  = vegas_malloc( sizeof(int) );

  //Start the DMA transfers to VEGAS
  vegas_dma_to_vector(vegas_v1,v1,vec_len*sizeof(int));
  vegas_dma_to_vector(vegas_v2,v2,vec_len*sizeof(int));

  //Set VEGAS VL and set address registers
  vegas_set( VCTRL, VL, vec_len );
  vegas_set( VADDR, V1, vegas_v1 );
  vegas_set( VADDR, V2, vegas_v2 );
  vegas_set( VADDR, V3, vegas_result );

  //Zero the accumulators before using them; whatever was
  //in them will be written to V3 which will be overwritten
  //with the result later
  vegas_vvw( VCCZACC, V3, VUNUSED, VUNUSED );

  //Wait for memory transfer to complete before computation
  vegas_wait_for_dma();

  //Multiply vector by scalar words
  vegas_vsw( VMULLO, V1, V1, const1 );
  vegas_vsw( VMULLO, V2, V2, const2 );

  //Multiply the two vectors together and accumulate the
  //result of the multiply will be written back to V2 while
  //the accumulation happens in the external accumulators
  vegas_vvw( VMAC, V2, V2, V1 );

  //Store result in first word of V3 and zero the accumulators
  vegas_vvw( VCCZACC, V3, VUNUSED, VUNUSED );

  //Extract the result.  This also syncs Nios II and VEGAS
  vegas_vsw( VEXT, result, V3, VUNUSED );

  //Free all scratchpad memory
  vegas_free();

  return result;
}
```

Figure 5: Example Dot Product

```
  //Bubble sort up to halfway
  for( j = 0; j < FILTER_SIZE/2; j++ ) {
    vegas_set( VADDR, V1, v_temp+j*IMAGE_WIDTH );
    vegas_set( VADDR, V2, v_temp+(j+1)*IMAGE_WIDTH );

    for( i = j+1; i < FILTER_SIZE; i++ ) {
      vegas_vvb( VMAXU, V4, V1, V2 );
      vegas_vvb( VMINU, V1, V1, V2 );
      vegas_vvb( VMOVA, V2INC, V4, V4 );
    }
  }
```

Figure 6: Example Median Filter Kernel

the scratchpad. Scatter takes a densely packed vector as the source, and writes a new vector with elements separated by gaps of $n-1$ elements, where n is called the stride. Gather works in the opposite direction. Furthermore, both scatter and gather are capable of converting elements of one data size into any other data size (words, halfwords, and bytes) through sign-extension, zero-extension, or truncation.

The alignment, scatter, and gather operations are achieved using a data alignment corssbar network (DACN). Instead of using a full crossbar, which is costly in resource usage, VEGAS employs a Benes network [4]. This network can realize all of the required permutations, but requires only $O(N \log N)$ instead of $O(N^2)$ logic in the switching network. However, the multistage nature of this network requires complex control algorithms. The control must first generate the output ports for each input data element, then the control values for each layer of the switch. Our current control logic is inefficient for scatter and gather operations with certain strides, and this limits the speedup achieved with certain applications.

3.6 Programming VEGAS

VEGAS is programmed in a manner similar to inline assembly in C. However, C macros are used to simplify programming and make VEGAS instructions look like C functions without any run time overhead. The sample code in Figure 5 multiplies two vectors by separate scalars and then computes their dot product.

To add byte vectors pointed by address registers `V2` to `V1`, increment `V1` and store the result in `V3` the required macro would be `vegas_vvb(VADD,V3,V1INC,V2)`. Vector register specifiers can take on any value `V0` through `V7`, but special register `V0` is also used by the system for temporary storage during automatic alignment operations. A register specifier written as `V7INC`, for example, post-increments the register by a signed amount previously stored in its respective increment register. A placeholder value `VUNUSED` can be used with certain instructions. In the function name, the `vvb` suffix refers to 'vector-vector' operation (`vv`) on byte-size data (`b`). Other combinations are scalar-vector (`sv`) or vector-scalar (`vs`), where the first or second source operand is a scalar value provided by Nios instead of a vector address register. These may be combined with data sizes of bytes (`b`), halfwords (`h`) and words (`w`). For example, computing a vector of halfwords in `V7` by subtracting a vector `V2` from the integer scalar variable k would be written as `vegas_svh(VSUB,V7,k,V2)`. In addition, conditional execution of individual vector elements is achieved via a vector mask, where a mask is a vector of 1-bit results from a vector comparison.

Data can be allocated in vector scratchpad memory using special `vegas_malloc(num_bytes)` which returns an aligned pointer. By default, `V0` tracks the end of the allocated scratchpad memory, allowing the use of all remaining space as an alignment buffer. The `vegas_free()` call simply frees all previous scrachpad allocations. DMA transfers and instruction synchronization are handled by macros as well.

While this approach requires programmers to manually manage vector address registers, it should be straightforward to create a compiler to manage register allocation and allow the programmer to deal with pointers directly. Such a compiler could also infer data size from the pointer type.

Num.	VEGAS				VIPERS				VESPA			
Lane	ALM	DSP	M9K	Fmax	ALM	DSP	M9K	Fmax	ALM	DSP	M9K	Fmax
1	3,831	8	40	131					5,556	10	46	133
2	4,881	12	40	131					6,218	14	48	130
4	6,976	20	40	130	5,825	25	16	106	7,362	22	76	128
8	11,824	36	40	125	7,501	46	25	110	12,565	39	98	124
16	19,843	68	40	122	10,995	86	41	105	20,179	79	98	113
32	36,611	132	40	116					34,471	175	196	98

Table 1: Resource Usage Comparison

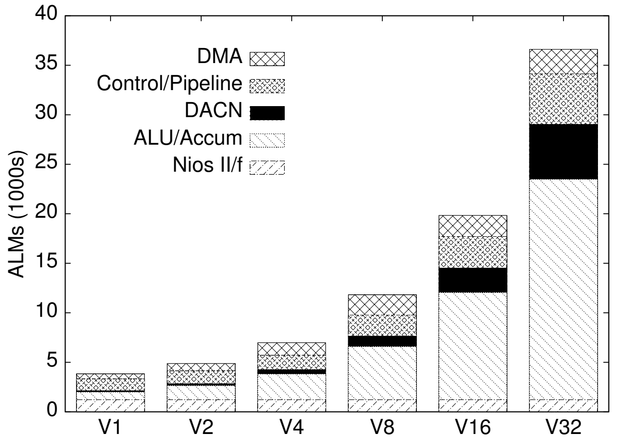

Figure 7: VEGAS Area Breakdown (ALM count)

4. RESULTS

In this section, the resource usage, performance, and area-delay product of the new VEGAS soft vector architecture is compared against existing soft vector architectures VESPA and VIPERS.

4.1 Resource Usage

Different configurations of the VEGAS architecture are compiled using Quartus II to measure their resource usage and compare them against the equivalent VESPA/VIPERS designs. VEGAS is compiled targeting the Stratix III EP3SL150F1152C3ES device that is found on the Altera DE3-150 development board. Since all three architectures target Altera's Stratix III devices on a DE3 platform, comparisons are easy.

The logic usage is summarized by the number of adaptive logic modules (ALMs) and DSP 18x18 elements, while memory usage is summarized by the number of M9K blocks. Table 1 lists the ALM/DSP/M9K usage for all three architectures with various number of lanes, as well as the maximum clock frequency of the design. The V1, V2, V4 labels indicate 1, 2, 4 vector lanes (each lane is 32 bits in width), respectively. For VEGAS, this means V4 can run instructions on byte-wide data using 16 vector sub-lanes. For these results, VEGAS was configured to use a 32kB scratchpad.

In terms of logic resources, the ALM usage of VEGAS typically settles between VIPERS and VESPA, except VEGAS surpasses VESPA in the 32-lane configuration. The breakdown of ALM usage in Figure 7 shows that ALUs account for the majority of ALM use. This is partly explained by the complexity of the fracturable ALUs, and partly by the complex min/max/absolute-difference instructions. The multiplier also requires significant ALM resources for its many modes and shifting operations. Future work should reduce ALM usage in the ALUs.

In terms of multiplier resources, the fracturable ALUs were carefully designed to limit use of DSP blocks, and we avoided using multipliers in address calculations. As a result, VEGAS is slighty better than both VIPERS and VESPA in DSP usage.

In terms of memory, the number of M9K memory blocks consumed by VEGAS is similar to that of both VIPERS and VESPA. However, it is much more efficient, allowing up to 8× more vector data elements to be stored in the same number of M9K blocks.

4.2 Performance

For performance comparisons, we adopted benchmarks from VIPERS (motion estimation, median filter), VESPA (EEMBC benchmarks autocor, conven, fbital plus VIRAM benchmarks imgblend and filt3x3), and also added a classic fir filter with 16 taps on 4096 halfword samples.

We compiled all applications using gcc with -O3 optimization and ran them on the Nios II/f processor at 200MHz to establish the baseline performance. Next, we recoded the benchmarks by hand using the VEGAS C macros and attempted to maximize the vector lengths, use the smallest data elements needed by each benchmark, and overlap DMA with computation. We compiled VEGAS vector core instances ranging from 1 to 16 lanes with 256kB, and 32 lanes with 128kB scratchpad memory.[1] The vector core was run at 100MHz. Performance was then measured using hardware timestamp counters running at 100MHz. The number of 100MHz clock cycles required for each configuration is shown in Table 2. For VEGAS, we also calculated the highest speedup over Nios II on these benchmarks. For VIPERS and VESPA results, we extracted cycle counts from published works [19, 23].

Compared to VIPERS, VEGAS matches it on the median filter benchmark, and beats it by 2× on motion estimation. However, we also see that VEGAS achieves peak performance with fewer lanes than VIPERS because of the fracturable ALUs. This gives VEGAS an area advantage. Also, the VIPERS results were achieved through aggressive loop unrolling in both benchmarks, but VEGAS does not use any loop unrolling.

Compared to VESPA, VEGAS matches it on imgblend and filt3x3, beats it on autocor and fbital, but runs more slowly on conven. The slower performance of conven is due to strided scatter/gather operations which do not yet execute at full speed in VEGAS due to complex DACN control. We are actively working to address this limitation. However, we note that VEGAS always achieves better perfor-

[1]The fbital benchmark requires a 256kB scratchpad for a lookup table, so we could not collect V32 performance data.

CPU Name	fir	motest	median	autocor	conven	fbital	imgblend	filt3x3
	Benchmark Name							
Nios II/f	509,919	1,668,869	1,388	124,338	48,988	240,849	1,231,172	6,556,592
	VIPERS			VESPA				
V1				125,376	17,610	358,070	1,127,965	2,754,216
V2				64,791	9,821	191,054	677,856	1,432,396
V4		157,792	189	32,556	4,479	106,911	229,014	608,222
V8		88,288	95	16,896	2,103	69,186	112,937	271,252
V16		55,328	48	10,287	1,638	47,303	64,176	145,729
V32				7,062	984	39,372	33,953	72,966
	VEGAS							
data size	halfword	byte	byte	word	byte	halfword	halfword	halfword
VL	dynamic	256	dynamic	1024	512	256	320	317
V1	85,549	82,515	185	45,027	3,462	165,839	175,890	813,471
V2	47,443	47,249	93	26,512	2,690	90,114	99,666	419,589
V4	25,346	30,243	47	14,470	2,154	52,278	61,169	219,816
V8	13,536	29,643	24	7,941	1,976	33,166	40,231	122,072
V16	7,690	27,344	12	4,563	1,924	23,999	35,656	75,776
V32	4,693	24,717	7	2,822	1,897	–	35,485	75,349
Speedup	108×	67×	208×	44×	25×	10×	34×	87×

Table 2: Performance Comparison (100MHz clock cycles)

CPU Name	fir	motest	median	autocor	conven	fbital	imgblend	filt3x3
	Benchmark Name							
Nios II/f	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
VIPERS		(V16) 0.31	(V16) 0.41					
VESPA				(V16) 1.37	(V8) 0.44	(V8) 2.95	(V32) 0.78	(V32) 0.31
VEGAS	(V16) 0.24	(V4) 0.10	(V16) 0.14	(V16) 0.60	(V2) 0.22	(V4) 1.24	(V4) 0.28	(V8) 0.18
Improvement		3.1×	2.9×	2.3×	2×	2.4×	2.8×	1.7×

Table 3: Best Area-Delay Product (Normalized to Nios II/f, lower=better)

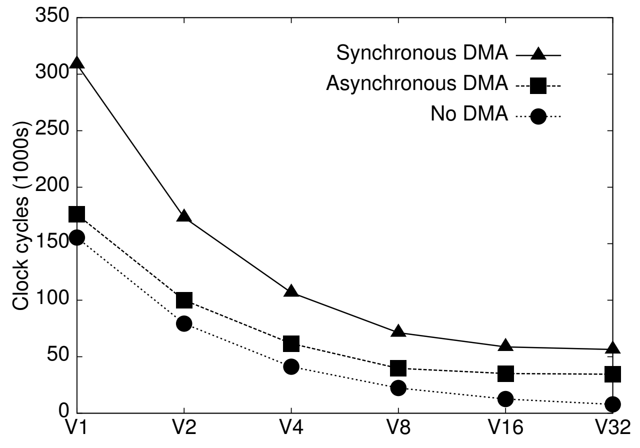

Figure 8: Performance of imgblend Benchmark

mance when using a smaller number of lanes (V1 to V8, and often V16). In particular, VEGAS-V2 is 6.8× faster than VESPA-V2 on imgblend.

Table 3 reports the best area-delay product achieved by each vector processor. For VEGAS, this is often at a small number of lanes. A lower area-delay is better, as greater throughput performance can be obtained by replicating these vector cores across the chip. The VEGAS area-delay product is up to 68% lower than VIPERS and up to 64% lower than VESPA, giving it up to 3.1× better throughput per unit area.

The ability to run DMA operations concurrently in a double-buffered fashion often allows all memory latency to be hidden. To demonstrate this, we ran the imgblend application in three modes: synchronous DMA, asynchronous DMA, and no DMA. The runtime of these three modes is shown in Figure 8. With synchronous DMA, we block waiting for each DMA operation to finish before performing any computation. With no DMA, we skipped the DMA transfer entirely, so runtime is entirely computational (and uses incorrect data). The asynchronous DMA result allows computation to occur while the DMA operates in parallel. The figure shows that most of the transfer latency is successfully hidden up to V8. Hiding the latency at V16 and V32 is difficult because the computation is so fast.

	Clock speed	1024×1024	4096×4096
Nios II/f	200MHz	77.78	5406.81
Intel Core 2	2.66GHz	1.09	71.66
VEGAS	100MHz	0.72	43.77

Table 4: Integer Matrix Multiply Runtime (seconds)

Finally, using integer matrix multiply, Table 4 shows that VEGAS outperforms a 2.66GHz Intel Xeon X5355 processor. We believe this is the first time a soft processor has outperformed a leading-edge hard processor in similar (65nm) technology. The Nios II and Intel versions ran individually tuned single-threaded C programs that used various loop orderings and tile sizes compiled with gcc -O3. Note that Intel SSE instructions were not used. The VEGAS version was the fastest code written among a graduate class of 10 students; the slowest code was roughly 2× slower.

5. CONCLUSIONS

Previous work on VIPERS and VESPA have demonstrated that soft vector processors can scale and accelerate data-parallel embedded applications on FPGAs. Vector processoring offers an easier way to explore the area-performance tradeoffs than designing custom logic accelerator in VHDL or Verilog.

This paper introduces VEGAS, a new soft vector architecture that optimizes use of limited on-chip memory in modern FPGAs. An on-chip scratchpad memory serves as the vector data storage medium and is accessed using vector address registers. This eliminates traditional restrictions on the maximum vector length and number of vectors. Instead of traditional vector load/store operations, double-buffered asynchronous DMA can potentially hide all memory latency. Better storage efficiency (up to 9×) allows more vector data to reside on-chip, avoiding the need to spill vector data. Additional performance enhancement is achieved by fracturable ALUs, which increases processing power up to 4× on byte-size data.

The VEGAS architecture supports very long vector lengths. Together with the auto-increment feature of the address registers, this can reduce overhead in tight loops. The address registers achieve the performance of unrolled code without doing any unrolling, making it easier to program and debug and reducing code bloat.

Using area-delay product on selected benchmarks, VEGAS provides up to 3.1× higher throughput per unit area than VIPERS, and up to 2.8× higher than VESPA. In raw performance, VEGAS is 10× to 208× faster than Nios II on the same benchmarks. Using integer matrix multiply, VEGAS is faster than a 2.66GHz Intel X5355 (Clovertown) processor. This may be the first time a soft processor has surpassed a leading-edge hard processor in performance.

Finally, the C-based macro system is easier to program than assembly language. While not offering the full convenience of an autovectorizing compiler, it relieves the programmer of scalar assembly instructions yet still allows explicitly scheduled data movement and vector instructions.

6. ACKNOWLEDGMENTS

Funding for this work was provided by NSERC and hardware donations by Altera Corp. We thank the following graduate students for coding their own matrix multiply algorithms and providing usability feedback: Kyle Balston, Stuart Dueck, Abdullah Gharaibeh, Jeffrey Goeders, Ziaeddin Jalali, Tim Rogers, Emalayan Vairavanathan, and Jiesheng Wei (the other 2 students are co-authors Alex and Zhiduo). Finally, thanks are due to Tor Aamodt, Philip Brisk, Philip Leong, Satnam Singh, and Greg Steffan for their useful feedback on this work, as well as Christos Kozyrakis for providing VIRAM-coded versions of EEMBC benchmarks for analysis.

7. REFERENCES

[1] The embedded microprocessor benchmark consortium. http://www.eembc.org/.

[2] K. Asanovic. *Vector microprocessors.* PhD thesis, University of California, Berkeley, 1998.

[3] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM TECS*, 1(1):6–26, 2002.

[4] V. E. Benes. Optimal rearrangeable multistage connecting networks. *Bell Systems Technical Journal*, 43(7):1641–1656, 1964.

[5] S. H. Dhong, O. Takahashi, et al. A 4.8 GHz fully pipelined embedded SRAM in the streaming processor of a CELL processor. In *ISSCC*, pages 486–487, 2005.

[6] B. Flachs, S. Asano, et al. A streaming processing unit for a CELL processor. In *ISSCC*, pages 134–135, 2005.

[7] 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.

[8] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. A Compiler-Based approach for dynamically managing Scratch-Pad memories in embedded systems. *IEEE TCAD*, 23(2):243–260, 2004.

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

[10] C. E. Kozyrakis and D. A. Patterson. Scalable vector processors for embedded systems. *IEEE Micro*, 23(6):36–45, 2003.

[11] Y. Lin, H. Lee, Y. Harel, M. Woh, S. Mahlke, T. Mudge, and K. Flautner. A system solution for High-Performance, low power SDR. In *SDR Technical Conference*, 2005.

[12] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. SODA: a low-power architecture for software radio. *SIGARCH Comput. Archit. News*, 34(2):89–101, 2006.

[13] B. Mathew and A. Davis. An energy efficient high performance scratch-pad memory system. *DAC*, 2004.

[14] P. R. Panda, N. D. Dutt, and A. Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM TODAES*, 5(3):704, 2000.

[15] J. Park, S. Park, J. D. Balfour, D. Black-Schaffer, C. Kozyrakis, and W. J. Dally. Register pointer architecture for efficient embedded processors. In *DATE*, pages 600–605. EDA Consortium, 2007.

[16] D. Pham, E. Behnen, M. Bolliger, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, B. Le, Y. Masubuchi, et al. The design methodology and implementation of a first-generation CELL processor: a multi-core SoC. In *CICC*, pages 45–49, 2005.

[17] R. M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21(1):63–72, 1978.

[18] M. Woh, Y. Lin, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, R. Bruce, D. Kershaw, A. Reid, M. Wilder, and K. Flautner. From SODA to scotch: The evolution of a wireless baseband processor. In *MICRO 41*, pages 152–163, 2008.

[19] P. Yiannacouras. *FPGA-Based Soft Vector Processors.* PhD thesis, Dept. of ECE, Univ. of Toronto, Canada, 2009.

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

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

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

[23] 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.

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