Vector Processing as a Soft-core CPU Accelerator

Jason Yu
jasony@ece.ubc.ca

Guy Lemieux
lemieux@ece.ubc.ca

Christopher Eagleston
ceaglest@ece.ubc.ca

Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada V6T 1Z4

ABSTRACT

The currently accepted method of accelerating applications in FPGA soft processor systems is to design a custom hardware accelerator. This paper suggests the alternative approach of adding a vector processing core to the soft processor as a general-purpose accelerator. The approach has the benefit of a purely software-oriented development model. With *no hardware design experience needed*, a software programmer can make area-versus-performance tradeoffs by scaling the number of functional units or *vector lanes*. This paper shows that a vector processing architecture maps efficiently into an FPGA and provides a scalable amount of performance for a reasonable amount of area. Three configurations of the soft vector processor with different performance levels are estimated to achieve scalable speedup ranging from 3–29× for 6–30× the area of a Nios II/s processor on three benchmark kernels. The results compare favourably to accelerators designed using Altera's C2H compiler, a C-to-hardware tool that is also easy to use.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*array and vector processors, single-instruction-stream, multiple-data-stream processors (SIMD)*

General Terms

Measurement, Performance, Design

Keywords

C2H, FPGA, application specific, configurable, data-level parallelism, embedded processor, soft processor

1. INTRODUCTION

Designers of FPGA-based systems find soft-core processors very convenient because software programming is far simpler than hardware design, thus shortening time-to-market and development costs. However, the amount of compute performance available from soft-core processors is strictly limited. For example, Altera's Nios II [1] is available in only three performance levels, the most sophisticated of which is a single-issue, in-order RISC pipeline. Hence, embedded applications that have plenty of data-level parallelism in tight, computationally-intensive inner loops [2] are performance-limited by the processor design itself.

Three ways of accelerating FPGA-based applications with plenty of data parallelism are: 1) use FPGA logic fabric to design a custom hardware accelerator, or 2) build a soft-core multiprocessor system and write parallel code, or 3) change the processor design to have more parallelism. The first approach requires some level of hardware design experience, even with high-level tools like Altera's C2H compiler [3], which compiles user-specified functions in the application into co-processors in a Nios II-based system. The second approach requires worrying about the complexity of parallel debugging and coping with incoherent memory. The third approach would provide all soft-core users with an improved, scalable processor core. However, traditional superscalar and VLIW techniques are not viable due to overheads implementing wide issue logic and multiple register file write ports. Instead, a different type of processor design is needed.

Another way to improve soft-core processors for data-parallel applications is to adopt a SIMD or vector architecture. In this paper, we show that this type of architecture maps well to FPGA devices after the vector registers are partitioned across several memories—a technique described in [4]. Unlike existing soft-core CPUs which have limited configurability, a soft-core vector processor can have a large number of control parameters which strongly influence performance and area. For example, the datapath width can be configured to 8, 16, or 32 bits, and performance can be scaled to the desired level by selecting the number of parallel functional units or *vector lanes*. Excluding unused instructions or microarchitectural features allows further customization.

The key benefit of using a soft-core vector architecture is achieving high performance with reduced development effort and faster time-to-market. We found that vector processing achieves performance benefits similar to a custom hardware accelerator but with *zero hardware design effort*. We also found the process of writing vector assembly code to be simpler than performance-tuning a custom accelerator, even with a high-level tool like C2H; presumably, a vectorizing compiler would make this even easier. Unlike custom accelerators, a vector processor can even serve multiple ap-

FPGA'08, February 24-26, 2008, Monterey, California, USA.

plications with the same hardware instance. This can reduce the need for several hardware development iterations or the need to store multiple device configurations.

The benefits of using a vector processor are more about rapid development than ultra-high performance. It is likely that higher performance could be obtained using any number of alternatives, including a traditional custom CPU, DSP, GPU, or even hand-crafted RTL in an FPGA. However, a custom CPU or GPU is not always available on the circuit board, and hand-crafting an RTL accelerator is very time-consuming. Hence, for comparison purposes, we use C2H to rapidly produce custom accelerators. Furthermore, we do not suggest that end-users each develop their own soft-core vector processor, as that is a complex task that should be adopted by the FPGA vendor or 3rd party vendors. Also, although we write vector assembly code by hand, we presume that a vectorizing compiler would be provided by the vendor to facilitate even more rapid design.

The remainder of the paper is organized as follows. Section 2 gives background on vector processing. Section 3 describes the soft vector processor architecture. Section 4 illustrates how the benchmark kernels are written for the soft vector processor. Section 5 gives performance estimates for the processor, and Section 6 presents some of our suggestions to FPGA architecture to ease implementation and improve performance of soft-core vector processors.

1.1 Related Work

Many previous attempts to implement vector processors in FPGAs targetted only a specific application, or were prototypes of ASIC implementations. Two vector processors that, similar to this work, were specifically designed for FPGAs are described in [5, 6]. The first vector processor [5] consists of two identical vector processors located on two Xilinx XC2V6000 FPGA chips. Each vector microprocessor runs at 70MHz, and contains a simplified scalar processor with 16 instructions, a vector unit consisting of 8 vector registers, 8 lanes (each containing a 32-bit floating-point unit), and supports a maximum vector length (MVL) of 64. Eight vector instructions are supported, but only matrix multiplication was demonstrated on the system. Although our vector processor lacks floating-point support, it presents a more complete solution consisting of full scalar unit (Nios II) and a full vector unit (based on VIRAM instructions) that supports over 45 distinct vector instructions plus variations.

The second vector processor [6] was designed for Xilinx Virtex-4 SX and operated at 169MHz. It contains 16 integer processing lanes and 17 on-chip memory banks connected to a MicroBlaze [7] processor through fast simplex links (FSL). It is not clear how many vector registers were supported. Compared to the MicroBlaze, speedups of 4–10× were demonstrated with four applications (FIR, IIR, matrix multiply, and 8×8 DCT). The processor implementation seems fairly complete.

Mainstream processors have also adopted vector or vector-like computing models. Vector-inspired SIMD extensions are supported by virtually all recent microprocessors from Intel, IBM, and some MIPS processors in the form of multimedia instruction extensions. SIMD extensions are oriented towards short vectors, with typically 128-bit wide multimedia register for storing vectors. In general, they lack support for strided memory access patterns and more complex memory manipulation instructions, with the result of devoting many instructions to address transformation and data manipulation to support the few instructions that do the actual computation [8]. Full vector architecture mitigates these effects by providing a rich set of memory access and data manipulation instructions, and longer vectors to keep functional units busy and reduce overhead [9]. The Torrent T0 [4] and VIRAM [10] are single-chip vector microprocessors that support a complete vector architecture and are implemented as custom ASICs. They share the most similarity in processor architecture to this work.

Automatic co-processor generation has recently become popular. Besides the Altera C2H compiler, Cascade [11] by Critical Blue is another tool that generates a customized co-processor to a main processor for SoC, structured ASIC and FPGA platforms. The Cascade co-processor has a VLIW architecture, which differs from C2H's custom-hardware and loop-pipelining-based co-processors. Cascade generates a co-processor by analyzing the compiled object code of an application, and connects it to the main processor via the bus interface of the main processor. Similar to this work, the co-processor is scalable in performance and area. The co-processor has options to configure the instruction format and control the amount of instruction decode logic. A single co-processor can also be reused to accelerate multiple non-overlapping portions of an application. This work differs by adopting a tightly-coupled vector processor architecture, which is more suited to the architecture of FPGAs. The CHiMPS compiler from Xilinx Labs [12] also generates hardware from software. Although little is published about CHiMPs, it appears to use a streaming dataflow model and deep pipelining for performance.

FPGAs excel in configurability, and configurable soft processors abound in both academic and industrial spaces. The SPREE [13] framework can generate application-specific soft processors with selectable features such as pipeline organization, multiplier and shifter implementation. The Altera Nios II and the Xilinx MicroBlaze are both configurable RISC soft processor cores designed for use on an FPGA with options to integrate custom-designed hardware accelerators.

2. VECTOR PROCESSING INTRODUCTION

Vector processors have traditionally excelled in scientific and engineering applications. Recently, vector microprocessors have also been shown to be more effective in embedded media applications such as the EEMBC suite [14] than superscalar and VLIW processors [15]. Below, we introduce the vector processing model using a FIR-filter example.

The vector processing model operates on vectors of data. Each vector instruction specifies one operation on the entire vector, generating tens of operations on independent data elements and producing tens of results at a time. Data to be operated on is stored in a large vector register file that can hold a moderate number of vector registers, each containing a large number of data elements. Entire vectors can be gathered from main memory to the vector register file through vector load instructions, and scattered to memory through vector store instructions. Data elements do not have to be located in adjacent memory locations. Vector architectures support *strided memory access*, which accesses data elements in memory with a constant size separation between elements, and *indexed memory access*, which accesses data elements by adding a variable offset for each element to a common base address.

Vector instructions are controlled by a vector length (VL) register, which specifies the number of elements within the vector to operate on. The vector length register, together with mechanisms such as vector flags, provide conditional execution in a vector instruction set. A vector architecture contains a vector unit and a separate scalar unit. The scalar unit executes non-vectorizable portions of the program, and most control flow instructions.

Vector processing is a simple programming model suited to describing data-level parallelism. Consider an 8-tap finite impulse response (FIR) filter

$$y[n] = \sum_{k=0}^{7} x[n-k]h[k],$$

which can be implemented in MIPS asembly code as shown in Figure 1. The loop will iterate 8 times for the 8-tap filter, executing a total of 65 instructions.

```
        movi    r4, 8
.L5:
        ldw     r7, 0(r5)
        ldw     r3, 0(r6)
        addi    r4, r4, -1
        addi    r5, r5, -4
        mul     r2, r7, r3
        addi    r6, r6, 4
        add     r16, r16, r2
        bne     r4, zero, .L5
```

Figure 1: 8-tap FIR filter MIPS assembly

The same FIR filter implemented in vector code is shown in Figure 2. The example assumes a vector ISA similar to that of VIRAM. A total of 11 instructions are needed including setting up base addresses and control registers, and loading a new sample after producing a result. One common operation in vector processing is reduction of the data elements in a vector register. In the FIR filter example, the multiplication products need to be sum-reduced to the final result. Multiply-accumulators in the DSP blocks of an FPGA can be used for sum reduction, and they are utilized by the `vmac` instruction to accumulate the multiplication results. The multiply-accumulator is a special feature of this vector processor further discussed in Section 3.1.

```
vmstc       vbase0, r1      ; Load base address
vmstc       vbase1, r2
vmstc       VL, r3          ; Set VL to num data
vld         v0, vbase0      ; Load input vector
vmstc       VL, r4          ; Set VL to num taps
vld         v2, vbase1      ; Load filter coefficients

vmac        v0, v2          ; Multiply-accumulate
                            ; up to 16 values
vcczacc     v3              ; Copy from accumulator and zero
veshift     v0, v0          ; Vector element shift
ldw         r5, 0(r1)       ; Load newest sample
vins.vs     v0, r5          ; Insert sample to vector register
```

Figure 2: 8-tap FIR filter vector assembly

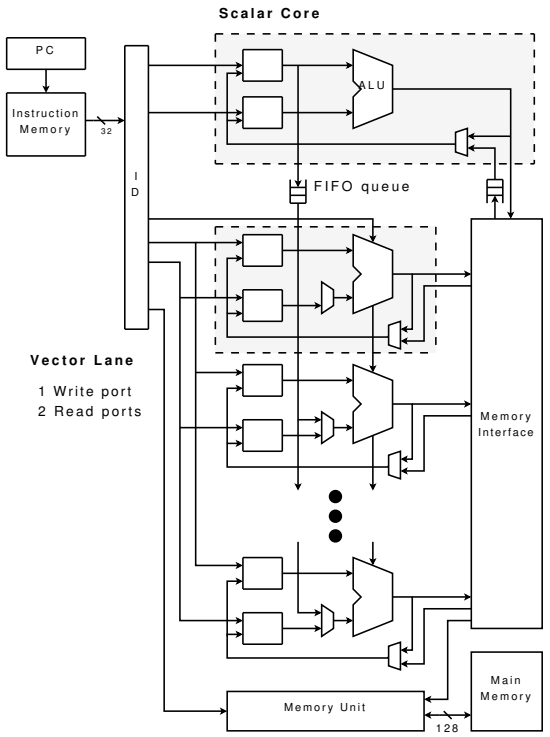

Figure 3: Scalar and vector core interaction

3. SOFT VECTOR ARCHITECTURE

The soft vector architecture specifies a family of soft vector processors with varying performance and resource utilization, and different features to suit different applications. A software generator uses a number of parameters to generate an application- or domain-specific instance of the processor. The configurability gives designers flexibility to trade-off performance and resource utilization, and to further fine-tune resource usage by removing unneeded processor features and instruction support. Table 1 lists the configurable parameters and features of the processor architecture described in this paper. Five configuration instances of the soft vector processor are shown and will be further discussed in Section 5.2. Our particular soft vector processor is tailored to the Altera Stratix III FPGA architecture. The sizes of embedded memory blocks, functionality of the hard-wired DSP blocks, and mix of logic and other resources in the Stratix III family drove many of our design decisions.

Figures 3 and 4 illustrate the soft vector processor. The architecture consists of a scalar core, a vector processing unit, and a memory interface unit. The scalar core is the single-threaded version of the UTIIe [16], a 32-bit Nios II-compatible soft processor with a four-stage pipeline. The scalar core and vector unit share the same instruction memory and instruction fetch logic. Vector instructions are 32-bit, and can be freely mixed with scalar instructions in the instruction stream. The scalar and vector units can execute different instructions concurrently, but will coordinate via the FIFO queues for instructions that require both cores, such as instructions with both scalar and vector operands.

The vector processing unit is shown in detail in Figure 4. The vector unit is composed of a number of vector lanes,

Table 1: List of configurable processor parameters

			Vector Processor Configuration				
Parameter	Description	Typical	*V4F*	*V8F*	*V16F*	*V16M32*	*V16W16*
NLane	Number of vector lanes	4–128	4	8	16	16	16
MVL	Maximum vector length	16–512	16	32	64	64	64
VPUW	Processor data width (bits)	8,16,32	32	32	32	32	16
MemMinWidth	Minimum accessible data width in memory	8,16,32	8	8	8	32	8
MultW	Multiplier width (bits, 0 is off)	0,8,16,32	16	16	16	16	16
MACL	MAC chain length (0 is no MAC)	0,1,2,4	1	2	4	0	4
LMemN	Local memory number of words	0–1024	256	256	256	256	0
LMemW	Local memory width (bits)	8,16,32	32	32	32	32	0
LMemShare	Shared local memory address space within lane	On/Off	Off	Off	Off	On	Off

with the number specified by the *NLane* parameter. Each vector lane has a complete copy of the functional units, a partition of the vector register file and vector flag registers, a load-store unit, and a local memory if parameter *LMemN* is greater than zero. The internal data width of the vector processing unit, and hence width of the vector lanes, is determined by the parameter *VPUW*. All vector lanes receive the same control signals and operate independently without communication for most vector instructions. *NLane* is the primary determinant of the processor's performance. With additional vector lanes, a fixed-length vector can be processed in fewer cycles, improving performance. In the current implementation, *NLane* must be a power of 2.

The soft vector processor uses a vector register file that is distributed across vector lanes. This differs from traditional vector architectures which employ a large, centralized vector register file with many ports. The vector register file is element-partitioned — each vector lane has its own register file that contains all the vector registers, but only a few data elements of each vector register [4]. The partitioning scheme naturally divides the vector register file into parts that can be implemented using the smaller memory blocks on the FPGA. It also allows SIMD-like access to multiple data elements in the vector register file by the vector lanes. Furthermore, the distributed vector register file saves area compared to a large, multi-ported vector register file. *The abundance of these small memory blocks (and multipliers) makes modern FPGAs good at implementing vector processors.* Each vertical dark-gray stripe in Figure 4 represents a vector register spanning all lanes. The ISA defines 64 vector registers. Assigning four 32-bit elements of each register to each lane fills one M9K RAM; this is duplicated to provide two read ports. For this reason, *MVL* is typically 4 times *NLane* for a 32-bit *VPUW*, and most vector instructions that use the full vector length execute in 4 clock cycles.

The memory interface unit handles memory accesses for both scalar and vector units. Scalar and vector memory accesses occur in program order. Vector memory instructions are processed independently from vector arithmetic instructions by the memory unit, allowing their execution to be overlapped. Load and store data are buffered by FIFO queues within the load-store unit of each vector lane. The memory unit generates addresses for vector memory accesses after receiving and decoding a memory instruction, and controls the memory alignment crossbar to align data to and from memory. The memory alignment crossbar supports memory accesses in granularity of word, halfword and byte, with the configurable parameter *MemMinWidth* specifying the smallest width data that can be accessed for all vector memory addressing modes. The memory crossbar can align up to 16 data elements per cycle for unit stride and constant stride loads, and 4 elements per cycle for stores. Indexed offset accesses execute at one data element per cycle. The vector operation bypass path allows the memory alignment crossbar to be used for vector manipulation instructions such as extracting part of a vector. The memory system is intended to be connected to an external 128-bit DDR-SDRAM module, which is suited for burst reading and writing of long vectors, or to large on-chip SRAMs.

The soft vector processor adopts a vector instruction set similar to the VIRAM instruction set, including 45 vector integer arithmetic, logical, memory, and vector and flag manipulation instructions. For nearly all instructions, the instruction opcode selects one of two *mask registers* to provide conditional execution. Complex execution masks are formed by special instructions that manipulate several *flag registers*. Some flag registers are general-purpose, while others hold condition codes from vector comparisons and arithmetic.

3.1 FPGA-Specific Vector Extensions

We extended the VIRAM-based vector architecture to take advantage of on-chip memory blocks and hardware MAC units common in FPGAs. The on-chip memory blocks are used in the AES benchmark, and the MAC units in the sample FIR filter and motion estimation benchmark.

A local memory is generated for each vector lane if *LMemN* is greater than zero. This local memory is non-coherent, and exists in a separate address space from main memory. Each vector lane supplies the address to access its own local memory. Like the distributed vector register file, it is normally split into 4 separate sections — one for each of the four data elements in a vector lane. However, if *LMemShare* is On, the four sections are merged, and the entire local memory becomes shared between all the elements that reside in the same lane. This mode is intended for table-lookup applications that share the same table contents between data elements. The memories can also be written by the scalar processor through a broadcast operation that writes the same value to all local memories (possibly to different addresses). *LMemW* specifies the data width of this memory.

In addition to the multipliers in the vector ALUs, the MAC feature of the Stratix III DSP blocks is used to construct distributed multiply-accumulators, also shown in Figure 4. The MAC units are especially useful for accumulat-

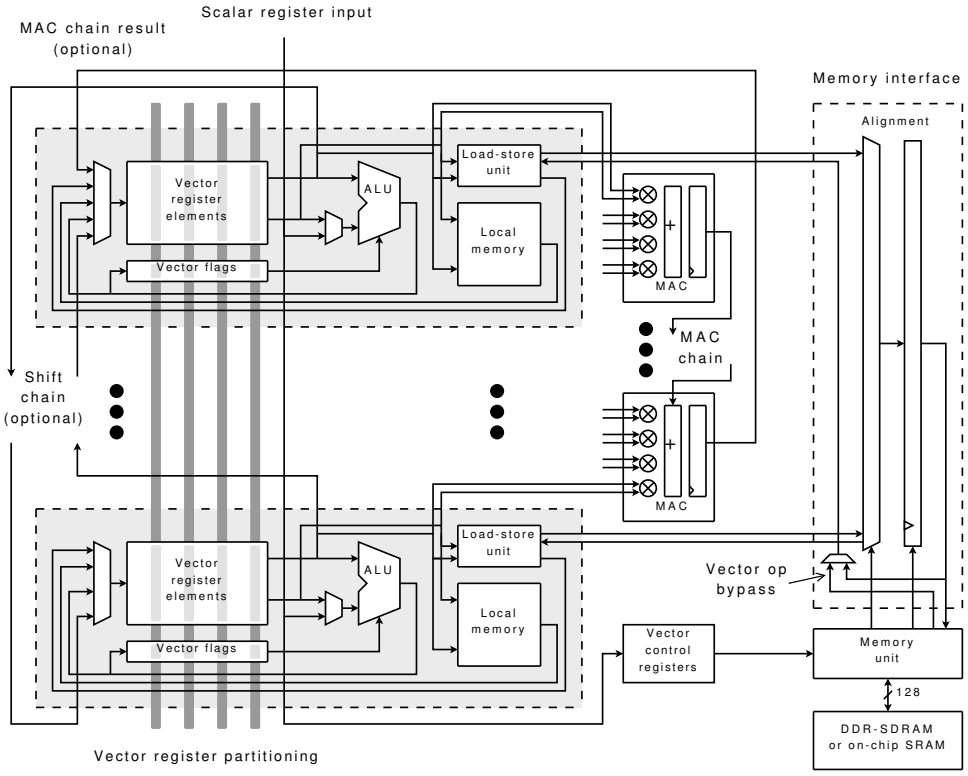

Figure 4: Vector co-processor system block diagram

ing multi-dimensional data. The `vmac` instruction multiply-accumulates 4 pairs of inputs from 4 vector lanes inside each MAC unit. Furthermore, the cascade chain in the Stratix III DSP blocks allows cascade adding of partial accumulation results across several accumulators, speeding up the otherwise inefficient vector reduction operation. *MACL* specifies the number of accumulators chained together to produce one accumulation result. The `vcczacc` instruction cascade adds the accumulator results in the MAC chain, copies the final result (partial results if the cascade chain does not span all MAC units) to a vector register, and zeros the accumulators.

4. DESIGN EXAMPLES

Three benchmarks representative of data-parallel embedded applications are chosen to demonstrate the ease-of-use and advantages of scalable vector processing. For each of the examples below, the *V8F* configuration of the soft vector processor is presumed. The assembly code for other configurations may be slightly different due to a different maximum vector length.

4.1 Block Matching Motion Estimation

Block matching motion estimation removes temporal redundancy within frames to provide coding systems with a high compression ratio. The algorithm divides each luma frame into blocks of size $N \times N$, and matches each block in the current frame with candidate blocks of the same size within a search area in the reference frame. The best matched block has the lowest distortion among all candidate blocks, and the displacement of the block, or the motion vector, is used to encode the video sequence. The metric is typically sum of absolute differences (SAD),

$$SAD(m,n) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |c(i,j) - s(i+m, j+n)|.$$

A full search block matching algorithm (FSBMA) matches the current block c to all candidate blocks in the reference frame s within a search range $[-p, p-1]$, and finds the motion vector of the block with minimum SAD among $(2p)^2$ search positions. Figure 5 shows example C code for the motion estimation kernel.

In a vector processor implementation, one of the dimensions is handled by vectorizing (removing) the innermost loop. With 8 lanes and *MVL* of 32, two windows separated by 16 pixels can be matched against the current block simultaneously, cutting the number of iterations in half. Figure 6 shows the vector code for the inner loop, plus vector code in the next outer loop to extract and accumulate results after processing the entire 16×16 window. The assembly code uses the MAC chain to reduce partial results in different accumulators to one final result as part of the `vcczacc` instruction. The ".1" instruction extension indicates conditional execution using `vf1` as the mask. The mask selects which of the two partial sums from the two windows to accumulate. This simple implementation requires 6 instructions in the innermost loop. To further improve performance, the number of memory accesses can be greatly reduced by unrolling the loop so entire rows of pixels can be loaded at a time from the reference frame, and so pixels from the cur-

```
for(l=0; l<nVERT; l++)
    for(k=0; k<nHORZ; k=++) {
      answer = 0;
      for(j=0; j<16; j++)
        for(i=0; i<16; i++)
          answer += abs(x[j][i] - y[l+j][k+i]);
      result[l][k] = answer;
    }
```

Figure 5: Motion estimation C code

```
vmov        v1, vzero              ; Zero sum
vmstc       r11, vbase0            ; Load x base
vmstc       r8, vbase1             ; Load y base

.L16:                              ; Innermost loop over rows
vld.b       v2, vbase0, vinc0      ; vinc0 = WINDOWSIZE
vld.b       v3, vbase1, vinc1      ; vinc1 = XSIZE
vabsdiff    v4, v2, v3
vadd        v1, v1, v4             ; Accumulate to sum
addi        r12, r12, 1            ; j++
bge         r13, r12, .L16         ; Loop again if (j<=15)

vfmov       vf1, vf2               ; Flag to mask data values
vmac.1      v1, vone               ; Accumulate across sum
vcczacc     v5                     ; Copy from accumulators
vext.vs     r5, v5                 ; Extract result to scalar core
```

Figure 6: Motion estimation vector assembly

rent block are loaded only once. To slide the window horizontally, the rows from the reference frame can be shifted using vector element shift. The unshifted rows of pixels can also be kept in the large number of vector registers to avoid additional pixel loads when the window shifts vertically to the next row.

4.2 Image Median Filter

The median filter is commonly used in image processing to reduce noise in an image and is particularly effective against impulse noise. It replaces each pixel with the median value of surrounding pixels within a window. Figure 8 shows example C code for a simple median filtering algorithm by calculating the median of a 5×5 image region. It essentially performs a bubble sort, stopping early when the top half is sorted to locate the median.

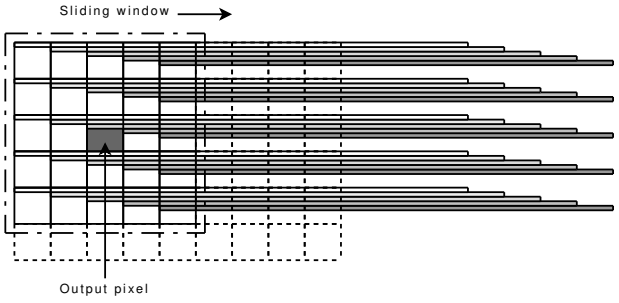

Figure 7: Vectorizing the image median filter

```
for (i=0; i<=12; i++) {
        min = array[i];
        for (j=i; j<=24; j++) {
                if (array[j] < min) {
                        temp = min;
                        min = array[j];
                        array[j] = temp;
                }
        }
}
```

Figure 8: 5×5 median filter C code

```
vmstc       vbase0, r1             ; load address of array[j]
vld.b       v1, vbase0             ; vector load array[j]
vld.b       v2, vbase1             ; vector load min
vcmpltu     vf1, v1, v2            ; compare, set vector flags
vmov.1      v3, v2                 ; copy min to temp
vst.b.1     v1, vbase1             ; vector store array[j] to min
vst.b.1     v3, vbase0             ; store temp to array[j]
```

Figure 9: Median filter inner loop vector assembly

One method to vectorize this kernel by exploiting outer-loop parallelism is shown in Figure 7. Each strip represents one row of *MVL* number of pixels, and each row is loaded into a separate vector register. The window of pixels that is being processed will then reside in the same data element over 25 vector registers. After initial setup, the same filtering algorithm can then be used. The vector processor uses masked execution to implement conditionals, and will execute all instructions inside the conditional block every iteration. Figure 9 shows the inner loop vector assembly, excluding address calculation and loop control scalar instructions. `vbase1` is initialized in the outer loop to the address of *min*. This implementation of the median filter can generate as many results at a time as *MVL* supported by the processor. *V8F* will generate 32 results at once, achieving a large speedup over scalar processing. This example highlights the importance of outer-loop parallelism, which the vector architecture, with help from the programmer, can exploit.

4.3 AES Encryption

The 128-bit AES Encryption algorithm [17] is a block cipher, and has a fixed block size of 128 bits. Each block of data can be logically arranged into a 4×4 matrix of bytes, termed the AES state. A 128-bit key implementation, which consists of 10 encryption rounds, will be illustrated in this example. Each round in the algorithm consists of four steps: `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`.

The first 3 steps can be implemented efficiently on 32-bit processors using a single 1 KB lookup table. A single round is then accomplished through four table-lookups, three byte rotations, and four EXOR operations [18].

The vector assembly code for loading data and for two rotate-lookup steps of a round transformation is shown in Figure 10. An implementation on the soft vector processor can first initialize all local memories with the substitution lookup table through broadcast from the scalar core. The AES state can be loaded from memory with four stride-four, load word instructions, which will load the four columns of multiple 128-bit AES blocks into four vector registers. Each

```
vlds    v1, vbase0, vstride4, vinc1   ; stride 4 word load one
                                      ; AES state column (4 times)

; Begin encryption round loop
vlds    v12, vbase1, vstride0, vinc1  ; stride 0 word load one
                                      ; round key column (4 times)
vmov        v8, v4          ; copy last column of AES state
vsrl.vs     v8, v8, r3      ; shift right 24 bits
vldl        v8, v8          ; S−box table lookup
vrot.vs     v8, v8, r8      ; rotate byte
vmov        v7, v3          ; copy 3rd column of AES state
vsrl.vs     v7, v7, r2      ; shift right 16 bits
vand.vs     v7, v7, r7      ; mask with 0x000000ff
vldl        v7, v7          ; S−box table lookup
vrot.vs     v7, v7, r8      ; rotate byte
vxor        v8, v8, v7      ; XOR 2 columns
```

Figure 10: Vector assembly for loading AES data and performing AES encryption round

AES block will then reside within a single vector lane, across the four vector registers. All the loaded AES blocks perform parallel table-lookup using the local memories. Assuming the round keys have already been generated, a total of 92 instructions are needed to perform the round transformation in the implementation. By exploiting outer-loop parallelism in the round transformation and encrypting blocks in parallel, the vector processor achieves speedup over scalar processing. A vector processor with 8 lanes has MVL of 32, and can encrypt 32 blocks or 4096 bits of data in parallel.

5. PERFORMANCE ESTIMATE

In this section, the three kernels are analyzed to produce performance estimates under idealized assumptions on the vector processor, Nios II, and Nios II with C2H compiler.

All of the instructions for the vector processor have been implemented and tested individually. We have simulated 70% of the instructions under more rigorous testing conditions, and are proceeding to verify the processor in hardware. We present our results as "estimates" that may be subject to minor fluctuation due to the incomplete testing and idealized assumptions made to simplify analysis.

5.1 Methodology

Performance of the different systems is estimated from instruction count, clock cycle count, and operating frequency. Instruction counts are obtained from compiling the kernels using the Nios II version of gcc (nios2-elf-gcc 3.4.1), with optimization O3, and manually counting the number of resulting assembly instructions. The vector assembly code is hand-written by substituting vector instructions into the Nios II assembly sources where applicable. Performance of the C2H compiler is calculated for the main loop from loop latency and cycles per loop iteration (CPLI) given in the compiler performance report.

The benchmarks include only the main loop section of the kernels. The median filtering kernel calculates one output pixel from the 5×5 window. The motion estimation kernel calculates SAD values for each position of a $[-16, +15]$ search range and stores the values into an array. It makes no comparisons between the values. The AES encryption kernel computes all rounds of encryption on 128 bits of data. Only the instruction and cycle counts for one intermediate round of the 10-round algorithm is reported, as the final round is handled differently and is not within the main loop.

Table 2: Vector instruction cycle model

Instruction Class	Instruction Cycles
Non-memory	$\lceil VL/NLane \rceil$
Local memory	$\lceil VL/NLane \rceil$
Memory store	$\lceil VL/NLane \rceil$
Memory load	$2 + \lceil VL/min(NLane, MaxElem) \rceil + \lceil VL/NLane \rceil$ $= 2 + 2 * \lceil VL/NLane \rceil$
$MaxElem$	$128/ElemWidth(bits)$

Table 3: Resource usage

Stratix III (C3)	ALM	DSP Elements	M9K	Fmax
EP3SL50 device	19,000	216	108	–
EP3SL340 device	135,000	576	1040	–
Nios II/s	489	8	4	153
UTIIe	324	0	3	193
UTIIe+V4F	5215	21	32	115
UTIIe+V8F	7011	34	53	114
UTIIe+V16F	10,266	58	95	113
UTIIe+V16M32	7384	34	93	112
UTIIe+V16W16	6743	54	69	109
UTIIe+V32M32	14,436	66	177	108
UTIIe+V32W16	10,298	102	127	107
Stratix II (C3)	ALM	DSP Elements	M4K	Fmax
Nios II/s + C2H				
Median Filtering	825	8	4	147
Motion Estimation	972	10	4	121
AES Encryption	2480	8	6	119

An Idealized Nios II processor is used as the baseline for performance comparison, while a Nios II/s processor is used for area and Fmax estimates. It is configured with 1 KB instruction cache, 64 KB each of on-chip program and data memory and no debug core. Area estimates are obtained from compiling the Nios II processors and the soft vector processor prototypes in Quartus II 7.2, and Fmax estimates are obtained from TimeQuest using the Slow 85C model.

We assume single-cycle execution of Nios II assembly instructions in both the Idealized Nios II and the vector processor; this presumes perfect caching and branch prediction. Vector instructions are separated into different classes, each taking a different number of cycles. Table 2 shows the number of cycles needed for each vector instruction class. Memory store instructions take the same number of cycles as non-memory instructions due to write buffering. The first non-constant term in memory load cycles models the number cycles needed to transfer data from memory to the load buffer. $MaxElem$ is the maximum number of data elements that can be transferred per cycle through the 128-bit memory interface. The second term models transferring data from load buffer to the vector register file. Note that the number of memory load cycles simplifies to $2 + 2 * \lceil VL/NLane \rceil$ when $NLane$ is the limiting factor. The architecture can overlap execution of vector arithmetic and vector memory instructions, or Nios II and vector instructions, but this enhancement is not considered by this simple performance model.

5.2 Vector Performance

Table 3 shows the estimated resource usage of several configurations generated for the benchmarks and illustrates the

Table 4: Performance summary

	Idealized Nios II	Nios II + C2H	Proposed Vector Architecture V4F	V8F	V16F
Fmax (MHz)					
Median Filtering	153	147	115	114	113
Motion Estimation	153	121	115	114	113
AES Encryption Round	153	119	115	114	113
Dynamic Instruction Count					
Median Filtering (per pixel)	5,375	n/a	275	137	69
Motion Estimation (per SAD value)	2,481,344	n/a	113,856	62,733	62,768
AES Encryption Round (per 128-bit block)	94	n/a	5.9	2.9	1.5
Clock Cycles					
Median Filtering (per pixel)	5,375	2,101	784	392	196
Motion Estimation (per SAD value)	2,481,344	694,468	492,736	260,050	160,682
AES Encryption Round (per 128-bit block)	94	25	23	12	6
Speedup					
Median Filtering (per pixel)	1	2.5	5.2	10.2	20.2
Motion Estimation (per SAD value)	1	2.8	3.8	7.1	11.4
AES Encryption Round (per 128-bit block)	1	2.9	3.1	6.1	12.0

scalable nature of the soft vector processor. *Vx* in the vector processor name specifies the number of lanes: *V4* has 4 vector lanes with *MVL=16*, *V8* has 8 lanes with *MVL=32*, and *V16* has 16 lanes with *MVL=64*. *VxF* configurations have full support of all features. The *Wx* tag indicates the bit width of the vector datapath, i.e. *W16* is 16 bits. The *M32* tag indicates that only 32-bit words are supported for vector memory accesses (no bytes or halfwords). The flexible memory interface which supports bytes, halfwords, and words is the single largest component, using 35% of the ALMs in the *V16F* processor. The complex control logic of the memory interface, needed to support arbitrary strided access, forms the critical path that prevents higher Fmax. Reducing the memory interface to 32-bit word access only, as in the *V16M32* configuration, leads to a large savings in area.

Table 4 shows estimated performance of the three sample *VxF* processors on the benchmarks measured by instruction count, clock cycle count, and speedup over the Idealized Nios II baseline. All three vector configurations show significant speedup, where greater performance is obtained when more vector lanes are used. The instruction count and clock cycle per result decreases for median filtering and AES encryption as more vector lanes are added, since more results are computed in parallel. In particular, the fractional instruction counts for AES encryption result from dividing the total instructions by the number of blocks encrypted in parallel. For the vectorized motion estimation kernel, VL is 16 for *V4*, and is 32 for both *V8* and *V16* when processing two search areas simultaneously. Instruction and cycle count per result decreases going from *V4* to *V8* due to parallel processing of two search areas. The *V16* configuration also processes two search areas, but requires more instructions to sum a vector across more lanes. Overall, however, *V16* produces additional speedup because the extra lanes reduce clock cycles needed to process the same vector length.

The soft vector processor achieves scalable speedup in all three benchmarks with performance proportional to resource usage. The vector assembly code is also able to take advantage of more vector lanes to improve performance with little or no modification.

5.3 C2H "Push-button" Performance

The resources and performance results for C2H-generated hardware accelerators of the three benchmarks are also shown in Tables 3 and 4, respectively. These C2H numbers represent results achieveable by "push-button" acceleration with the compiler, with no modification to the original C code.[1]

The "push-button" C2H acceleration results are similar to those of the *V4F* processor, but they do not match those of the larger vector processor configurations.

5.4 C2H "Extra-effort" Performance

Applying vector processing concepts and loop unrolling makes it possible to increase C2H performance at the expense of increased resources. However, to get good performance, it is necessary to understand how it maps C to hardware.

The documentation clearly shows how each C statement is translated to hardware. While this is helpful to hardware designers, it is potentially confusing to software designers. Dependent assignment statements form a pipelined datapath, with every complex operator or assignment statement being registered. This pipelined datapath is automatically scheduled by the compiler. Every memory reference (load or store) turns into a dedicated port to the memory system. These ports automatically compete for memory access through arbiters in the Avalon system fabric.

The current C2H implementation also has some limitations. Instead of unrolling loops to form deeper or wider pipelines, C2H turns iteration into a state machine. In limited situations, C2H can generate parallel memories if they are entirely used by the accelerator and never accessed by the software portion. However, it does not automatically partition data across parallel memory banks, so memory arbitration quickly becomes a bottleneck. Fixing this requires manual effort: multiple memory banks must be created in SOPC Builder, and data partitioned using pragmas in C.

[1]Stratix II was used for the C2H estimates because Quartus II does not yet support C2H hardware accelerators for Stratix III-based designs. We expect results to be similar to Stratix II due to the similar ALM architecture.

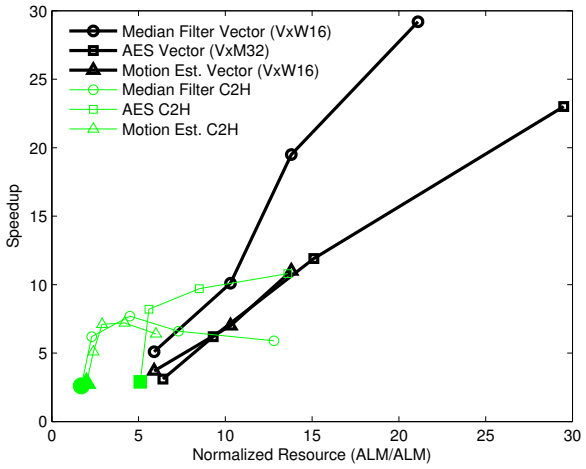

Figure 11: Speedup versus resource utilization

In this section, we performed manual loop unrolling, but we did not perform data partitioning because it is cumbersome and time-consuming.

For the median filter, we used the same technique as the vector assembly code to find up to 64 medians in parallel. This creates a memory bottleneck because each iteration requires 64 new data items to be read, and 64 new values to be written back.

For motion estimation, we moved the horizontal-move outer loop inside and unrolled it. This results in one pixel of the moving block being compared to up to 32 possible horizontal locations in parallel, creating 32 accumulators for the *SAD* operation. One pixel datum is loaded from memory to register once, then re-used 32 times by the 32 accumulators. Hardware knowledge is needed to know that the 32 accumulators will be inferred to hold the intermediate results. However, reading from 32 possible horizontal locations creates a memory bottleneck.

For AES, we knew memory access to the large lookup tables would be a bottleneck. To solve this, we added 4 memory blocks for the 256-entry, 32-bit table lookups in the Nios II system. We replicated the AES engine up to four times, but these contended for the same lookup table memories. Memory access remains a bottleneck; resolving it would require a dedicated copy of all tables for each engine. Alternatively, unrolling the 10 AES rounds would also require 10× copies of the table memories.

5.5 C2H versus Vector

The C2H compiler results for the three benchmarks are compared against the soft vector processor results in Figure 11. The figure shows speedup versus ALM usage over the Nios II baseline for the various C2H and vector processor configurations. Speedup versus area "push-button" C2H results are shown as 3 solid-gray markers. The thin/gray lines show the performance improvement when "extra-effort" is expended with C2H, which required some hardware design knowledge. Notice that performance saturates as memory access becomes the bottleneck. Additional performance would require additional hardware-aware design effort and a change to the Nios II memory system. Also, if these different applications must run on the same FPGA device, a custom memory system for each application might be needed. This could overwhelm the on-chip memory resources, at which point multiple device configurations would be needed.

In contrast, the bold/dark lines show the how the vector processor results are scalable and obtained with the same unified memory system and zero hardware design effort. To save resources, application-specific configurations of the vector processor were created. Median filtering and motion estimation do not require any 32-bit vector processing, so the *VxW16* configurations are used. The AES encryption kernel only requires 32-bit word memory access, so the *VxM32* configurations, which lack byte and halfword memory access crossbars, are used. The C2H co-processors achieved 3–11× speedup over the baseline Nios II processor using 2–14× the number of ALMs. The soft vector processor achieved 3–29× speedup using 6–30× the number of ALMs.

As a rough comparison of effort, it took approximately 3 days to learn to use the C2H compiler, modify the three benchmark kernels so they compile, and apply the simple loop unrolling software optimization. It took another full day to apply the single hardware optimization for the AES benchmark of adding the additional memories. With the vector processor, it took 2 days to design vectorized algorithms and assembly code for all three kernels. After the initial learning to think in vector, it took less than half a day to design a revised AES vector algorithm (which we had to do) and rewrite the assembly.

6. ARCHITECTURAL SUGGESTIONS

The architecture of FPGAs is well-suited for SIMD and vector computing. While targetting the Stratix III, we noted a few architectural features that could be improved in this family to create better soft vector processors.

High-performance register files usually need 2 read ports and 1 write port. The read ports are implemented by duplicating the register file memory. For a small 32-bit soft-core processor, this is a modest overhead, but duplicating the large register file of a vector processor is costly.

DSP blocks in Stratix III are optimized for 16-bit fixed-point operations. The narrow data types forced us to restrict certain instructions, such as multiply-accumulate, to only 16-bit inputs even in the 32-bit processor. A MAC unit that can switch between 16 and 32-bit inputs would be useful.

The cascade adder chain in the Stratix III DSP blocks is useful for the accumulate reduction operation. However, we could not use the shift chain at the DSP block inputs because the shift mode cannot be dynamically selected at runtime. This prevents the shift chain from being useful in the vector architecture. Stratix II supported this feature.

From prototyping the soft vector processor, the single structure that consumes the most resources is the byte-level crossbar to rearrange data from vector lanes to their correct positions in the 128-bit datapath to memory. This is needed for strided memory access, as well as writes to non-128-bit aligned memory locations. Since datapath structures are so prevelant in this design, datapath-oriented FPGA structures could reduce resource usage and improve clock speed.

7. CONCLUSION

Soft-core CPUs offer limited performance for data-parallel applications. This type of parallelism can be exploited in an FPGA using custom hardware accelerators. As an alternative, vector processing can also accelerate this type of parallelism. The key advantage of the vector approach is that it can be employed by software developers without any hardware design knowledge. Also, a single vector processing unit can be used to accelerate several different tasks with the same FPGA bitstream. In contrast, a custom accelerator is usually good for only one task, requiring the designer to design and integrate several accelerators for multiple tasks.

Sophisticated tools like Altera's C2H compiler greatly simplify the design of custom-built accelerators. However, fully exploiting the tool requires some hardware design knowledge. Memory bandwidth is frequently a bottleneck, and solving this currently requires manual intervention at the hardware design level. While a custom memory system can be defined for each application, this can become increasingly cumbersome when several accelerators are required simultaneously. Ultimately, this may result in the need for several device configurations as well.

In contrast, vector processing can be used as a purely software-oriented solution to many problems. This does not eliminate the usefulness or need for a tool like C2H, but it provides a viable alternative when hardware designers are busy on other projects. A soft-core vector processor is most suitable when rapid development time is required, or when a hardware designer is not available, or when several different applications must share a single accelerator or a single FPGA bitstream. It offers a simple programming model that can be readily understood by software developers with little or no hardware design knowledge. It is also easy to scale performance with little or no change to the software by only modifying a few simple processor parameters. Scaling the number of vector lanes naturally offers both more memory bandwidth (at the register file) and more functional units. Scaling performance with C2H required more extensive hardware and software changes to match the computational power to the available memory bandwidth.

The FPGA-based soft vector processor architecture proposed in this paper efficiently maps to a Stratix III FPGA. Three specific changes were made to better exploit FPGAs: the register file was partitioned across multiple vector lanes, MAC hardware units were used to improve accumulate reduction operations, and local memory blocks were added in each vector lane to accelerate table-lookup applications. The first optimization was necessary to create an area-efficient register file, while the latter two are used to improve benchmark performance. The ability to customize several aspects of a soft vector processor for the needed applications provides further ability to trim area.

Acknowledgments

The authors would like to thank Blair Fort for providing the UTIIe processor for this research. We would also like to thank Ralph Wittig and Jorn Janneck for their helpful feedback on early drafts of this work. This research was partially supported by funding from NSERC.

8. REFERENCES

[1] Nios II. [Online]. Available: http://www.altera.com/products/ip/processors/nios2/ni2-index.html

[2] K. Diefendorff and P. Dubey, "How multimedia workloads will change processor design," *Computer*, vol. 30, no. 9, pp. 43–45, September 1997.

[3] D. Lau, O. Pritchard, and P. Molson, "Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions," in *IEEE Symp. on Field-Programmable Custom Computing Machines*, April 2006, pp. 45–56.

[4] K. Asanovic, "Vector microprocessors," Ph.D. dissertation, EECS Department, University of California, Berkeley, 1998.

[5] H. Yang, S. Wang, S. G. Ziavras, and J. Hu, "Vector processing support for FPGA-oriented high performance applications," in *Int. Symp. on VLSI*, March 2007, pp. 447–448.

[6] J. Cho, H. Chang, and W. Sung, "An FPGA based SIMD processor with a vector memory unit," in *Int. Symp. on Circuits and Systems*, May 2006, pp. 525–528.

[7] Microblaze. [Online]. Available: http://www.xilinx.com/

[8] D. Talla and L. John, "Cost-effective hardware acceleration of multimedia applications," in *Int. Conf. on Computer Design*, September 2001, pp. 415–424.

[9] J. Gebis and D. Patterson, "Embracing and extending 20th-century instruction set architectures," *Computer*, vol. 40, no. 4, pp. 68–75, April 2007.

[10] C. Kozyrakis and D. Patterson, "Scalable, vector processors for embedded systems," *IEEE Micro*, vol. 23, no. 6, pp. 36–45, Nov./Dec. 2003.

[11] Cascade, Critical Blue. [Online]. Available: http://www.criticalblue.com

[12] D. Bennett, "An FPGA-oriented target language for HLL compilation," Reconfigurable Systems Summer Institute, July 2006.

[13] P. Yiannacouras, J. G. Steffan, and J. Rose, "Application-specific customization of soft processor microarchitecture," in *ACM/SIGDA Int. Symp. on FPGAs*, February 2006, pp. 201–210.

[14] The embedded microprocessor benchmark consortium. [Online]. Available: http://www.eembc.org/

[15] C. Kozyrakis and D. Patterson, "Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks," in *IEEE/ACM Int. Symp. on Microarchitecture*, November 2002, pp. 283–293.

[16] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown, "A multithreaded soft processor for SoPC area reduction," in *IEEE Symp. on Field-Programmable Custom Computing Machines*, April 2006, pp. 131–142.

[17] "Specification for the advanced encryption standard (AES)," Federal Information Processing Standards Publication 197, 2001.

[18] J. Daemen and V. Rijmen, *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.