A Case for Soft Vector Processors in FPGAs

Jason Yu Guy Lemieux
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada V6T 1Z4
Email: {`jasony,lemieux`} @ *ece.ubc.ca*

Abstract

Embedded applications today require high computational power that is not met by current FPGA-based soft processors. Although performance of data-parallel applications can be addressed by custom-designed hardware accelerators, such an approach is difficult for embedded software developers with little hardware design experience. Instead, vector processing can be used to speed up these same data-parallel applications. The vector programming model is easy to understand by software developers, making it easier for them to extract the parallelism without any hardware design knowledge. This paper proposes a soft vector processor for the Stratix III FPGA that can be scaled to different levels of performance and resource utilization. It has several configurable features that can be included or excluded to optimize the soft processor for a given application. Performance estimates of the soft vector processor using three embedded benchmark kernels show speedup of up to $16.6\times$ *over an idealized Nios II processor while using* $10.9\times$ *the area.*

1. Introduction

Performance of software applications is determined by a few hotspots in the program. This is especially true for embedded media applications, which tend to have tight, computationally-intensive inner loops [1]. However, current commercial FPGA-based soft-core processors such as MicroBlaze and Nios II provide only limited and non-scalable performance. The effectiveness of traditional architectural approaches to improve soft-core processor performance is limited by how well these architectures can be mapped into the FPGA fabric. Techniques such as deep pipelining and wide-issue superscalar do not work well in FPGAs, usually because any potential performance gains they offer are negated by reduced clock speed or prohibited by the FPGA architecture itself. For example, it is difficult to implement the several write ports needed by a superscalar register file.

Instead, a vector architecture can be used to accelerate applications that are rich in data parallelism.

Table 1. Configurable processor parameters

Parameter	Description	Typical	V4/V8/V16
NLane	Number of vector lanes	4–128	4/8/16
MVL	Maximum vector length	16–512	16/32/64
VPUW	Processor data width (bits)	8,16,32	32
MultW	Multiplier width (bits, 0 is off)	0,8,16,32	16
LMemN	Local memory number of words	0–1024	256
LMemW	Local memory width (bits)	8,16,32	32

Although such applications would clearly benefit from custom-designed hardware accelerators, doing this would require a hardware designer to create and debug a different accelerator for each application. In contrast, a vector processor can be easily programmed by software developers without any hardware experience. Previous work such as [2], [3] have successfully implemented vector processing in FPGAs for specific tasks. In this work, we focus on creating a soft vector architecture with a more general instruction set. We describe a range of soft vector processors using a few key architectural parameters, enabling a software developer to specify one implementation instance to meet given performance and area requirements. Furthermore, one soft vector processor instance can speed up several different data-parallel applications with no additional hardware design or device reconfiguration.

We propose that FPGA soft-core processors adopt vector processing as a parallel programming model to more efficiently run embedded software applications containing significant data-level parallelism. To facilitate this, we demonstrate that a soft vector processor maps efficiently into an FPGA. Furthermore, we show that a wide range of configurable options are possible, allowing a significant range of resource usage and performance scalability that is unmatched by traditional soft-core processors.

2. Soft Vector Processor Architecture

A soft vector processor architecture contains a family of vector processors with varying performance and resource utilization, and different features to suit different applications. A generator program uses a

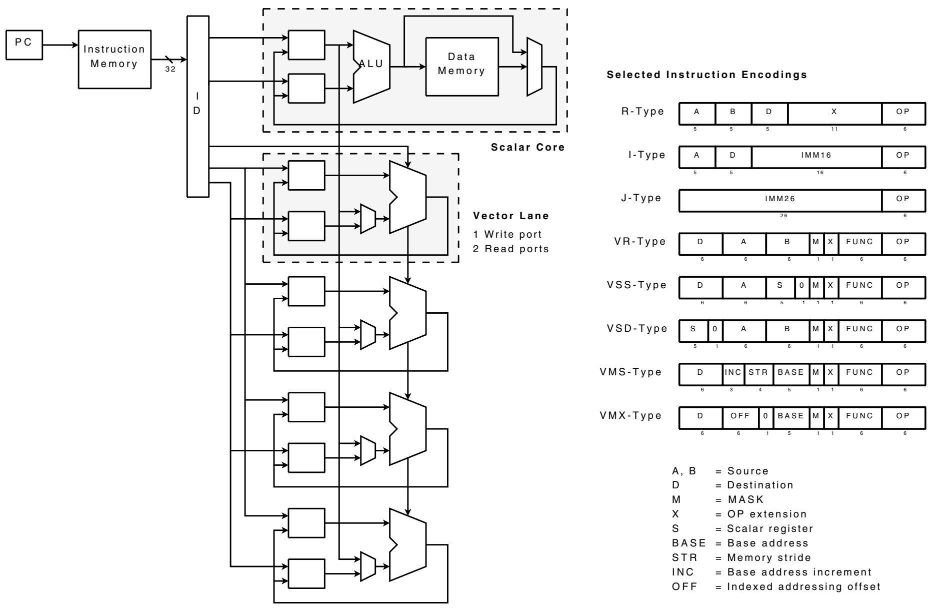
PC
Instruction Memory
32
I D
ALU
Data Memory
Scalar Core
Vector Lane
1 Write port
2 Read ports
Selected Instruction Encodings
R-Type: A 5 | B 5 | D 5 | X 11 | OP 6
I-Type: A 5 | D 5 | IMM16 16 | OP 6
J-Type: IMM26 26 | OP 6
VR-Type: D 6 | A 6 | B 6 | M 1 | X 1 | FUNC 6 | OP 6
VSS-Type: D 6 | A 6 | S 5 | 0 1 | M 1 | X 1 | FUNC 6 | OP 6
VSD-Type: S 5 | 0 1 | A 6 | B 6 | M 1 | X 1 | FUNC 6 | OP 6
VMS-Type: D 6 | INC 3 | STR 4 | BASE 5 | M 1 | X 1 | FUNC 6 | OP 6
VMX-Type: D 6 | OFF 6 | 0 1 | BASE 5 | M 1 | X 1 | FUNC 6 | OP 6
A, B = Source
D = Destination
M = MASK
X = OP extension
S = Scalar register
BASE = Base address
STR = Memory stride
INC = Base address increment
OFF = Indexed addressing offset

Figure 1. Interaction between scalar and vector cores

number of parameters to define an application-specific instance of the processor. The configurability of soft vector processors in FPGAs gives significant flexibility to trade-off performance and resource utilization. Resources can be further trimmed by removing unneeded processor features and instruction support.

Table 1 lists some of the configurable parameters and features of our soft vector processor architecture. The final resource usage of a soft vector processor is very sensitive to many of these parameters. V4, V8, and V16 are three sample configurations of the processor. Our implementation 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 1 and 2 illustrate our soft vector processor architecture. It 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 [4], 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 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 2. The vector unit is composed of a number of vector lanes, 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.

Different from traditional vector architectures that employ a large, centralized vector register file with many ports, the soft vector processor uses a distributed vector register file that is spread across vector lanes. The vector register file is element-partitioned—each vector lane has its own register file that contains all the vector registers in the complete register file, but only a few data elements of each vector register [5]. The partitioning scheme naturally divides the vector register file into parts that can be implemented using the smaller memory blocks on the FPGA, and allows simultaneous access to multiple data elements in the vector register file by functional units in the vector lanes. Furthermore, the distributed vector register file saves area compared to a large, multi-ported vector register file. *It is the abundance of these small memory blocks (and multipliers) that makes modern FPGAs exceptionally good at implementing vector processors.* The vertical gray stripes in Figure 2 represent one vector register. The data elements of the vector register are partitioned both across multiple lanes and within each lane, with up to 4 elements in a lane. The architecture defines 64 vector registers, which occupy exactly one M9K RAM when 4 elements per vector register are stored in each lane. For this reason, *MVL* is typically 4 times *NLane*. To provide dual read ports,

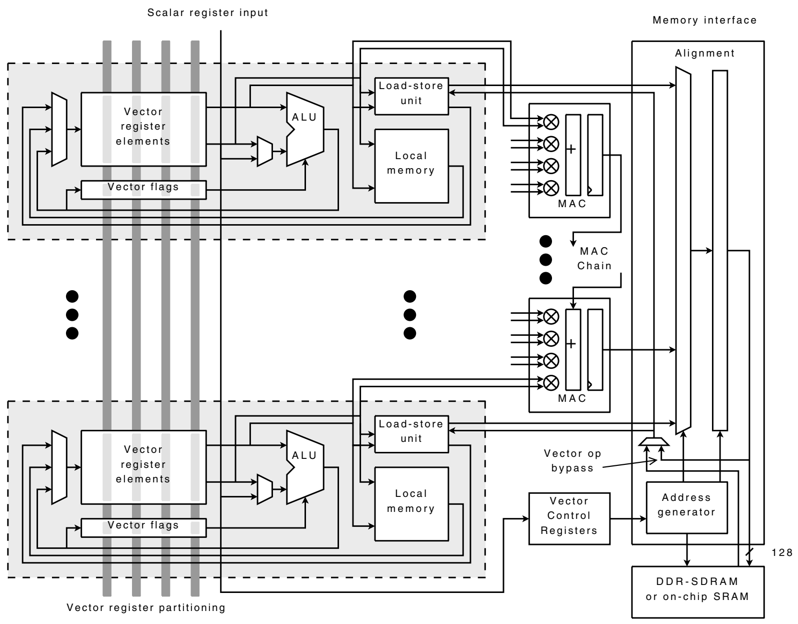

Figure 2. Vector coprocessor system block diagram

this M9K RAM is duplicated.

The memory interface unit handles memory accesses for both scalar and vector units. Scalar and vector memory accesses are performed in program order. No consistency checking against the scalar cache is performed. The address generator generates addresses for vector memory accesses, and controls the memory alignment crossbar to align data to and from memory. The crossbar supports memory accesses in granularity of word, halfword and byte. 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 128-bit memory system is intended to be connected to a large on-chip SRAM or an external DDR-SDRAM, both of which are well-suited for burst reading and writing of long vectors.

The soft vector processor adopts a vector instruction set similar to the VIRAM instruction set [6]. Conditional execution is accomplished using mask registers. For nearly all instructions, the processor can select one of two execution mask registers in the instruction opcode. A number of general-purpose vector flag registers that are written to by compare instructions and operated on by vector flag logical instructions allow manipulation of mask values for complex conditional operations.

Two extensions were made to VIRAM to exploit FPGA architecture for better performance: local memories and MAC chains. If *LMemN* is greater than zero, a local memory is generated for each vector lane. The vector lane local memory uses register indirect addressing, in which each vector lane supplies the address to access its own local memory. The memories can also be written to 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 the local memory and the maximum address width. If *LMemW* is less than *VPUW*, data to the local memory is truncated. These memories are useful for accelerating histogram or table-lookup operations such as AES encryption.

The second extension uses the MAC feature of the Stratix III DSP blocks to efficiently implement the addition reduction operation (i.e., sum all elements in a vector), as shown in Figure 2. The `vmac` instruction multiply-accumulates 4 pairs of inputs from 4 vector lanes into each MAC unit. Furthermore, the cascade chain in the Stratix III DSP blocks allows cascade adding of partial accumulation results across several accumulators, further accelerating the otherwise inefficient vector reduction operation. The `vccacc` instruction copies the result of the accumulate reduction to a vector register. The MAC feature is used, for example, in motion estimation to sum the absolute difference between pixels.

3. Performance

Three representative data-parallel embedded applications were chosen to benchmark scalable vector processing: 5×5 median filtering, motion estimation, and AES encryption. The benchmarks include only the main loop section of the kernels. Instruction counts are obtained from compiling the kernels using the Nios II version of gcc with –O3 optimization, and manually counting the number of assembly instructions. This

Table 2. Estimated performance results

		Proposed Vector Architecture			
	Nios II/s	V4	V8	V16	V16W16
Area Estimate					
ALM Count	489	4061	5450	8159	5306
DSP Elements Count	8	17	30	54	38
M9K Count	4	35	64	122	96
Fmax Estimate (MHz)	153	86	91	83	89
Instruction Count Estimate					
Median Filtering (per pixel)	5,375	275	137	69	69
Motion Estimation	2,481,344	113,856	62,733	62,768	62,768
AES Encryption Round (per 128-bit block)	94	5.9	2.5	1.5	n/a
Clock Cycle Estimate					
Median Filtering (per pixel)	5,375	753	377	188	188
Motion Estimation	2,481,344	492,736	260,050	150,442	150,442
AES Encryption Round (per 128-bit block)	94	23	12	6	n/a
Speedup Estimate					
Median Filtering (per pixel)	1	4.0	8.5	15.5	16.6
Motion Estimation	1	2.8	5.7	8.9	9.6
AES Encryption Round (per 128-bit block)	1	2.3	4.8	8.8	n/a

code was then manually modified to use vector instructions.

We estimated both Nios II and vector performance using idealized assumptions: scalar instructions take 1 cycle, non-memory vector instructions take $\lceil VL/NLane \rceil$ cycles, and vector memory instructions take $1+2\times\lceil VL/NLane \rceil$ cycles. Area and Fmax estimates are obtained from Quartus II 7.1. The Nios II/s processor is configured with 1 KB instruction cache, 64 KB each of on-chip program and data memory, no debug core, and a single off-chip PIO. The HDL for the vector processor is not yet fully debugged, so we present our results as "estimates". Speedup figures include the impact of clock frequency.

Table 2 shows the estimated performance of the different processors for the three applications. V16W16 is a 16 lane, 16-bit processor (*VPUW* of 16) with no local memory. It can execute the median filtering and motion estimation kernels as they do not require 32-bit processing, but not the AES encryption kernel. This configuration illustrates how an application-specific soft vector processor can yield significant resource savings, and an increase in performance. All four vector configurations show significant speedup over idealized Nios II.

4. Conclusion

Many embedded applications today demand high performance computing platforms that enable the programmer to easily exploit data-level parallelism. FPGAs are inherently parallel processors, but considerable design effort and hardware design knowledge is needed to design a custom parallel system in HDL. Current FPGA soft processors do not provide enough performance to meet these application requirements. A soft vector processor platform can address these deficiencies by providing a simple and familiar programming model to describe data-level parallelism for parallel computing. The FPGA-based soft vector processor proposed in this paper efficiently maps the vector architecture to a Stratix III FPGA. It leverages the configurability of FPGAs to allow the designer to trade-off performance level with resource usage, and to optimize the processor for the target application by generating an application-specific instance with only the needed features—all with zero hardware design.

Acknowledgment

The authors would like to thank Blair Fort for providing the UTIIe processor, and Christopher Eagleston for his help with analyzing the benchmarks.

References

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

[2] J. Cho, H. Chang, and W. Sung, "An FPGA based SIMD processor with a vector memory unit," in *ISCAS*, May 2006, p. 4.

[3] H. Yang, S. Wang, S. G. Ziavras, and J. Hu, "Vector processing support for FPGA-oriented high performance applications," in *ISVLSI*, 2007, pp. 447–448.

[4] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown, "A multithreaded soft processor for SoPC area reduction," in *FCCM*, 2006, pp. 131–142.

[5] K. Asanovic, "Vector microprocessors," Ph.D. dissertation, EECS Department, University of California, Berkeley, 1998. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1998/6404.html

[6] C. Kozyrakis and D. Patterson, "Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks," in *MICRO-35*, 2002, pp. 283–293.