

FPGA-based Vector Processing for Matrix Operations*

Hongyan Yang, Sotirios G. Ziavras and Jie Hu
New Jersey Institute of Technology
Department of Electrical and Computer Engineering
University Heights, Newark, NJ 07102
Email: {hy34, ziavras, jhu}@njit.edu

Abstract

A programmable vector processor and its implementation with a field-programmable gate array (FPGA) are presented. This processor is composed of a vector core and a tightly coupled five-stage pipelined RISC scalar unit. It supports the IEEE 754 single-precision floating-point standard and also the efficient implementation of some sparse matrix operations. The processor is implemented on the Xilinx XC2V6000-5 FPGA chip. To test the performance, the W-matrix sparse solver for linear equations is realized. W-matrix was first proposed for power flow analysis and is prone to parallel computing. We show that actual power matrices with up to 1723 nodes can be dealt with in less than 1.1ms on the FPGA. A comparison with a commercial PC indicates that the vector processor is competitive for such computation-intensive problems.

1. Introduction

Computation-intensive problems are a great challenge to general-purpose processors due to the latter's underlying sequential architecture. Application-specific integrated circuits (ASICs) with abundant calculation units are suitable for such tasks but it takes a long time to develop relevant systems; also, they are resilient to potential modifications required to fit new applications. This makes the ASIC approach prohibitively expensive for small production and drives designers to search for flexible solutions. On the other hand, in the last decade there have been significant FPGA improvements in logic resource capacity, speed and architectural features, thus presenting us with a configuration-based alternative to high-performance computing. Although FPGAs have been used in the past primarily for prototyping and digital glue-logic purposes, several recent high-performance computers contain FPGAs (e.g., Cray). Also, impressive performance improvement has been reported for applications running on reconfigurable computing systems containing FPGAs [1–8]. New generation FPGAs with million gates have also made feasible powerful System-on-chip (SOC) designs.

Vector processing is an advanced technique widely used in supercomputers to achieve high performance by exploiting regularities in array processing. In real-time applications, a vector processor may be the best choice if the application requires a heavy amount of calculations involving vectors; vector processors can provide high throughput by applying the same operation simultaneously to many array/vector elements [5, 9, 10]. For data parallel applications, a vector processor can easily outperform VLIW (very large instruction width) and superscalar processors at low cost [11]. An FPGA-based implementation of a vector processor would be a promising task.

In this paper, we present a programmable vector processor implemented on the Annapolis Wildstar-II FPGA board [12]. A vector register file having multiple ports is located in the center of our vector processor. By dividing it into several banks, a higher bandwidth can be provided in a much smaller area. The vector register file is divided into eight banks where each bank has two read ports and one write port. The arithmetic units and data memory are also organized in eight banks to match the vector register file structure. A larger number of elements in a vector register can reduce the effect of the startup time and speed up the execution for large vectors, but it also increases the circuit complexity and may cause a dramatic system frequency decrease. Vector registers with various numbers of elements were implemented, and their resource usage and resulting speed are reported.

The W-matrix equation solver for power flow analysis is used to benchmark our vector processor. It has been proposed as an efficient way to solve linear equations by changing sequential substitutions into matrix multiplications which can run in parallel [13, 14]. Some successful W-matrix solvers have run on shared-memory parallel computers [15], vector supercomputers [16, 17] and multiprocessors [18]. We show that the W-matrix method works efficiently on our FPGA-based vector processor. Real power network matrices are used to test our approach and the results are compared with those of a commercial PC.

*This work was supported in part by the U.S. Dept. of Energy under grant DE-FG02-03CH11171.

2. Architecture of the vector processor

The vector processor is composed of a vector core and a tightly coupled five-stage pipelined scalar unit as shown in Fig. 1. The scalar unit is organized as a Harvard architecture with separate bus interface units for instruction and data access. The scalar processor fetches and decodes instructions. It does the actual work for scalar commands and forwards the vector instructions to the vector core. The vector core is structured as eight parallel lanes, where each lane contains a portion of the vector register file, a floating-point multiplier, a floating-point adder and connection to the eight-bank memory system. It can produce up to eight results and get a maximum of eight data items from the memory banks per clock cycle. To focus on the actual vector design, the floating-point IP (Intellectual Property) cores were purchased from Quixilica [19]. We have designed our own assembler to translate programs written in assembly language into machine code targeting our system. The dramatically reduced code size resulting from vector processing makes our programming job easier.

2.1. Scalar unit

The scalar processor in our system supports 16 instructions for control, register and memory access, and arithmetic operations. There is a five-stage pipeline (fetch, decode, execute, memory access, and write back), as shown in Fig. 2. This scalar processor includes an arithmetic logic unit (ALU), a register file, a data hazard detection unit, and a data forwarding unit. For the sake of simplicity, Fig. 2 does not depict all the hardware. The shaded areas are unique to the vector system design; they are used to transfer useful information to the vector core. The two specialized registers in the register file are used to control vector operations. They are: vector-length register (VLR) and vector-mask register (VMR). VLR is used to control the length of vector operations and VMR indicates that

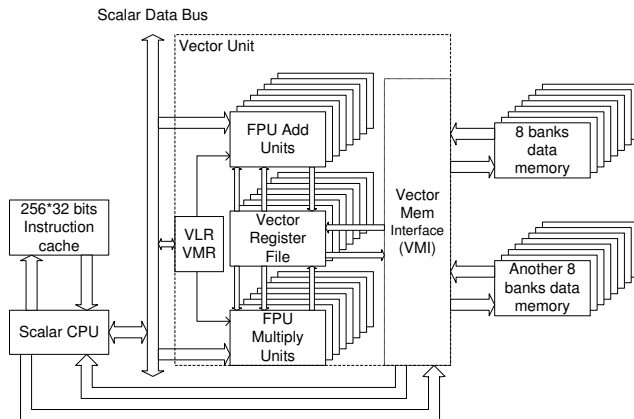


Figure 1. Block diagram of the vector processor (VLR: Vector Length Register, VMR: Vector Mask Register)

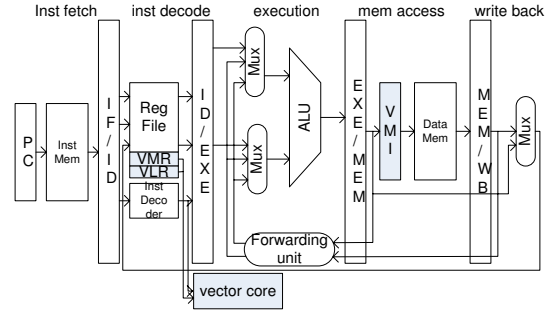


Figure 2. Scalar processor architecture

operations are to be applied only to the vector elements with corresponding entries equal to 1 in VMR.

To avoid EXE and MEM data hazards due to pipelining, data hazard detection and forwarding units are implemented. All scalar pipeline hazards can be avoided either with data forwarding or interlocking in hardware, so scalar instruction scheduling is not required for correctness; however, it may improve the performance. This greatly eases code writing for our processor.

2.2. Vector register file

The vector register file lies in the heart of the vector unit. It provides both temporary storage for intermediate values as well as interconnectivity between the vector floating-point units (VFUs) [9]. A straightforward way to implement the vector register file is to use a single multi-ported memory. But this is a very expensive solution that requires many logic resources and also increases the power consumption of the FPGA chip. Take the example of eight vector registers each having 32 32-bit elements; the left diagram in Fig. 3 shows the slice usage for a Xilinx XC2V6000 chip and the right one shows the power consumption assuming that it runs at 70MHz. We can observe that the slices will be used up quickly and the power consumption increases greatly for an increased number of ports. All the results presented in this paper are after the place-and-route step for the XC2V6000 chip.

To reduce the cost, we could divide the vector register

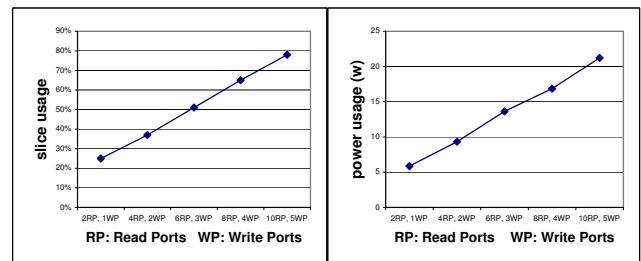


Figure 3. Resource and power consumption for single-block implementation of a vector register file containing 8 vector registers of 32 32-bit elements

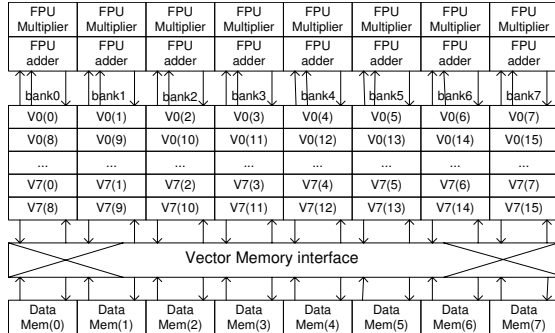


Figure 4. Vector register file organization

file into banks having smaller numbers of registers and ports. A similar method has been used in a media processor [20] and a smart memory structure [21]. In our design, the vector register file is divided into eight banks, where each bank has two read ports and one write port. The vector memory interface (VMI), FPU adder and FPU multiplier share the read/write ports of the register file in a time-multiplexed way. Take the example of eight vector registers, each having sixteen 32-bit elements; the vector register file construction and its connections with other components are shown in Fig. 4. The bandwidth of the vector register file in this configuration can be 6.72 GBytes/sec when operating at 70 MHz.

Besides the structure of the vector register file, we also need to determine its size. Eight vector registers are chosen in our implementation. Although increasing the number of vector registers can reduce the memory bandwidth requirements by allowing more data reuse, most matrix-based applications have little data reuse. Thus, eight vector registers suffice and can demonstrate the effectiveness of our design. Each vector element has 32 bits, which is required for single-precision floating-point calculations. More vector elements in a vector register could amortize the startup time and speedup the overall execution. So, we decided to implement as many elements as allowed by the available resources without increasing the circuit complexity tremendously. We experimented with 8, 16, 32 and 64 elements per vector in our design. In Section 3, the resource usage and system frequency of our vector processor are shown for various numbers of vector elements.

2.3. Vector memory interface (VMI)

VMI controls all the data transfers to/from the data memory banks. It supports scalar loads/stores from/to any for data memory bank, vector loads/stores starting with any data memory bank and for any length, and indexed loads/stores for sparse matrices. The execution times of vector load/store and indexed load/store are not deterministic. The starting point in memory and the length of the data affect their execution time. Also, different data storage patterns in the eight data memory banks may result in different contention patterns for the indexed load/store,

thus resulting in different execution times. A circuit implemented with 16 eight-to-one 32-bit multiplexers can transfer data between any pair of memory-register banks in a single clock cycle.

3. FPGA implementation

Our vector processor resides in one of the two Xilinx XC2V6000-5 chips on the Annapolis Micro Systems Wildstar-II board. We will present here an overview of this platform and the vector processor resource usage. The Wildstar-II board contains two Xilinx Virtex II XC2V6000-5 chips. Each FPGA chip is surrounded by six Samsung 512kx36 DDR SRAMs. The Wildstar board can be mounted on the mother board of a commercial PC through a PCI socket. Software API libraries are provided for communications between the board and the host computer. The registers, on-chip and off-chip memories can be written/read by the host.

For efficient usage of the available logic resources and better performance, dedicated functional units on the FPGA chip are used whenever possible. In our design, the instruction memory and 16 data memory banks are implemented with block memories. Each block memory on the XC2V6000 can hold 512*36 bits of data and each data memory bank in our processor is designed as a 512*32-bit unit to fit in one block. 17 of the 144 block memories are used. One of them is used for the 256*32-bit instruction memory. Deeper data memories are not used because of timing constraints. Increasing the block memory size can increase the complexity of the circuit routing process and can cause system frequency reduction. The eight floating-point multipliers and the 16-bit integer multiplier are implemented with 18-bit*18-bit dedicated multipliers; 33 of the 144 multiplier blocks are used.

There are eight vector registers in our design and various numbers of elements per vector register were investigated: 8, 16, 32 and 64. Table I shows the resource usage for different implementations. 64 is the largest number of elements that we can achieve limited by the slice resources. With the increased circuit complexity and congestion of on-chip routing resources for more elements, the system frequency of the design drops from 70 MHz for 8, 16 and 32 elements to 62.5 MHz for 64 elements. A more substantial reduction should be expected for more elements. The XC2V6000 is equivalent to 6,000,000 system gates and contains 33,792 slices. The major components in a slice are two 4-input LUTs (Lookup Table), two storage elements, and arithmetic logic gates.

4. The W-matrix method

Numerous practical problems in many application areas require the repetitive solution of a set of linear equations given in the form

$$Ax = b \quad (1)$$

where A is a large, sparse and symmetric matrix [13, 14, 16–18]. In some application areas, like power engineering, A is extremely sparse, often containing less than 7% non-zero elements. A conventional way to derive the solution is to factorize matrix A into triangular matrices and then calculate the result by substitutions; these are computation-intensive and essentially sequential processes [6, 7, 14]. Many efforts have been made to apply parallel processing. For example, the W-matrix method that was proposed for power flow analysis [13, 14] uses inverse triangular matrices to get the solution via matrix-vector multiplications. Unlike the inverse of a sparse matrix, which is almost full, the inverses of sparse triangular factors using the W-matrix partitioning method are sparse, though less sparse than the factors themselves. We have for the solution:

$$x = A^{-1}b = (LDU)^{-1}b = U^{-1}D^{-1}L^{-1}b \quad (2)$$

where L , D and U represent the decomposition of A into a lower triangular, diagonal and upper triangular matrix, respectively. With appropriate ordering [13], we can first reduce the factorization fill-ins and factorize A into the form LDL^T . After this ordering, assume that $W = L^{-1}$. Then, (2) can be rewritten as:

$$x = W^T D^{-1} W b. \quad (3)$$

It is obvious that (3) can be solved in three steps:

$$z = W b; y = D^{-1} z; x = W^T y \quad (4)$$

that replace forward and backward substitutions with matrix-vector products. Within each step, all multiplications can be carried out concurrently, which is suitable for parallel programming and vector computing. W-matrix is associated with algorithms that partition the inverses of L and U into elementary matrices with no fill-ins or only user controlled fill-ins. Based on [13] and [14], we can write matrix L as

$$L = L_1 L_2 \cdots L_n \quad (5)$$

where L_i is an identity matrix except that its i -th column is actually the i -th column of matrix L . Then:

$$W = L^{-1} = L_n^{-1} L_{n-1}^{-1} \cdots L_1^{-1} = W_n W_{n-1} \cdots W_1 \quad (6)$$

where W_n is equal to L_n with the sign of its off-diagonal elements reversed. Plugging (6) into (3), we get the expression:

$$x = W_1^T W_2^T \cdots W_n^T D^{-1} W_n \cdots W_2 W_1 b. \quad (7)$$

Table I. Resource usage as a function of the elements per vector register

element size	flip flops	LUTs	slices	system gates
8	13%	23%	34%	1,605,040
16	14%	32%	43%	1,651,709
32	16%	44%	63%	1,874,184
64	21%	75%	99%	2,328,603

To avoid fill-ins induced by (7), we need $2n + 1$ sequential steps of multiplication to get the final solution; it has no advantage over the common substitution method. But according to [13, 14], adjacent matrices W_i , for $1 \leq i \leq n$, can be combined in several ways to form various partitions:

$$x = W_1^T W_2^T \cdots W_p^T D^{-1} W_p \cdots W_2 W_1 b. \quad (8)$$

Now the triangular factors are partitioned into p parts, where we can have $p \ll n$ for a large n . According to (8), the solution x can be obtained after $2p + 1$ steps, where many operations can be executed concurrently in each matrix-vector product step. Different reordering and partitioning schemes based on the factorization path length tree [13, 14] show that the W partitions can be chosen without adding new fill-ins or with adding only user controlled fill-ins in efforts to minimize the number of arithmetic operations. Thus, the combined sparsity of the p factors can be the same as that of L .

5. Our W-matrix implementation

Before mapping the W-matrix method to the vector processor, we introduce the pseudo-column and last partition notions and then modify the linear equation solver accordingly. ‘‘Pseudo-column’’ is an effective way to arrange the storage for the W-matrix partition using long vectors. The ‘‘last partition’’ method combines the last lower triangular matrix with the last upper triangular matrix into a unique one in order to reduce the overall process by one big step.

The performance of the vector processor highly depends on the length of the vectorizable do-loop; the longer the vector, the better the performance. To solve the short vector problem, the concept of pseudo-column was proposed [17]. The recurrence problem, normally affecting the addition part in the linear equation solver, can be eliminated if each pseudo-column contains only matrix elements having different row indices as shown in Fig. 5 [17]. Also, this way columns of a W-matrix partition are combined to achieve greater column density, resulting in better vectorization. Our linear equation algorithm uses the pseudo-column method to store W-matrix data. Thus, in each W-matrix partition the multiplications and additions can be realized with long vectorizable loops.

Since the last partition W_p in (8) is always very dense, actually almost full in our experiments, it becomes advantageous to combine W_p and W_p^T into a unique one [18]. (8) can then be expressed as:

$$\begin{aligned} x &= W_1^T W_2^T \cdots W_p^T D_{p-1}^{-1} D_p^{-1} W_p \cdots W_2 W_1 b \\ &= W_1^T W_2^T \cdots D_{p-1}^{-1} W_p^T D_p^{-1} W_p \cdots W_2 W_1 b \end{aligned} \quad (9)$$

where the diagonal matrix D^{-1} is split into D_{p-1}^{-1} , concerning the previous $p - 1$ partitions, and D_p^{-1} , concerning the last partition. Let $W_p = W_p^T D_p^{-1} W_p$. (9) can then be written as:

$$x = W_1^T W_2^T \cdots W_{p-1}^T D_{p-1}^{-1} W_p W_{p-1} \cdots W_2 W_1 b \quad (10)$$

Table II. Number of non-zero elements (NNZs) after each preprocessing step

Matrix size	49 × 49	118 × 118	443 × 443	1454 × 1454	1723 × 1723
Original NNZs	118 / 4.9%	358 / 2.6%	1180 / 0.6%	3840 / 0.18%	4782 / 0.16%
NNZs after LU	160 / 6.7%	526 / 3.8%	1936 / 1.0%	6878 / 0.33%	8984 / 0.30%
NNZs in W-matrix	265 / 11%	792 / 5.7%	3543 / 1.8%	11434 / 0.54%	14307 / 0.48%

This partitioning reduces the number of serial matrix-vector multiplication steps by combining the forward, diagonal and backward calculations into one piece. Also, no more pseudo-columns are generated because the last row in the last partition is always full and only a few non-zero numbers are induced, therefore performance is improved.

6. Performance results

Real matrices from the power industry were taken as input. we discuss here how to map the W-matrix linear equation solver onto our vector processor and we compare with a Dell PC. Algorithms for approximate minimum degree ordering and LU factorization were applied at static time to the input matrix, then the elimination tree of the matrix was generated and the W-matrix was finally transformed based on the path lengths in the elimination tree [13, 14].

Table II shows the changes in the non-zero elements after each preprocessing step without counting the diagonal elements. It can be seen that after the W-matrix factorization the sparsity of the matrix is still large. After the W-matrix partitions are formed, pseudo-columns and the last block are generated by the host computer, and data is downloaded into the FPGA board for actual computations.

The W-matrix method was run on our vector processor to show that the system can yield high performance for such complex problems. Since quite a lot of preprocessing work is needed before FPGA execution, this method is also suitable for other applications that require iterative calculations using the same input matrices; this is not uncommon in power network problems [6, 7, 13–18]. Other computationally intensive problems, like dense/sparse matrix

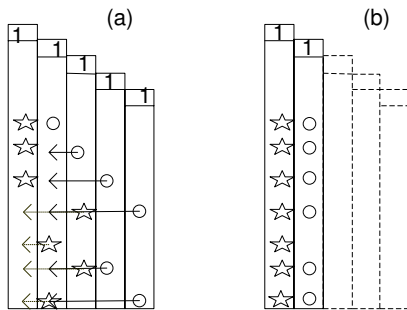


Figure 5. Storage arrangement for W-matrix partitioning: (a) Original columns (b) Pseudo-columns [17]

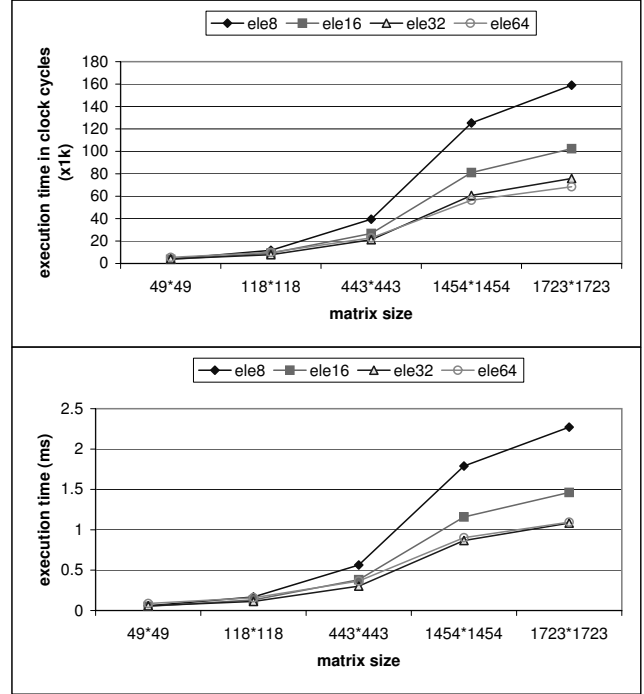


Figure 6. W-matrix execution times for various vector sizes (ele_x: x elements per vector register)

multiplication, are easier to map onto our vector processing system for good performance.

Fig. 6 shows the execution time of the linear equation solver on our vector processor for various element-size implementations. When the matrix size is small, 8 or 16 elements per vector register may consume fewer clock cycles than 32 or 64 elements per vector because the vectorization of the small sparse matrix cannot generate large-sized arrays for the latter cases. With matrix size increases, more elements per vector result in fewer clock cycles. The case of 64 elements per vector is an exception and this can be explained in two ways. First, a high FPGA logic cell utilization (99% slice usage in this case) induces congestion of the on-chip routing resources, thus decreasing the system clock rate (62.5Mhz); second, the size of the test matrices is still not large enough to show the efficiency of the approach. It is not easy to tell whether 32 or 64 is better; 64 elements may yield better performance for a larger or denser matrix. We can only say that for our input matrices the vector processor implementation with 32 elements per vector is a good choice.

Fig. 7 shows a performance comparison with a 1.2 GHz Pentium-III processor. We assumed 32 elements per

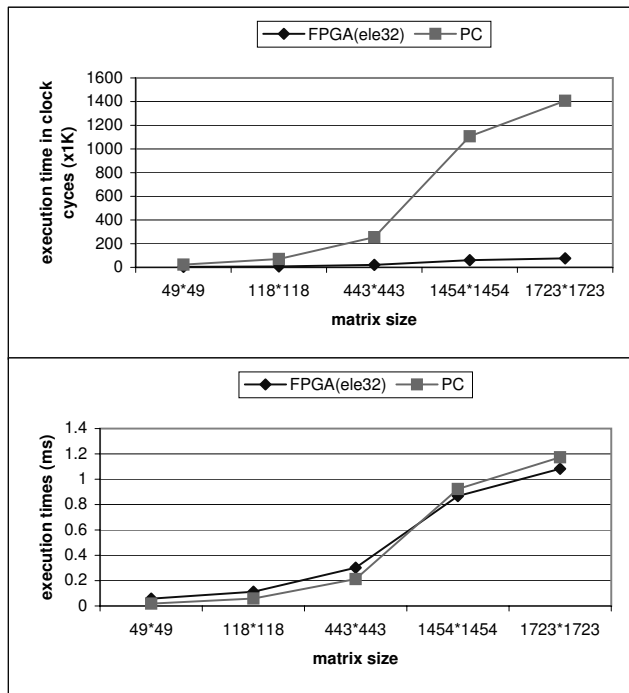


Figure 7. Performance comparisons

vector for the processor implementation in our comparison with the PC. We can observe from Fig. 7 that the clock cycles used on the vector processor are about 1000 times fewer than those on the PC. The performance gain comes from the well designed data storage scheme, the tightly coupled on-chip memory and the abundant floating-point units. But, because of the low frequency (70 MHz) of the FPGA organization, the real speed up is not significant. A more recent FPGA could yield much higher performance. Despite the low frequency, we can still see that our vector processor on the FPGA board can outperform the PC for larger matrices. The results prove that vector processor designs for FPGAs can provide high performance at low cost.

7. Conclusions

New generation, million-gate FPGAs have become increasingly attractive for high performance and cost effective SOC designs. Additionally, the cost of their logic cells has been reduced 30-fold from their introduction, to as little as less than 50 cents for 1000 logic cells. We presented in this paper a vector processor implemented on an FPGA platform. This vector processor has abundant parallel calculation units and supports floating-point calculations. Specialized hardware and respective user instructions for efficient sparse matrix operations were implemented as well. W-matrix, a linear equation solution method that enhances parallelism for sparse matrices, was mapped onto the vector processor. Our comparisons with a commercial PC demonstrate that our implementation is very efficient despite its low frequency. With continued

advances in FPGA technologies, the expected increased speeds and densities of resources could yield much better performance in the near future.

8. References

- [1] B. Radunovic, "An overview of advances in reconfigurable computing systems." *Hawaii Intern. Conf. System Sciences*, 1999.
- [2] M. Gschwind, V. Salapura, and D. Maurer, "FPGA prototyping of a RISC processor core for embedded applications," *IEEE Trans. VLSI Syst.*, pp. 241–250, April 2001.
- [3] C. Chang, J. Wawrzynek, and R. W. Brodersen, "BEE2: A high-end reconfigurable computing system," *IEEE Design Test comp.*, pp. 114–125, 2005.
- [4] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, Eds., *Splash2 : FPGAs in a custom computing machine*. IEEE, 1996.
- [5] J. Wawrzynek, K. Asanovic, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan, "Spert-II: A vector microprocessor system," *IEEE Computer*, pp. 79–86, 1996.
- [6] X. Wang and S. Ziavras, "Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines," *Concur. Comput. Prac. Exper.*, pp. 391–343, April 2004.
- [7] —, "Performance optimization of an FPGA-based configurable multiprocessor for matrix operations," *IEEE Intern. Conf. Field-Prog. Tech.*, Dec. 2003, pp. 303–306.
- [8] X. Xu and S. Ziavras, "A hierarchically-controlled SIMD machine for 2D DCT on FPGAs," *IEEE Intern. Systems-On-Chip Conf.*, Sept 2005, pp. 276–279.
- [9] K. Asanović, "Vector microprocessors," Ph.D., Berkeley, 1998.
- [10] R. Krashinsky, C. Batten, S. Gerding, M. Hampton, B. Pharris, J. Casper, and K. Asanović, "The vector-thread architecture," *31st Intern. Symp. Computer Arch.*, Munich, Germany, June 2004.
- [11] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann, 2002.
- [12] WildstarII Datasheet. <http://www.annapmicro.com>.
- [13] F. L. Alvarado, D. C. Yu, and R. Betancourt, "Partitioned sparse A^{-1} methods," *IEEE Trans. Power Syst.*, pp. 452–459, May 1990.
- [14] M. K. Enns, W. F. Tinney, and F. L. Alvarado, "Sparse matrix inverse factors," *IEEE Trans. Power Syst.*, pp. 466–473, May 1990.
- [15] J. Q. Wu and A. Bose, "A new successive relaxation scheme for the W-matrix solution method on a shared memory parallel computer," *IEEE Trans. Power Syst.*, pp. 233–238, Feb 1996.
- [16] A. Gómez and R. Betancourt, "Implementation of the fast decoupled load flow on a vector computer," *IEEE Trans. Power Syst.*, pp. 977–983, Aug 1990.
- [17] G. P. Granelli, M. Montagna, G. L. Pasini, and P. Marannino, "A W-matrix based fast decoupled load flow for contingency studies on vector computers," *IEEE Trans. Power Syst.*, pp. 946–953, Aug 1993.
- [18] A. Padilha and A. Morelato, "A W-matrix methodology for solving sparse network equations on multiprocessor computers," *IEEE Trans. Power Syst.*, pp. 1023–1030, Aug 1992.
- [19] The Quixilica FP datasheet. <http://www.qinetiq.com>.
- [20] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens, "A bandwidth-efficient architecture for media processing," *Intern. Symp. Microarch.*, 1998, pp. 3–13.
- [21] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart memories: A modular reconfigurable architecture," *27th Annual Intern. Symp. Computer Arch.*, June 2000, pp. 161–171.