Reconfiguration support for vector operations

Hongyan Yang, Sotirios G. Ziavras* and Jie Hu

Department of Electrical and Computer Engineering,
New Jersey Institute of Technology,
Newark, New Jersey 07102, USA
E-mail: hy34@njit.edu
E-mail: ziavras@njit.edu
E-mail: jhu@njit.edu
*Corresponding author

Abstract: A programmable vector processor and its implementation on a Field-Programmable Gate Array (FPGA) board 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 runs at 70 MHz on the Xilinx XC2V6000-5 chip. FPGA resource utilisation information is included in this paper. To test the performance, the W-matrix sparse solver for linear equations is realised on this platform. 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 1.1 ms on the FPGA. A comparison with a commercial PC indicates that the vector processor is very competitive for such computation-intensive problems.

Keywords: system-on-a-chip; SOC; field programmable gate array; FPGA; vector processor; W-matrix method; matrix sparsity; linear equation solver.

Reference to this paper should be made as follows: Yang, H., Ziavras, S.G. and Hu, J. (2007) 'Reconfiguration support for vector operations', *Int. J. High Performance Systems Architecture*, Vol. 1, No. 2, pp.89–97.

Biographical notes: Hongyan Yang is a PhD candidate in the Department of ECE at New Jersey Institute of Technology (NJIT). She received an MS in Electrical Engineering from Beijing University of Posts and Telecommunications in 2001.

Sotirios G. Ziavras is a Professor in the Department of ECE at NJIT. He received a PhD in Computer Science from George Washington University in 1990. His research interests are reconfigurable computing, parallel processing and computer architecture.

Jie Hu is an Assistant Professor in the Department of ECE at NJIT. He received a PhD in Computer Science and Engineering from the Pennsylvania State University in 2004. His research interest is computer architecture.

1 Introduction

Computation-intensive problems are a great challenge to general-purpose processors due to the latters' 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 productions and drives designers in the search of flexible solutions. On the other hand, in the last decade, there have been significant Field Programmable Gate Array (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 (Buell et al., 1996; Chang et al., 2005; Gschwind et al., 2001; Radunovic, 1999; Wang and Ziavras, 2003, 2004; Wawrzynek et al., 1996; Xu and Ziavras, 2005). New generation FPGAs with million gates have also made feasible powerful System-On-a-Chip (SOC) designs.

To ease programming with FPGAs and decrease the overall design time, several types of Intellectual Property (IP) cores for various application areas are available by third-party companies or FPGA producers. General-purpose programmable IP processors are also available; relevant representative products are the soft-core NIOS of Altera and the MicroBlaze and PicoBlaze processors of Xilinx. Hardwired processors, as well, are sometimes available on FPGA chips, such as the PowerPC on some Xilinx products. These processors have brought about great flexibility in design and can be used to implement successfully on-chip Single-Instruction Multiple-Data (SIMD)/Multiple-Instruction Multiple-Data (MIMD) parallel computing engines (Wang and Ziavras, 2003, 2004; Xu and Ziavras, 2005). Such processor cores have adopted advanced architectural techniques like pipelining, caching and branch prediction.

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 (Asanović, 1998; Krashinsky et al., 2004; Wawrzynek et al., 1996). For data parallel applications, a vector processor can easily outperform Very Large Instruction Width (VLIW) and superscalar processors at low cost (Patterson and Hennessy, 2002). An FPGA-based implementation of a vector processor would be a promising task. However, a limited memory bandwidth may become a major bottleneck in vector processors. Fortunately, recent FPGA-based systems, such as Annapolis Micro Systems Wildstar boards (Annapmicro web site) and Starbridge Hypercomputers (Starbridge systems web site), can provide high memory bandwidth at the gigabyte range by having 4–6 high speed SRAMs connected to one FPGA chip. This makes the vector processor design on these systems an attractive approach.

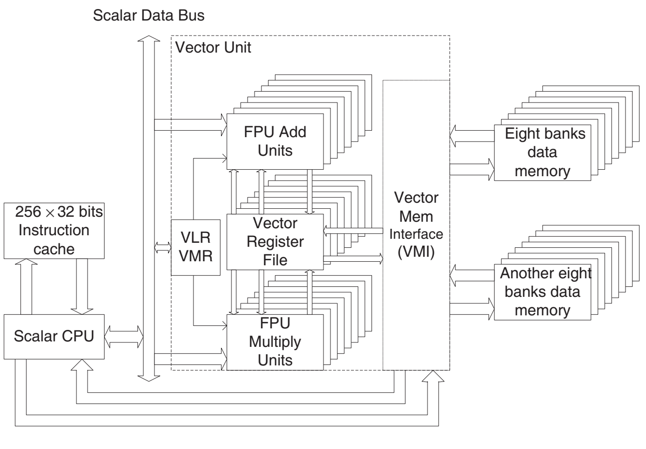
In this paper, we present a programmable vector processor implemented on the Annapolis Wildstar-II board. A vector register file having multiple ports is located in the centre of our vector processor. By dividing it into several banks, a higher bandwidth can be provided in a much smaller area. The vector 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 organised 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.

Matrix operations are widely used in various areas such as power flow analysis, image processing and human-machine interaction. In this paper, a matrix-oriented linear equation solver for power flow analysis is employed to show the efficiency of our vector processor. The repetitive solution of linear equations is sometimes the most time consuming part in an application. The traditional solution method involves forward and backward substitutions (Alvarado et al., 1990; Wang and Ziavras, 2004). These steps are essentially sequential and prohibit intense parallel computing approaches for high-performance. The W-matrix method has been proposed as an efficient way to solve linear equations by changing sequential substitutions into matrix multiplications which can run in parallel (Alvarado et al., 1990; Enns et al., 1990). Some successful W-matrix solvers have run on shared-memory parallel computers (Wu and Bose, 1996), vector supercomputers (Gómez and Betancourt, 1990; Granelli et al., 1993) and multiprocessors (Padilha and Morelato, 1992). We show that the W-matrix method works efficiently on our FPGA-based vector processor. A pseudo-column technique is used to generate longer vector arrays in the application. Real power network matrices are used to test our approach and the results are compared with those of a commercial PC. Our design permits general-purpose programmability and can be applied to various other application areas as well. Such applications having similar types of data parallelism could benefit from reduced costs and smaller execution times on our vector processor.

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 Figure 1. It is organised 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. In order to focus on the actual vector design, the floating-point IP cores were purchased from Quixilica (Tekmicro web site).

Figure 1 Block diagram of the vector processor

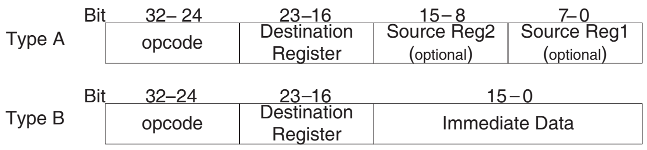


Note: VLR: Vector Length Register; VMR: Vector Mask Register.

2.1 *Instruction set architecture*

Our vector processor has a RISC architecture supporting 24 instructions. There are 16 scalar instructions in the areas of data transfer, arithmetic operations and programme control. The other eight instructions run in the vector mode for data transfers and arithmetic operations. The latter instructions are of Types A or B, as shown in Figure 2. Type A instructions use up to two source registers and one destination register. Type B instructions use one destination register and a 16-bit immediate operand. Although we do not need eight bits to represent the opcode or registers, it is a good choice for a possible extension of the instruction set or register populations in the future. Two addressing modes are used in our design: direct and register indexed.

Figure 2 Instruction formats

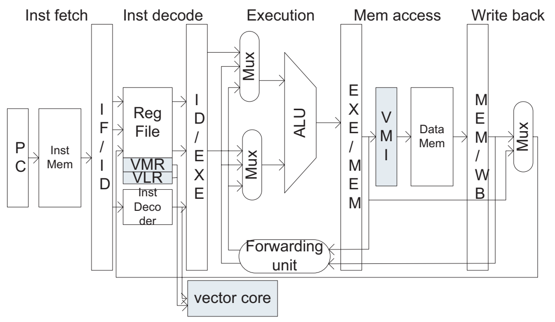


We have designed our own assembler to translate programs written in assembly language into machine code targeting our system. The dramatically reduced code size resulting from vector processing makes our programming job easier. Creating our own compiler for high-level language support is not an easy task and is beyond the scope of our research.

2.2 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 Figure 3. 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, Figure 3 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 ALU is able to deal with 16-bit integer addition/subtraction and multiplication. The register file includes 30 general-purpose registers and two special-purpose registers for vector processing. It supports two read ports and one write port.

Figure 3 Scalar processor architecture



The two specialised 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 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. We must emphasise that 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 performance. This greatly eases code writing for our processor.

2.3 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 the interconnect between the Vector Floating-Point Units (VFUs) (Asanović, 1998). A straightforward way to implement the vector register file is to use a single multiported memory. But this is a very expensive solution requiring many logic resources that increase the power consumption of the FPGA chip. Take the example of eight vector registers each having 32 32-bit elements; the left diagram in Figure 5 shows the slice usage for a Xilinx XC2V6000 chip and the right one shows the power consumption assuming that it runs at 70 MHz. 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 file into banks having smaller numbers of registers and ports. A similar method has been used in a media processor (Rixner et al., 1998) and a smart memory structure (Mai et al., 2000). 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 16 32-bit elements; the vector register file construction and its connections with other components are shown in Figure 4. The bandwidth of the vector register file in this configuration can be 6.72 GBytes/sec when operating at 70 MHz. If the equivalent bandwidth is to be provided by a single register bank, a register file with 16 read ports and 8 write ports would be required. This is significantly less efficient in terms of area, speed and power consumption than the bank-based architecture since the latter only consumes 24% of the slices and 4.3 W of power while the resource and power consumptions increase dramatically with an increase in the number of ports, as shown in Figure 5. The additional cost of the bank structure corresponds to a circuit for data transfers between any pair of memory-register banks. This circuit uses quite a few FPGA resources and lies in the critical path; but comparing to the single register file implementation, this resource usage is much smaller and does not change the fact that the bank structure is much more efficient than the single block implementation.

Figure 4 Vector register file organisation

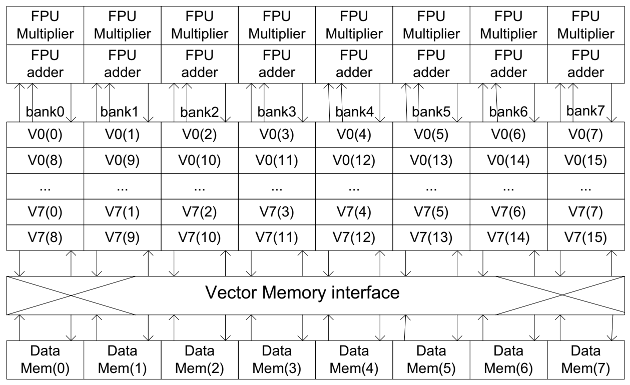
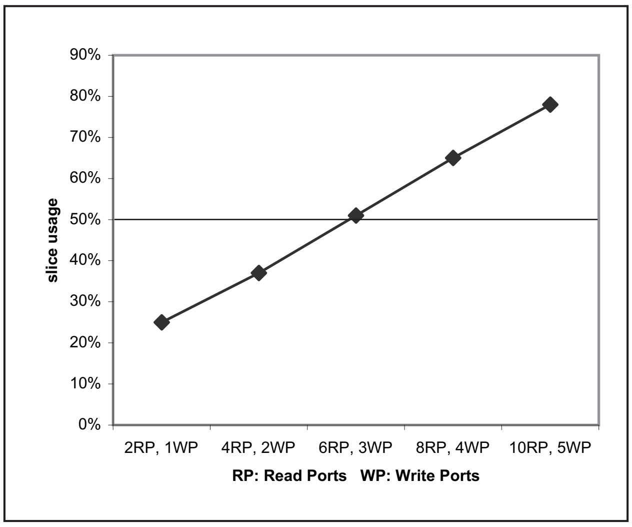
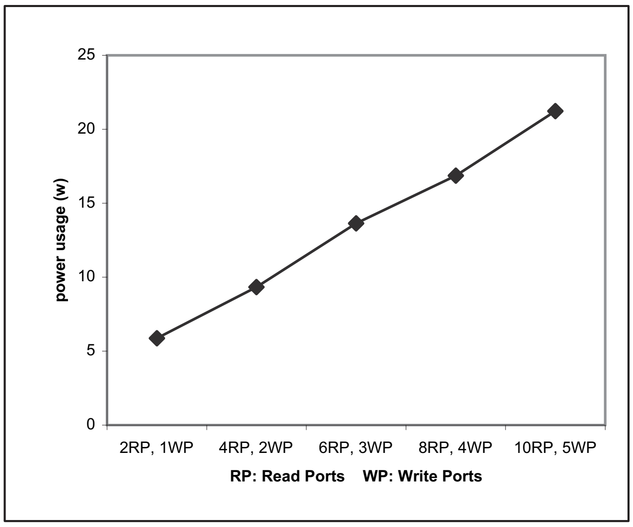

FPU Multiplier
FPU adder
bank0
bank1
bank2
bank3
bank4
bank5
bank6
bank7
V0(0) V0(1) V0(2) V0(3) V0(4) V0(5) V0(6) V0(7)
V0(8) V0(9) V0(10) V0(11) V0(12) V0(13) V0(14) V0(15)
...
V7(0) V7(1) V7(2) V7(3) V7(4) V7(5) V7(6) V7(7)
V7(8) V7(9) V7(10) V7(11) V7(12) V7(13) V7(14) V7(15)
Vector Memory interface
Data Mem(0) Data Mem(1) Data Mem(2) Data Mem(3) Data Mem(4) Data Mem(5) Data Mem(6) Data Mem(7)

Figure 5 Resource and power consumption for single-block implementation of a vector register file containing eight vector registers of 32 32-bit elements





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 amortise the startup time and speed up the overall execution; the time to fill up the pipeline is eight clock cycles for floating-point multiplication and eleven clock cycles for floating-point addition. 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.

2.4 *Vector memory interface*

VMI controls all the data transfers to/from the data memory banks. It supports scalar loads/stores from/to any 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 time of vector load/store and indexed load/store is not deterministic. The starting point in memory and the length of the data affect the execution time of these operations. Besides the impact of the vector length, 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 is used to transfer data between any pair of memory-register banks in a single clock cycle.

The vector memory interface uses quite a few FPGA resources. For example, assume the configuration of eight vector registers each having 32 32-bit elements; the vector memory interface consumes 5% of the flip flops, 21% of the LUTs and 24% of the slices in the XC2V6000 chip, which is equivalent to 143,081 system gates out of the 6 M available.

2.5 *Data storage structures*

Three levels of data storage are used in this implementation: separate register files in the scalar and vector units; an instruction memory and two sets of eight data memory banks using on-chip RAMs; finally, the host computer which is temporarily used as a substitute for off-chip SRAMs.

The register file in the scalar unit can hold 32 words and the register fie in the vector unit can hold up to 512 words for the 64-element configuration. The instruction memory has 256×32 bits and each data memory bank has 512×32 bits. There are two sets of eight data memory banks each, so the on-chip data memory can have up to 32 Kbytes. Besides our consideration to match the eight vector register banks, it is always better to divide deep memories into several smaller parts for better performance. Actually, if we do not divide the 32 Kbytes into 16 banks, this design cannot satisfy the time requirements for communications between the host and the FPGA board.

3 FPGA implementation

Our vector processor resides in one of the two Xilinx XC2V6000-5 chips on the Annapolis Micro Systems Wildstar-II board. In this section, we will present an overview of this FPGA platform, our design flow, an overview of the FPGA chip structure and the resource usage for our implementation.

3.1 *Overview of the Wildstar-II board*

The Wildstar-II board contains two Xilinx Virtex II XC2V6000-5 chips. Each chip is surrounded by six Samsung 512kx36 DDR SRAMs; each SRAM has a 36-pin connection to the FPGA chip. The Wildstar board can be mounted on the mother board of a commercial PC through a PCI socket. Software API libraries on the host computer and interfaces to the PCI bus on the FPGA side are provided by Annapolis for communications via the PCI bus between the board and the host computer. The registers, on-chip memory and off-chip memory can be written/read by the host computer.

3.2 Design flow

For our vector processor, we employed an HDL-based design flow: all the processor components were first described in a subset of synthesisation VHDL, then Modelsim was used for gate-level simulation to check for correctness and Synplify pro was then chosen for synthesis; at last, Xilinx place-and-route tools were used to put the design on the FPGA chip. Other development tools, such as Viva of Starbridge Systems (Starbridge Systems web site) and Corefire of Annapolis Micro Systems (Annapmicro web site), are also available in our laboratory. These tools generally provide user friendly graphic interfaces and incorporate a bunch of high-performance IP cores. The user can just use a 'drag and drop approach'. They aim to be a substitute for hardware-based methodologies like VHDL and Verilog by providing a much quicker and flexible approach. Their benefit is short design time and development that does not descend into low-level hardware details. But, on the other hand, they limit the designer's control and, therefore, the hardware implementation can be hardly fully optimised. Our experiments show that a processor written in VHDL and running at 90 MHz on the Annapolis Wildstar-II board can only run at 60 MHz when designed by Corefire.

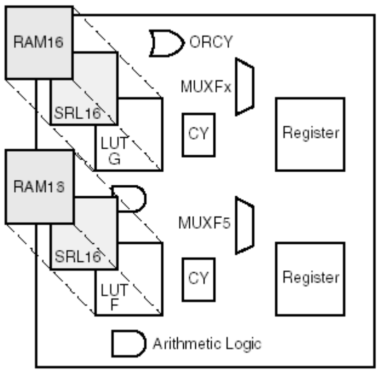
3.3 Resource usage

Advanced FPGA organisations are characterised by large numbers of programmable logic cells, abundant dedicated functional units and high system speed. The distinct complexity and structure of the logic resources is often used to distinguish among FPGA families (Gschwind et al., 2001). In the following, we focus on the Xilinx Virtex-II series of FPGA chips that serve as the platform for our vector processor. More recent Xilinx FPGAs, like the Virtex-II pro and Virtex-4, improve chip programmability by incorporating hardcore processors, third-party IPs and DSP slices. However, these types of resources are not appropriate for our vector design.

There are two kinds of programmable resources in FPGAs: logic and routing. The basic logic element in a Xilinx FPGA is the Configurable Logic Block (CLB); each block is tied to a switch matrix for routing. In the Xilinx Virtex-II series, each CLB includes four slices and each slice contains two four-input function generators, two carry-generating logic elements, arithmetic logic gates, wide function multiplexors and two storage elements, as shown in Figure 6. The Xilinx XC2V6000 chip contains 33,792 slices, 144 18-bit $\times$ 18-bit multiplier blocks, 144 18 kbit block memories as dedicated functional units and up to 1104 I/O pins. It has a total of six million system gates. A system gate is equivalent to an NAND gate in ASIC technology. The capacity of this chip is large enough for our medium-complexity processor design.

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 Xilinx XC2V6000 chip can hold 512 $\times$ 36 bits of data and each data memory bank in our processor is designed as a 512 $\times$ 32-bit storage unit to fit in one block. About 17 out of the 144 block memories are used. One of them is used for the 256 $\times$ 32-bit instruction memory. Deeper data memories are not used because of our 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$\times$18-bit dedicated multipliers; 33 out of the 144 multiplier blocks are used. The vector register banks are implemented with distributed RAMs configured by function generators, as shown in Figure 6.

Figure 6 Slice structure of the Xilinx Virtex-II FPGA series (Xilinx web site)



There are eight vector registers in our design and various numbers of elements per vector register were investigated: 8, 16, 32 and 64. Table 1 shows the resource usage of these different implementations for the XC2V6000 chip. It can be observed that 64 is the largest number of element that we can achieve limited by the slice resources. With the increased circuit complexity and congestion of the 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 frequency reduction should be expected for more elements.

Table 1 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

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

$$Ax = b \tag{1}$$

where A is a large, sparse and symmetric matrix (Alvarado et al., 1990; Enns et al., 1990; Gómez and Betancourt, 1990; Granelli et al., 1993; Padilha and Morelato, 1992). In some application areas, like power engineering, A is extremely sparse, often containing less than 7% of non-zero elements. A conventional way to derive the solution is to factorise matrix A into triangular matrices and then calculate the result by substitutions; these are computation-intensive and essentially sequential processes (Enns et al., 1990; Wang and Ziavras, 2003, 2004). Many efforts have been made to apply parallel processing. For example, the W-matrix method that was proposed for power flow analysis (Alvarado et al., 1990; Enns et al., 1990) 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:

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

where L, D and U represent the decomposition of A into a lower triangular, diagonal and upper triangular matrix, respectively. With appropriate ordering (Alvarado et al., 1990), we can first reduce the factorisation fill-ins and factorise A into the form LDL^{T}. After this ordering, assume that $W = L^{-1}$. Then, (2) can be rewritten as:

$$x = W^{\mathrm{T}}D^{-1}Wb \tag{3}$$

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

$$z = Wb;\ y = D^{-1}z; \quad x = W^{\mathrm{T}}y \tag{4}$$

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 Alvarado et al. (1990) and Enns et al. (1990), we can write matrix L as

$$L = L_1L_2\cdots L_n \tag{5}$$

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

$$W = L^{-1} = L_n^{-1}L_{n-1}^{-1}\cdots L_1^{-1} = W_nW_{n-1}\cdots W_1 \tag{6}$$

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:

$$x = W_1^{\mathrm{T}}W_2^{\mathrm{T}}\cdots W_n^{\mathrm{T}}D^{-1}W_n\cdots W_2W_1b \tag{7}$$

To avoid fill-ins induced in (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 Alvarado et al. (1990) and Enns et al. (1990), adjacent matrices W_i, for $1 \leqslant i \leqslant n$, can be combined in several ways to form various partitions:

$$x = W_1^{\mathrm{T}}W_2^{\mathrm{T}}\cdots W_p^{\mathrm{T}}D^{-1}W_p\cdots W_2W_1b \tag{8}$$

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 factorisation path length tree (Alvarado et al., 1990; Enns et al., 1990) show that the W partitions can be chosen without adding new fill-ins or with adding only user controlled fill-ins in efforts to minimise the number of arithmetic operations. Thus, the combined sparsity of the p factors can be the same as that of L.

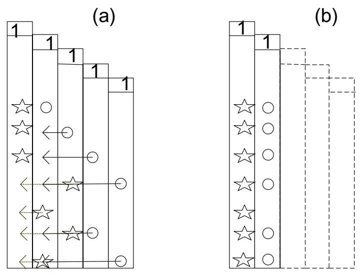
5 W-matrix implementation on the vector processor

Before mapping the W-matrix method to the vector processor, we introduce the pseudo-column and last partition notions in Sections 5.1 and 5.2, respectively 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.

5.1 Pseudo-column

Efforts to implement the W-matrix method on vector supercomputers began in 1990 (Gómez and Betancourt, 1990). However, they did not yield good performance because the sparsity in the matrices and the recurrence problem force the linear algebra solution to use short vectors. The performance of the vector processor highly depends on the length of the vectorisable do-loop; the longer the vector, the better the performance. To solve the above short vector problem, the concept of pseudo-column was proposed (Granelli et al., 1993). 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 Figure 7 (Granelli et al., 1993). Also, this way columns of a W-matrix partition are combined to achieve greater column density, resulting in better vectorisation. The linear equation algorithm in this paper uses the pseudo-column method to store W-matrix data. Thus, in each W-matrix partition the multiplication and addition operations can be realised with long vectorisable loops.

Figure 7 Storage arrangement for W-matrix partitioning: (a) original columns and (b) pseudo-columns



Source: Granelli et al. (1993).

5.2 *Last partition*

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 (Padilha and Morelato, 1992) (8) can then be expressed as:

$$\begin{aligned} x &= W_1^{\mathrm{T}} W_2^{\mathrm{T}} \cdots W_p^{\mathrm{T}} D_{p-1}^{-1} D_{lp}^{-1} W_p \cdots W_2 W_1 b \\ &= W_1^{\mathrm{T}} W_2^{\mathrm{T}} \cdots D_{p-1}^{-1} W_p^{\mathrm{T}} D_{lp}^{-1} W_p \cdots W_2 W_1 b \end{aligned} \tag{9}$$

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

$$x = W_1^{\mathrm{T}} W_2^{\mathrm{T}} \cdots W_{p-1}^{\mathrm{T}} D_{p-1}^{-1} W_{lp} W_{p-1} \cdots W_2 W_1 b \tag{10}$$

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 will be generated because the last row in the last partition is always full and only a few non-zero numbers will be induced, therefore performance can be improved.

6 Performance results

W-matrix was used to test the performance of our vector processor. Real matrices from the power flow area were taken as input. Additional computation-intensive applications as well, like dense/sparse matrix-matrix/matrix-vector multiplication, can run on the vector processor and yield good performance. In the following subsections, we discuss how to map the W-matrix linear equation solver onto our vector processor and a comparison with a Dell PC is performed as well.

6.1 *Mapping the W-matrix method onto the vector processor*

At static time algorithms for approximate minimum degree ordering and LU factorisation were applied 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 (Alvarado et al., 1990; Enns et al., 1990). Finding out how to partition the inverse triangular factors in order to get the shortest solution time on the vector processor is difficult since there are numerous ways to form W-matrix partitions. The partitioning method used in this paper is easy to implement and can also guarantee good performance. For the sake of brevity, we do not present the detailed partitioning scheme here; we prefer to focus on the vector-based operations. Table 2 shows the changes in the non-zero elements after each preprocessing step without counting the diagonal elements. It can be seen that, after W-matrix factorisation 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. This experiment is suitable for those applications that require repetitive linear equation solutions without any change in the input matrices.

We apply these steps:

1 the application program is written in assembly language and our assembler is used to translate it into machine code targeting the vector processor

2 the data storage structure is prepared

3 the instructions and initial data are written into the on-chip instruction and data memories by the host computer and calculations on the board then begin

4 the vector processor sends a signal to the host computer for more input data if all the data in the on-chip memories have been used and calculation is stalled

5 the host fills up the on-chip data memories with remaining data and forces the vector processor to continue execution

6 steps (2) and (3) are repeated until the calculation is finished

7 the final result can be read from the registers or memories.

A PCI-based transfer can take unacceptably long time due to the processing time for interrupts targeting the host computer. In future work, we could use off-chip memories to store data and do data transfers from off-chip memories to on-chip memories without stalling the processor. Since each data item is used exactly once and matrix operations on the FPGA take a long time, data can be prefetched in an effort to overlap communications with computations. In this paper, we emphasise the vector processor design and implementation, so the memory prefetch part has not been implemented. Thus, the performance results given in this paper do not include the stalling time due to data transfers on the PCI bus.

6.2 *Performance analysis*

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 only suitable for those applications that require iterative calculations using the same input matrices; this is not uncommon in power network problems (Alvarado et al., 1990; Enns et al., 1990; Gómez and Betancourt, 1990; Granelli et al., 1993; Padilha

Table 2 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%	11,434 / 0.54%	14,307 / 0.48%

and Morelato, 1992; Wang and Ziavras, 2003, 2004; Wu and Bose, 1996). Other computationally intensive problems, like dense/sparse matrix multiplication, are easier to map onto our vector processing system for good performance.

Figure 8 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 vectorisation of the small sparse matrix cannot generate large-sized arrays for the latter cases. With the 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. Firstly, a high FPGA logic cell utilisation (99% slice usage in this case) induces congestion of the on-chip routing resources, thus decreasing the system clock rate (62.5 MHz); secondly, 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.

Figure 8 *W*-matrix execution times for various vector sizes (elex: x elements per vector register)

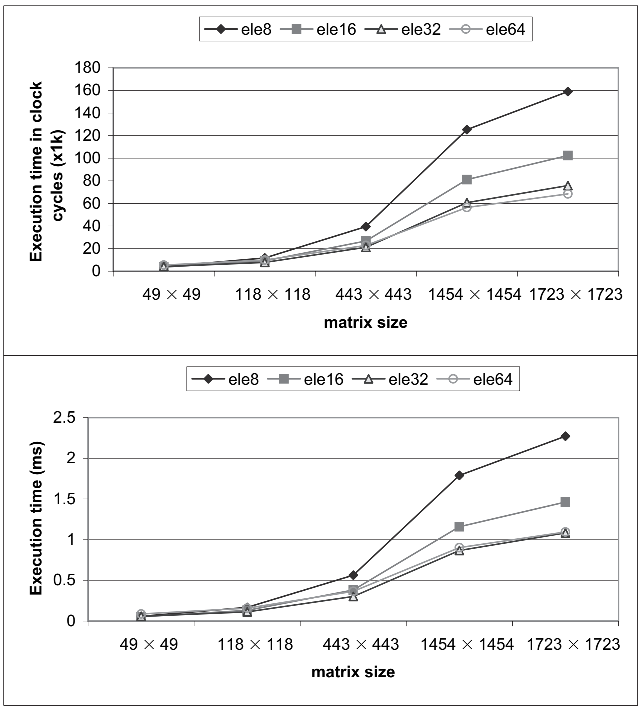
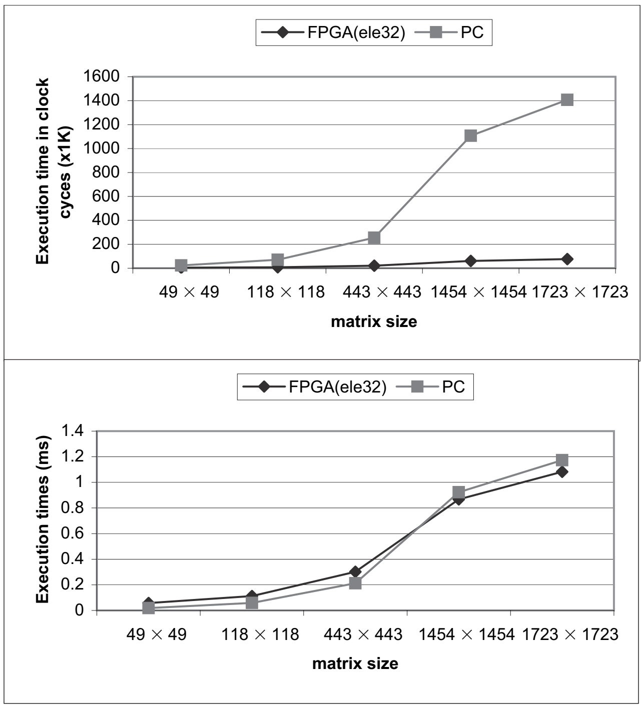


Figure 9 shows a performance comparison with a 1.2 GHz Pentium-III processor. We assumed 32 elements per vector for the processor implementation in our comparison with the PC. We can observe from Figure 9 that the clock cycles used on the vector processor are about 1000 times fewer than those on the 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 organisation, the real speedup 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 SOC designs on FPGA boards can provide high-performance vector-processing systems at low cost.

Figure 9 Performance comparison of our vector processor and a PC



7 Conclusions

New generation, million-gate FPGAs have become increasingly attractive for high performance and cost effective SOC designs. Additionally, FPGA providers continue to decrease their price. The cost of 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. Specialised 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 comparison with a commercial PC demonstrates that our implementation is very efficient despite its low frequency. With continued advances in FPGA technologies, the expected increased speeds and densities could yield much better performance in the near future for such computationally intensive problems on FPGA-based vector implementations.

References

Available at: http://www.starbridgesystems.com.

Available at: http://www.xilinx.com.

Available at: http://www.tekmicro.com. The Quixilica FP datasheet.

Available at: http://www.annapmicro.com. The WildstarII datasheet.

Alvarado, F.L., Yu, D.C. and Betancourt, R. (1990) 'Partitioned sparse A^{-1} methods', *IEEE Transactions on Power System*, Vol. 5, No. 2, pp.452–459.

Asanović, K. (1998) 'Vector microprocessors', PhD thesis, University of California, Berkeley.

Buell, D.A., Arnold, J.M. and Kleinfelder, W.J. (Eds). (1996) *Splash2: FPGAs in a Custom Computing Machine*, CA: Los Alamitos, *IEEE Computer Society Press*.

Chang, C., Wawrzynek, J. and Brodersen, R.W. (2005) 'BEE2: a high-end reconfigurable computing system', *IEEE Design and Test of computers*, Vol. 22, No. 2, pp.114–125.

Enns, M.K., Tinney, W.F. and Alvarado, F.L. (1990) 'Sparse matrix inverse factors', *IEEE Transactions on Power System*, Vol. 5, No. 2, pp.466–473.

Gómez, A. and Betancourt, R. (1990) 'Implementation of the fast decoupled load flow on a vector computer', *IEEE Transactions Power System*, Vol. 5, No. 3, pp.977–983.

Granelli, G.P., Montagna, M., Pasini, G.L. and Marannino, P. (1993) 'A W-matrix based fast decoupled load flow for contingency studies on vector computers', *IEEE Transaction Power System*, Vol. 8, No. 3, pp.946–953.

Gschwind, M., Salapura, V. and Maurer, D. (2001) 'FPGA prototyping of a RISC processor core for embedded applications', *IEEE Transactions on VLSI System*, Vol. 9, No. 2, pp.241–250.

Krashinsky, R., Batten, C., Gerding, S., Hampton, M., Pharris, B., Casper, J. and Asanović, K. (2004) 'The vector-thread architecture', *31st International Symposium on Computer Architecture*, Munich, Germany.

Mai, K., Paaske, T., Jayasena, N., Ho, R., Dally, W.J. and Horowitz, M. (2000) 'Smart memories: a modular reconfigurable architecture', *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp.161–171.

Padilha, A. and Morelato, A. (1992) 'A W-matrix methodology for solving sparse network equations on multiprocessor computers', *IEEE Transactions Power System*, Vol. 7, No. 3, pp.1023–1030.

Patterson, D.A. and Hennessy, J.L. (2002) *Computer Architecture: A Quantitative Approach*, (3rd edition,) San Mateo: Morgan Kaufmann.

Radunovic, B. (1999) 'An overview of advances in reconfigurable computing systems', *Hawaii International Conference on System Sciences*.

Rixner, S., Dally, W.J., Kapasi, U.J., Khailany, B., Lopez-Lagunas, A., Mattson, P.R. and Owens, J.D. (1998) 'A bandwidth-efficient architecture for media processing', *International Symposium on Microarchitecture*, pp.3–13.

Wang, X. and Ziavras, S. (2003) 'Performance optimization of an FPGA-based configurable multiprocessor for matrix operations', *IEEE International Conference on Field-Programmable Technology*, pp.303–306.

Wang, X. and Ziavras, S. (2004) 'Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines', *Concurrency and Computation: Practice and Experience*, Vol. 16, No. 4, pp.391–343.

Wawrzynek, J., Asanovic, K., Kingsbury, B., Johnson, D., Beck, J. and Morgan, N. (1996) 'Spert-II: A vector microprocessor system', *IEEE Computer*, Vol. 29, No. 3, pp.79–86.

Wu, J.Q. and Bose, A. (1996) 'A new successive relaxation scheme for the W-matrix solution method on a shared memory parallel computer', *IEEE Transactions on Power System*, Vol. 11, No. 1, pp.233–238.

Xu, X. and Ziavras, S. (2005) 'A hierarchically-controlled SIMD machine for 2D DCT on FPGAs', *IEEE International Systems-On-Chip Conference*.