An FPGA Based Vector Computer for Sparse Matrix-Vector Processing

MUHAMMAD ZAFRUL HASAN and SOTIRIOS G. ZIAVRAS
Department of Electrical and Computer Engineering
New Jersey Institute of Technology
Newark, NJ 07102
U.S.A.

Abstract: - Vector computers are suitable for processing vectors and matrices. Nevertheless, sophisticated algorithms are needed to process sparse vectors and matrices. Additionally, vector computers are not always the best computing platform because of their very high cost. We have designed a special architecture of a vector computer and implemented it within an FPGA. Our main objective is to build multiprocessors and vector machines within FPGAs because they offer a low cost alternative to supercomputers and other parallel machines. We demonstrate here that FPGAs provide a cost effective and reconfigurable means to implement vector computers. The implementation and performance of the above vector computer on an FPGA platform are also discussed.

Key-Words: - Vector, Matrix, Sparse, FPGA, Scalar, Memory Interleaving, Pipelining

1 Introduction

Multiple issue, pipelined processors are particularly suitable for large scientific and engineering applications. Issuing multiple instructions in a clock cycle and pipelining a processor deeply could greatly enhance its performance. However, the problem of scheduling operations has the same complexity for both architectures. Vector processors are equipped with high-level operations on linear arrays of numbers - vectors. A vector instruction is equivalent to a loop implementation, where an iteration of the loop calculates one of the elements in the result. Vector instructions perform well for many vector problems associated with multiple-issue, deeply pipelined processors. A single vector instruction represents a lot of accumulated tasks. Thus the instruction bandwidth requirement is reduced. In contrast, the data bandwidth requirement increases substantially. Vector load and store instructions normally have a known access pattern. A single access is required for the complete vector rather than for a single element. As a result, memory-latency is encountered once for the entire vector. Also, control hazards arising from loop branches are avoided as the vector instruction (which has predetermined behavior) replaces an entire loop. Accessing the memory with a predetermined index or stride (for non sequential accesses) makes a vector processor suitable for matrix handling.

Many problems in engineering and science, such as power systems and simulation of complex systems, need the handling of matrices and vectors that are sparse. Vector processors have been suggested [1] and have been used [2, 3] to handle sparse matrices and vectors with significant speedup compared to scalar processors. Many algorithms have been proposed [4, 5] that further enhance the efficiency of the solutions to the above problems on vector machines. An architecture is proposed here to efficiently handle sparse matrices and vectors with FPGAs. Conventional vector processors are expensive. FPGAs provide an alternative platform for cost effective implementation of vector processors. The ease of reconfiguring FPGA-based platforms is also attractive. We have implemented our design on an FPGA and tested it successfully.

Let us consider the solution to a set of linear equations of the form $A.x = b$, where A is a symmetric, nonsingular matrix (often sparse in many applications), x and b are vectors. Matrix A can first be factored as $A = L.D.U$ for finding the solution based on an LU-decomposition algorithm, where L is the lower triangular matrix, U is the upper triangular matrix, and D is a diagonal matrix. The computation of L and U is preceded by ordering the rows and columns of A to minimize the number of new nonzero terms compared to A. The solution to 'x' involves a series of matrix-vector operations, which could be handled efficiently by a vector processor for the reasons explained above. Our ultimate objective in this project is to design and implement a vector machine in an FPGA to support high performance solution of linear equations.

If we consider the multiplication of two matrices (stored in memory), one row of the first one and one column of the second one need to be loaded for each element of the result. This could be done using two vector instructions with the respective starting addresses of each of the matrices. Matrices are stored in either row-order or column-order. In either case, vector load operation would need access to memory locations that are a fixed distance apart (the distance being equal to the size of the row or column). This operation could be performed using a single vector instruction with a stride for accessing elements.
Moreover, if the matrices are sparse and the location of nonzero elements is known, then a vector of indices could be formed with the offsets of nonzero elements. This index vector can be used with the starting address of the matrix to access those nonzero elements. This could also be achieved using a single vector instruction with an index vector.

As an illustration, we can consider a result vector (V3) produced from two input vectors (V1, V2) using the equation $V3 = (a \times V1) + V2$, where 'a' is a scalar. Using a scalar processor, it would require 2n loads, where n is the size of the vector. It also requires n multiplications and n additions. The number of instructions to represent the above would be approximately eight, considering only one control instruction per loop (there are four loops). Using a vector processor, there would be two loads (each with n elements), one multiplication and one addition. The number of instructions to represent the above would be four. This implies almost an n-fold reduction in instruction memory access latency. Also, the number of instruction fetches is half of that in the scalar operation.
As the above operations are repeatedly encountered in handling matrices, the vector processor promises much better performance.

A similar work with sparse matrix-vector operations has been reported in [7]. However, it deals with the FPGA implementation of the multiplication algorithm for the sparse matrix-vector product. This paper did not encompass the design of a complete vector processor. Work has been reported in [8] to handle this problem using multiple processors. However, the communication overhead among the processors and resources required to build such a machine can not be ignored. Also, the approach taken to build the multiprocessor was to use (repeatedly) a soft IP core. Taking advantage of the sparse nature in matrices for loading, processing and storing vector data elements suggests a substantial performance gain. Our approach is mainly to exploit this aspect and to design the vector processor from scratch. In fact, we have designed our vector machine to handle matrix operations, with a primary focus on LU decomposition.

This paper discusses the architecture of our vector machine in terms of hardware and software. It also points out at specific improvements to the basic architecture proposed in [6] since our design is based on that early architecture. Then it presents the implementation details including the design hierarchy, implementation steps, verification, and implementation results. The paper also summarizes the performance of the machine. It concludes with direction for future research.

2 Architecture of the Vector Processor

The basic architecture is proposed in [6]. In our case, the vector computer consists of a vector processor and several memory modules. At present, the vector processor fetches, decodes and executes its own instructions. The processes of fetching and decoding are overlapped. The execution is sequential. One instruction is completely executed before the next one is started.

It contains a program memory and eight data memory modules. The data memory modules are Low-Order Interleaved (LOI) and their access is pipelined. The program memory and the data memories are High-Order Interleaved (HOI). The processor has four 16-bit vector registers with sixteen elements in each. The complexity of the vector processor was kept low in this effort because we first wanted to see the viability of our FPGA-based approach. The processor will be enhanced substantially in the near future. It also contains a few scalar registers such as vector length register and vector mask register. The machine can handle only integer numbers at present. Each instruction is represented by a word (16 bits). The upper 8bits represent the op-code. The lower 8bits represent the address of the operand (if any). Operands are accessed using the direct, implied and register addressing modes. All the operations on vector data are vector-register based.

The processor can access 128 words of instruction memory. Each of the eight data memory modules can have 128 words (total = 128×8 = 1024 words). The memory capacities can be easily expanded by providing additional address lines to the processor. However, there is a limit imposed by the FPGA platform that we used. The data memories are alternately accessed in (almost) consecutive clock cycles using two control lines. The memory mapping is shown in Table 1.

Type	**Address**
Instruction Memory	000-7FFH
Data Memory 0	800-FF0H
Data Memory 1	801-FF1H
Data Memory 2	802-FF2H
Data Memory 3	803-FF3H
Data Memory 4	804-FF4H
Data Memory 5	805-FF5H
Data Memory 6	806-FF6H
Data Memory 7	807-FF7H

Table 1: Memory Mapping of the Vector Computer

As seen from the above table, the instruction (program) and data memories are high order interleaved as the MS part of the address selects one between the two. The data memory modules are pipelined sharing the interleaved address space (as shown in Fig. 1). The LS three bits of the last hexadecimal digit of the data address selects the module. The next bit of the data address is left unused. This means that 800H and 808H maps to the same data address, as we have only eight data modules. This arrangement simplifies the readability of the addresses as shown in the above table. This implies that eight data elements can be read from/written into the memory with a single memory access. The data memories are accessed implicitly in low-order interleaved locations (in almost consecutive cycles) but the data within individual modules are stored in consecutive addresses. It ensures a low access latency that is characteristic of vector processors.

All the memory modules are connected to the same address and data buses of the processor. The MS hexadecimal digit of the address is decoded to select between the program and the (eight) data memories (they are high order

interleaved). This is evident from the address mapping shown in the above table. The data memories share the total data address space with interleaving. They are activated one at a time within an address cycle (using a control signal). As a result, the data from Data Memory 0 is put first on the data bus followed by the data from Data Memory 1, Data Memory 2, and so on (finally from Data Memory 7). However, the processor treats them as eight separate items and puts them in successive elements of a vector register. Writing into the memory is similar. It is to be noted that the processor generates only one address for accessing eight consecutive data elements as the memories are low order interleaved and pipelined.
With the current design of eight data memories, it requires two memory accesses to load into/store from a sixteen element vector register. In general, it would be equal to the elements of the vector register divided by the pipelined memory modules.

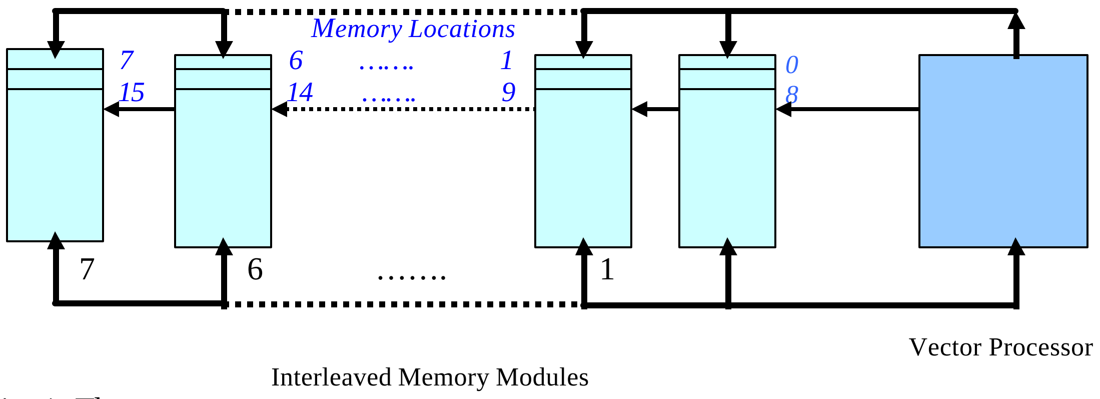


Fig. 1: The vector computer.

instruction is fetched and decoded only once. Subsequently, an entire vector can be loaded from/stored into the memory.

When the knowledge of the nonzero elements in a sparse vector is available, an index can be created with the positions of the nonzero elements. Using this index, those elements can be accessed efficiently. The 'indexed load' feature provides this facility. Similarly, if the sparse nature is regular (constant distance between nonzero elements), the nonzero elements could be accessed with a fixed stride.
This is supported by the 'stride load' feature. The 'stride load' feature also supports access of the elements on a column/row in a row-order /column-order stored matrix. The above two features are fully consistent with the 'scatter and gather' concepts for vector machines.

3 Instruction Set Architecture (ISA)

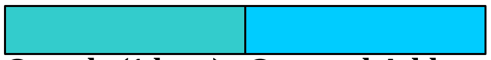
The vector processor's instructions can be divided into the following groups.

Load and store instructions
Load/store instructions access vectors. Apart from the normal (vector) load and (vector) store, these instructions also support (vector) load and (vector) store with an index as well as with a stride. These instructions are particularly helpful for sparse vector/matrix loading/storing. Each

Arithmetic instructions
These instructions allow addition, subtraction and multiplication of either two vectors or a scalar with a vector. The operations between the vectors are well reasoned. Addition and subtraction of vector elements with a scalar may be necessary when a 'change of reference' is needed. Similarly, multiplication of vector elements with a scalar may be necessary when 'scaling' is needed.

Comparison instructions
These instructions allow comparison of either two vectors or a scalar with a vector. Various types of comparisons are supported such as ‘Equal To’, ‘Greater Than’, ‘Less Than’, ‘Greater Than or Equal To’, ‘Less Than or Equal To’, and ‘Not Equal To’. The comparison of two vectors is needed to make a ‘relative’ decision. The comparison of elements in a vector with a scalar is needed to make an ‘absolute’ decision based on a reference.
Vector pre-processing instructions
These instructions allow preparation of the vector processor. They include loading the vector length register, loading the vector mask register, and creating an index vector. The vector length register can be used to dynamically control the size of the vector register or the number of memory accesses. The mask register can be used to indicate which elements in a vector register need to be processed. It also (indirectly) indicates the number of elements to be processed. This number may be helpful in estimating the processing time. All these features contribute to further efficiency.

Our implementation contains twenty four instructions at this time with provision for the load/store to be pipelined or non-pipelined. The instruction format is shown below.



Opcode (1 byte) Operand Address(1 byte)

Let us now focus on the potential enhancements compared to [6]. The architecture [6] could be improved by extending the role of the vector mask register and the vector length register. The mask register could be used to specify the element-offsets that should be used in the processing. This would support efficient processing of sparse vectors. This register should be used (when necessary) in all the Arithmetic, Comparison and Load/Store instructions. The vector length register could be used to dynamically specify the length of the vector to be processed. This also would support efficient processing of sparse vectors with all zero elements after a certain length. This register should also be used (when necessary) in all the Arithmetic, Comparison and Load/Store instructions.These are under active consideration for implementation.

Adding the elements of a vector register is required often in matrix-vector multiplication. This is not supported by the architecture [6]. Also, filling up a row (column) vector from each partial result is a useful operation. Both operations are not supported by the architecture in [6]. Both, however, are included in our design.

4 Annapolis Reconfigurable Platform

Reconfigurable chips (including FPGAs) can be programmed on the fly. They provide the flexibility of changing and adapting an implemented design. As a result they provide higher efficiency in applications that need frequent adaptations. The present state of the art in FPGAs provides a good compromise among cost, performance and power consumption. FPGAs are often used to implement circuits in consumer products. But little has been reported in the literature about the implementation of high-speed processors. Since current FPGAs are characterized by significant gate densities, we are motivated to implement the vector processor in an FPGA. In addition to providing flexibility, an FPGA-based platform provides the opportunity for hardware/software co-design.
The Annapolis Micro Systems (AMS) provides such a platform – WILDSTAR II board (Fig. 2). It provides a hardware board containing multiple FPGAs. The FPGAs can be programmed according to the needs of the application. They can communicate with the host computer. This provides the ability to run an application on the reconfigurable chips and read back the results for further processing. The cycle could be repeated with a different circuit implemented in the reconfigurable chips running a different application.

5 Implementation and Testing

The design is coded in behavioral VHDL. The top level design consists of the vector processor and memory modules (program and data). In the next level, the vector processor is composed of a group of components (Length register, Mask register, R1, F0), arithmetic unit, and comparator unit. Please refer to Fig. 3 for the details of the simulated version of the vector processor.

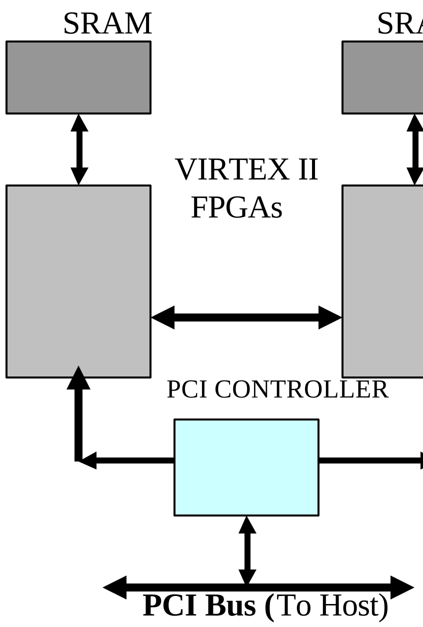


Fig. 2: The block diagram of the Annapolis Micro Systems WILDSTAR II reconfigurable board

5.1 Verification and Implementation Steps

The design has been simulated at the functional level. Every instruction has been executed and all of them produced the desired results. The design has been synthesized and mapped onto a Virtex II FPGA from Xilinx. The whole process was successful. However, there have been many modifications in the design (that was simulated earlier) in order to make it work in the actual target device. All the instructions have been tested successfully. A major modification is to implement a 'dual-port' memory. This was done to ensure compatibility with the available RAM in the FPGA.

The target FPGA is embedded in a board that can be accessed with high level language (HLL) routines. These routines allow one to program an FPGA (with the desired design file), set clock frequency, reset design, write to and read from the design.

The memory that works with the vector processor is first loaded with the program and the vector(s) using the HLL routine.

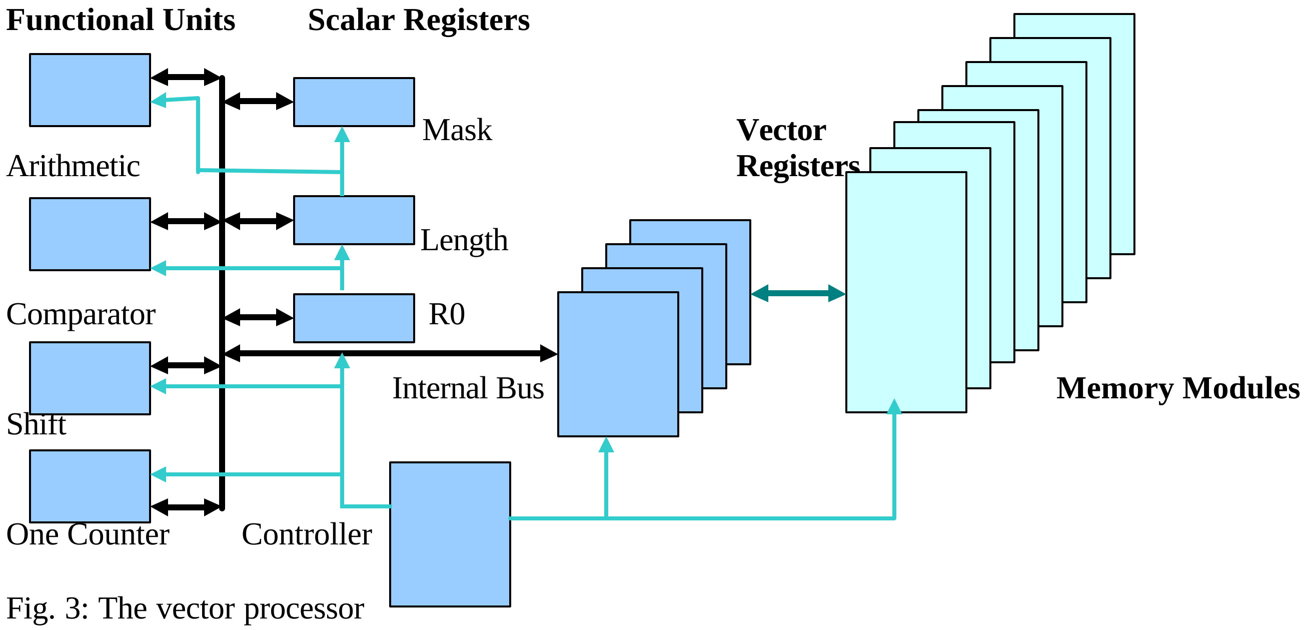


Fig. 3: The vector processor

Then the design is reset. The vector processor then processes the input vector and stores back the resulting vector in the memory. The memory is read back by the HLL routine to verify the result. As the size of the programs was very short, the results could be read back after a few seconds.

The total gate count, at present, for the complete design (the vector processor, one code memory and eight data memory modules) is about 677K equivalent gates. The maximum clock frequency used was 69MHz. However, due to the limited use of 'pipelining' techniques, some instructions execute at lower frequencies. The number of cycles needed for each instruction is shown in Tables 2 - 4.

5.2 Performance Summary

Cycles Per Element (**CPE**) is the clock cycles needed to process one element of the vector.

Instruction	CPE
Load vector	3
Store vector	2
Load vector with ***stride***	3
Store vector with ***stride***	3
Load vector with ***index***	3
Store vector with ***index***	3

Table 2: Load/Store Instructions

Instruction	CPI
Set all the bits of mask	1
Move R1 to the length register	1
Move the length register to R1	1
Move F0 to the mask register	1
Move the mask register to F0	1
Count the '1's in mask	1

Table 3: Pre-processing Instructions

Instruction	CPE
Add two vectors	3
Add scalar to a vector	2
Subtract two vectors	3
Subtract scalar from a vector	3
Subtract a vector from scalar	3
Multiply two vectors	3
Multiply vector with scalar	2
Compare two vectors	1
Compare vector with scalar	1
Create an index vector	2
Add the elements of C	3

Table 4: Arithmetic/Comparison Instructions

5.3 Performance Comparison

Table 5 lists the performance of the scalar and vector processors for two different cases.

8*8 Matrix Multiplication

	Scalar	Vector	Reduction (in %)
Instruction Count	256	48	81.25
Cycles Needed	10752	6552	39.06

16*16 Matrix Multiplication

	Scalar	Vector	Reduction (in %)
Instruction Count	1024	96	90.62
Cycles Needed	66560	26208	60.62

Table 5: Performance Comparisons

6 Acknowledgement

This work was supported in part by the U.S. Department of Energy under grant ER63384.

7 Future Work

The design will soon include the following improvements:

Pipelining: More pipelining would be introduced among the various components of the vector processor. This would increase the execution rate.

Expansion:

-More data memory modules are to be added.

- Longer vector registers are to be implemented.

Controlled Execution: Execution of instructions controlled by the mask and the length registers is to be incorporated.

Comprehensive testing: A real-life application would be used to run on the vector machine.

References:

[1] D. J. Tylavsky and A. Bose "Parallel Processing in Power System Computation", *IEEE Transactions on Power Systems*, Vol. 7, No. 2, May 1992.

[2] H. S. Huang and C. N. Lu "Efficient Storage Scheme and Algorithms for W-Matrix Vector Multiplication on Vector Computers", *IEEE*

Transactions on Power Systems, Vol. 9, No. 2, May 1994.

[3] G. P. Granelli, M. Montagna, and G. L. Pasini "A W-Matrix Based Fast Decoupled Load Flow for Contingency Studies on Vector Computers", *IEEE Transactions on Power Systems,* Vol. 8, No. 3, August 1993.

[4] M. K. Enns, W. F. Tinney, and F. L. Alvarado "Sparse Matrix Inverse Factors", *IEEE Transactions on Power Systems*, Vol. 5, No. 2, May 1990.

[5] F. L. Alvarado, D. C. Yu, and R. Betancourt "Partitioned Sparse A^{-1} Method", *IEEE Transactions on Power Systems*, Vol. 5, No. 2, May 1990.

[6] R. M. Russel "The CRAY-1 Processor System", *Comm. of the ACM* Vol. 21, No.1, January, 1978, 63-72.

[7] ElGindy. H and Yen-Liang Shue "On Sparse Matrix Vector Multiplication with FPGA Based System", *11th Annual IEEE Symposium on Field Programmable Custom Computing Machines*, April 2002.

[8] X. Wang and S.G. Ziavras "Parallel LU Factorization of Sparse Matrices on FPGA-Based Configurable Computing Engines", *Concurrency and Computation*, March 2004.

[9] J. L. Hennessy and D. A. Patterson "Computer Architecture: A Quantitative Approach", (Second Edition*), Morgan Kauffman Publishers Inc.*, 1996.