FPGA-Based Vector Processing for Solving Sparse Sets of Equations

Muhammad Z. Hasan and Sotirios G. Ziavras
Electrical and Computer Engineering Department
New Jersey Institute of Technology
Newark, NJ 07102

Abstract

The solution to a set of sparse linear equations $Ax = b$, where A is an $n\times n$ sparse matrix and b is an n-element vector, can be obtained using the W-matrix method. An enhanced vector processor is implemented on an FPGA for this problem. Performance results are presented. The effect of pipelined multiple functional units, multiple data buses, instruction chaining, hardware synchronization, pipelined scattering, matrix density, and distribution of non-zero elements is analyzed.

1 Introduction

We study power analysis [1]. This method can also be applied to neutron-deuteron scattering, ECG tracking, generation of pseudo-random numbers and verbal data analysis. Instead of a direct method [4], W-matrix is used [3]. It offers the potential for vector operations [2, 5].

Let L, D, and U represent the lower triangular, diagonal and upper triangular matrices, respectively, of $A = L.D.U$. Let $L^{-1} = W^L$ and $U^{-1} = W^U$. Then, the solution can be obtained as $x = W^U.D^{-1}.W^L.b$ [2, 3]. It involves three steps: $W^L.b = z$, $D^{-1}.z = y$, and $W^U.y = x$. Each W-matrix can be partitioned to increase sparsity. Let $L = L_1 L_2 \ldots L_n$, where L_i is an identity matrix with the i-th column the same as that of L. W^L can then be written as: $W^L = L^{-1} = (L_1L_2 \ldots L_n)^{-1} = L_n^{-1}\ldots L_2^{-1}L_1^{-1} = W_n \ldots W_2 W_1$, where W_i is L_i with the signs of the off-diagonal elements reversed. Thus, $x = (L_1^t)^{-1}(L_2^t)^{-1}\ldots (L_n^t)^{-1} D^{-1} L_n^{-1}\ldots L_2^{-1}L_1^{-1}b = (W_1^t)(W_2^t)\ldots(W_n^t) D^{-1} W_n\ldots W_2W_1b$. If we combine the W_i's with their transposes into two factors, then $x = (W_a^t)(W_b^t)D^{-1} W_bW_ab$. The number of non-zeros increases as we combine W factors. There exists a trade-off in performance between the numbers of non-zeros and serial operations.

Maintaining sparsity keeps multiplications to a minimum. Node ordering minimizes the number of non-zeros [3]; it requires to find the number of fill-ins in a matrix after a certain operation. Hardware support to c*ount the non-zeros* after an operation is needed. The ordering phase includes sorting the nodes based on counts. Thus, a group of vector registers should support *sorting by magnitude*. The capability to *identify the non-zeros* and *form a vector* is needed. It is also important to record the column number of non-zeros for appropriate multiplications (*form a vector with the column indices* of non-zero elements). A mechanism to *control the number of multiplications based on the number of non-zeros* is desirable to reduce processor cycles. However, such a multiplication needs post processing. As the vector is formed exclusively from non-zeros belonging to different rows, the resulting vector also contains elements for different rows of the result. Multiplication result elements belonging to the same row must be added together to find the appropriate row element in the result. It is needed to selectively *add elements of a vector*. This also requires keeping a record of non-zeros on each row of the original matrix. In bus power computation of load flow, the contribution of diagonal elements is evaluated separately from the contribution of off-diagonal elements. It is needed *to access only the diagonal elements*. Our vector computer consists of a vector processor with pipelined 32-bit FPUs, a memory controller for pre-fetching and several memory modules. At present, it fetches, decodes and executes its own instructions. It contains a program memory and eight data memory modules.

2 Customized Instructions

We decided to add the following instructions to a standard vector processor: **1.** Count the number of non-zeros in a matrix. **2.** Sort the rows based on the counts. **3.** Select a row based on the minimum count. **4.** Add only certain elements of two vectors. **5.** Add only certain elements of a vector with certain other elements of the same vector. **6.** Multiplication followed by addition. **7.** Load-add-store with the same indices. **8.** Create pseudo-columns from many partially filled columns. **9.** Access only diagonal elements. **10.** Create a vector from all non-zeros of a matrix. **11.** Create a vector from the column indices of all non-zeros of a matrix. **12.** Create a vector from the last row numbers of partitions of a matrix. Most of the instructions were modeled in VHDL. Subsequently they were synthesized and mapped to a Xilinx Virtex II FPGA on the Annapolis Micro Systems WildStar II board. Multiplication followed by addition, load-add-store with the same indices, create a vector from the last row numbers of partitions of a matrix and add only certain elements of a vector are under development.

3 Performance Evaluation

14-, 30- and 57-bus IEEE power systems containing 27.55%, 12% and 6% non-zeros, respectively, are used. We calculated bus currents using matrix-vector multiplication. To evaluate the sparse techniques, all the admittance matrices in our experiments were made sparse with density between 2 and 7%. For each row of the sparse matrix, a load operation is followed by a count operation. At the end of such steps, a vector containing only the non-zeros of the matrix is created. Also, a vector is created containing elements with the number of non-zeros on each row. A column index vector is also available at the end; it is used to load the appropriate elements of the multiplier vector. Then, multiplication is carried out and subsequently the partial results are added to form the elements of the final result. The cycles needed for bus current calculations of the modified IEEE systems were determined experimentally. At the application level (for bus current calculations), the sparse handling techniques reduce the processor cycles by 20-25%. The density vary between 2 % and 7%. The execution cycles vary almost linearly with the density of the matrix. For the 30- and 57-bus systems, partial results are added to form the final result. This overhead is significant for low-density matrices. As a result, the curve is non-linear at low densities. But for higher densities, the curve is linear as the above overhead becomes insignificant.

In W-matrix, a matrix row is often loaded and multiplied with a vector. This readily indicates that a chain of the type 'load followed by multiply' would speed up the execution of the method. Moreover, resulting elements are formed (after multiplication) by accumulating partial results. The resulting elements are subsequently stored in the memory. This process provides the potential for another chaining of the type 'add followed by store'. These chains were implemented and tested both at the instruction and application levels. The results show that the cycle savings are higher at the operation than the application level. At the application level, there are other 'non-chained' instructions in addition to the chained instruction. So, the effect of 'chaining' is less pronounced.

Hardware synchronization is achieved by means of a 'ready' signal activated by the memory controller. The signal is 'polled' by the vector processor whenever it wants a data transfer to/from the memory. The cycle savings are more pronounced when vector-chaining is used. The processor with software synchronization and chaining consumes less cycles than with hardware synchronization but without chaining. There is a time gap between 'request' for a load to the memory controller and the availability of the data vector. So, it is wise to issue such a 'request' ahead of time. A three stage chain was implemented based on this idea. A 'request' for a load was chained to the existing 'Load-Multiply' chain, initiating a 'pre-fetch' for the next vector data. By employing different types of chaining, we can save on cycles to calculate bus currents by 27-31%.

For systems involving sparse matrices, the non-zeros are 'gathered' in a vector. Then, the appropriate elements of the multiplier vector are loaded into another vector using indices. After the multiplication, the partial results belonging to a particular row need to be added and placed appropriately in the resulting vector. This implies 'scattering' of the resulting elements. The above addition and scattering require variable numbers of cycles depending on the number of rows the results are distributed into. The cycle increase is steeper in the case of non-pipelined scattering than that of the pipelined case. The number of rows in which the non-zeros are distributed or the results are to be scattered affects the cycles. But with pipelining, this increase is nominal as results are produced every two clock cycles.

As all the elements of vector operand registers are simultaneously available, it is worth processing them concurrently by employing multiple functional units. This is done by engaging eight adders and eight multipliers to carry out the arithmetic operations faster. As the size of the vector registers is sixteen, it needs two iterations to apply all the inputs to the functional units. The results are collected after the specified latency of the adder and multiplier, as mentioned earlier. The cycle savings are marginal. However, for larger size vector registers, the savings would be higher. For example, it takes 11 cycles to execute 'Multiplication' between two vector registers of length 16. It would take only 12 cycles to carry out the same operation for vector registers of length 24 and 13 cycles for vector registers of length 32. Also, there is only one multiplication after 'gathering' non-zeros. It implies that there are many 'Load' operations before one multiplication. So, the cycle savings during 'Load' operations would have more impact on the overall application. Realizing this fact, the data bus between the vector processor and the memory controller was broadened to 4*32 bits from 32 bits. This enables the vector processor to transfer all the elements of a vector register in four cycles. It is observed (as expected) that the impact of widening the data bus is more than that of employing multiple functional units. The effect of multiple functional units and data buses is favorable. However, the cycle savings fall for larger systems as there are more partial results that need to be loaded/stored and added.

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

References

[1] D. J. Tylavsky and A. Bose "Parallel Processing in Power System Computation," *IEEE Trans. Power Sys.*, 7-2, May 1992.

[2] G. P. Granelli, M. Montagna, and G. L. Pasini "A W-Matrix Based Fast Decoupled Load Flow for Contingency Studies on Vector Computers," *IEEE Trans. Power Sys.*, 8-3, August 1993.

[3] F. L. Alvarado, D. C. Yu, and R. Betancourt "Partitioned Sparse A^{-1} Method," *IEEE Trans. Power Sys.*, 5-2, May 1990.

[4] X. Wang and S.G. Ziavras "Parallel LU Factorization of Sparse Matrices on FPGA- Based Configurable Computing Engines," *Concur. Comput*., March 2004, pp.319-343.

[5] A. Gomez, R. Betancourt, "Implementation of the Fast Decoupled Load Flow on a Vector Computer," *IEEE Trans. Power Sys.*, 5- 3, Aug. 1990.