PERFORMANCE ANALYSIS OF LOAD FLOW COMPUTATION USING FPGA[1]

J. Johnson, P. Vachranukunkiet, S. Tiwari, P. Nagvajara, C. Nwankpa
Drexel University
Philadelphia, PA

Abstract – Full-AC load flow constitutes a core computation in power system analysis. Current techniques for solving full-AC load flow rely on the iterative solution of the load flow equations using the Newton-Raphson method, and the bulk of this computation is devoted to solving a sparse-linear system needed for the update each iteration. In this paper, we evaluate the performance gain that is possible with a hardware implementation of a sparse-linear solver using a Field Programmable Gate Array (FPGA). Using benchmark data, from four representative power systems, we compare the performance of a software solution using UMFPACK, a state-of-the-art sparse linear solver, running on a high-end PC with the proposed hardware solution. Performance of the hardware solution is obtained from a parameterized simulation of the hardware that allows a variety of architectural configurations and can take into account different FPGA technology. Our studies show that the proposed hardware solution can provide an order of magnitude performance gain using existing FPGA technology.

Keywords: FPGA, load flow, LU decomposition, sparse matrix

1 INTRODUCTION

In power flow computations, sparse linear solvers are used to calculate the update vector for the Jacobian matrix at each iterative step during Newton-Raphson iteration for load flow solution. The bottleneck in Newton power flow iteration is the solutions of a sparse system of linear equations, which is required each iteration, and typically consumes 85% of the execution time in large-scale power systems [1]. In practice many related load flow computations are performed as part of contingency analysis. These multiple load flow computations can easily be performed in parallel. It is the purpose of this paper to speedup the computation of an individual load flow computation.

Typical power flow computation is performed using general-purpose personal computing platforms based on processors such as the Intel Pentium 4. High performance uni-processor sparse linear solvers on these platforms utilize only roughly 1-4% of the floating-point throughput. Attempts have also been made to increase performance through the use of cluster computing [1], however due to the irregular data flow and small granularity of the problem these methods do not scale well.

In this paper we propose to use application-specific hardware to reduce the computation time for the solution of the sparse linear systems arising in load flow computation. The use of special-purpose hardware reduces the overhead in computation, better utilizes floating-point hardware, and provides fine-grained parallelism. The use of Field Programmable Gate Arrays (FPGA) provides a convenient platform to design and implement such hardware. Other attempts have been made to utilize FPGAs to accelerate the computation of sparse linear systems by using soft-core embedded processors on FPGAs [2]. Our approach utilizes the FPGA resources differently. By building hardware that is specifically designed to solve the sparse matrices found in power system calculations, rather than utilizing general-purpose processors and parallel processing, we seek to improve the efficiency of the linear solver and hence reduce the computing time compared to traditional platforms.

In this paper, we describe the design of hardware for the direct solution of sparse linear systems. The hardware design takes advantage of properties of the matrices arising in power system computation obtained from a detailed analysis of actual power systems. A simulation study, using the benchmark systems is performed to compare the performance gain of the proposed hardware as compared to existing software solutions.

1.1 Overview of FPGA Technology

Field programmable gate arrays (FPGA) are reconfigurable logic devices that allow users to configure device operation through software. Arrays of programmable logic blocks, combined with programmable I/O blocks, and programmable interconnect form the underlying reconfigurable fabric for FPGAs. Software code, typically written in a hardware description language (HDL) such as VHDL or Verilog, is mapped to these devices allowing the user to specify the functionality of the FPGA. FPGAs can be programmed multiple times, unlike other programmable logic devices (PLD), which

[1] The research reported in this paper was supported by the United States Department of Energy (DOE) under Grant No. CH11171. The authors are also grateful to PJM Interconnection for data provided.

can only be programmed once. In addition, unlike application specific integrated circuits (ASIC) there is not a long lead time in between design and test since the designer does not have to wait for masks and then for manufacturing to return a part for test. The user simply compiles the design for the desired FPGA and then programs the device.

In 2002, FPGA device density was in the thousands of logic gates range with clock rates in the tens of megahertz. Today (2005), device densities are in the millions for logic gates with clock rates of 500 MHz for synthesized logic at a cost of sever hundred to several thousand dollars for high end FPGAs (see www.xilinx.com). Beyond increases in logic density, the addition of high performance embedded arithmetic units, large amounts of embedded memory, high speed embedded processor cores, and high speed I/O has allowed FPGAs to grow beyond just simple prototyping devices. FPGAs have evolved to the point where high performance floating-point computation is now feasible and with additional hardware devoted to floating point computation FPGAs can outperform high-end personal computers [3]. Based on synthesis results, a high-end FPGA such as a Xilinx Virtex-4 contains enough embedded multiplier and logic resources to support over 40 high-speed single precision floating-point multipliers operating at 300 MHz. Unlike personal computer processors, FPGAs have the advantage of being able to be mapped to the desired application, rather than being designed to perform well across a wide range of applications. This allows the greatest amount of utilization of the available resources on the FPGA.

1.2 Overview of Power Flow by Newton's Method

Power flow solution via Newton method [4] involves iterating the following equation:

$$-J \cdot \Delta x = f(x) \quad (1)$$

Until f(x) = 0 is satisfied. The Jacobian, J, of the power system, is a large highly sparse matrix (very few non-zero entries), which while not symmetric has a symmetric pattern of non-zero elements. Delta x is a vector of the change in the voltage magnitude and phase angle for the current iteration. And f(x) is a vector representing the real and imaginary power mismatch. The above set of equations are of the form Ax = B, which can be solved using a direct linear solver. Direct linear solvers, perform Gaussian Elimination, which is performed by decomposing the matrix A into Lower (L) and Upper (U) triangular factors, followed by forward and backward elimination to solve for the unknowns.

Since the Jacobian is a sparse matrix, dense matrix linear solvers are not as efficient as specialized solvers designed for sparse matrices. These solvers do not compute the non-zero entries however they do suffer from a phenomenon known as fill-in. Fill-in of a sparse matrix during LU decomposition arises when a previously zero entry in the matrix becomes non-zero during the elimination process. Large amounts of fill-in will degrade the performance of sparse solvers by increasing the number of mathematical operations required for the LU decomposition.

By using an appropriate elimination order, the amount of fill-in can be dramatically reduced [5]. Selecting the optimal ordering is an NP complete problem [6], however, effective heuristics are known that work quite well for the matrices in load flow computation. In addition, sparse matrix solvers do not share the regular computational pattern of dense matrix solvers and as a consequence there is significant overhead in accessing and maintaining data structures that utilize the sparse data.

2 BENCHMARK SYSTEM AND PROPERTIES

An in-depth analysis of the elimination process was performed using four power systems of various sizes. Two of the systems, the 1648 Bus and 7917 Bus systems, were obtained from the PSS/E data collection, and the other two systems, the 10279 Bus and 26829 Bus systems, were obtained from power systems used by PJM Interconnect. The purpose of the analysis was to obtain features of the systems arising in typical power system computation that could be utilized in the design of special-purpose hardware. In addition the systems were also used to benchmark the proposed hardware and compare it against a state-of-the art software solution.

Table 1 summarizes the number of branches and number of non-zeros (NNZ) in the Ybus matrices and the size and number of non-zeros of typical Jacobian matrices for the benchmark systems.

System	Branches	NNZ YBUS	NNZ Jacobian	Jacobian Size
1648 Bus	2,602	6,680	21,196	2,982
7917 Bus	13,014	32,211	105,522	14,508
10279 Bus	14,571	37,755	134,621	19,285
26829 Bus	38,238	99,225	351,200	50,092

Table 1: Summary of Power System Matrices

2.1 Operation Count (opcounts) and Fill-In Study

The number of floating-point subtractions (FSUB), multiplications (FMUL), divisions (FDIV), and the growth of the matrices as a result of fill-in were collected for the Jacobians and are summarized in Table 2 and Table 3. This information was used to project performance and estimate storage requirements for the resulting L and U factors. The MATLAB functions COLAMD and SYMAMD were used to determine elimination ordering. The number of fill-ins is equal to the number of non-zeros in the L and U factors compared to the Jacobian. SYMAMD utilizes the symmetric structure of the non-zeros in the Jacobian and provides a substantial reduction in fill-in and arithmetic operations.

System	NNZ LU	# FDIV	# FMUL	# FSUB
1648 Bus	82,378	0.039	1.088	1.027
7917 Bus	468,757	0.225	9.405	9.041
10279 Bus	450,941	0.206	6.267	5.951
26829 Bus	1,424,023	0.661	40.110	39.037

Table 2: COLAMD NNZ and MFLOP Count

System	NNZ LU	# FDIV	# FMUL	# FSUB
1648 Bus	45,595	0.021	0.268	0.243
7917 Bus	236,705	0.110	1.737	1.606
10279 Bus	293,688	0.130	1.976	1.817
26829 Bus	868,883	0.392	11.223	10.705

Table 3: SYMAMD NNZ and MFLOP Count

Given a floating point throughput limited calculation, the performance is proportional to the number of floating point operations (FLOPs) divided by the throughput of the floating-point hardware. Based on the flop counts and a throughput of 200 million floating point operations per second (MFLOPs), the projected solve time for the largest system is approximately 0.11 seconds, solved using the SYMAMD pre-permutation.

The storage requirement for the L and U factors is proportional to the number of non-zero elements for a non-restoring design, where the original inputs are over written by the resulting solution. A sparse matrix storage scheme is much more space efficient than dense storage when dealing with matrices of large size, but very few non-zero values. Row compressed storage is a common method used by sparse solvers [7]. This scheme stores the non-zero values of a sparse matrix by a set of row vectors, one vector containing non-zero values and the other containing the corresponding column indices. Assuming 32-bit indices and 32-bit values for the matrix entries, the amount of storage required to store the L and U factors is approximately 7 MB for the largest test system, solved using the SYMAMD pre-permutation. This amount of storage is well within the space available on off the shelf memory chips (www.micron.com).

2.2 Elimination Details

The pattern of non-zero elements that arise during the elimination process was analyzed in order to estimate the available parallelism, intermediate storage requirements, and potential for data reuse.

Table 4 shows the average number of non-zero elements in the rows and columns of the matrix as the elimination process proceeds. It is used as an estimate for the amount of row or column parallelism (separate rows can be updated in parallel) that can be achieved by utilizing multiple FPUs. Observe that the reduced fill-in obtained by SYMAMD leads to reduced parallelism. The amount of resources on an FPGA can support the number of floating-point units required to achieve maximal parallelism.

Figure 1 shows the distribution of the number of non-zero entries in the rows that occur during the LU factorization for the 1648 Bus system. The x-axis is the number of non-zeros and the vertical bars indicate the number of rows with that many non-zero entries. The bulk of the elimination rows are at or below the average. This histogram is representative of all of the row and column results for both COLAMD and SYMAMD across all four benchmark systems.

The maximum number of non-zero elements in a row determines the size of buffers that store rows. Table 5 shows the maximum NNZ in a row for the entire row as well as the portion of the row used during elimination (Table 5). Using 32-bit indices and 32-bit floating point values, 19Kbits of storage is required for the largest elimination row of the 26829 Bus system. Todays FPGAs contain thousands of Kbits of high speed embedded memory, which is enough to buffer hundreds of rows or columns.

	COLAMD		SYMAMD	
System	**Row**	**Col**	**Row**	**Col**
1648 Bus	13.4	14.2	7.2	8.1
7917 Bus	15.8	16.5	7.8	8.6
10279 Bus	11.7	11.7	7.5	7.8
26829 Bus	14.2	14.2	8.5	8.9

Table 4: Mean elimination row and column size

	COLAMD		SYMAMD	
System	**Elim.**	**Matrix**	**Elim.**	**Matrix**
1648 Bus	90	231	66	218
7917 Bus	147	477	125	275
10279 Bus	116	416	186	519
26829 Bus	258	886	293	865

Table 5: Maximum elimination row and matrix row size

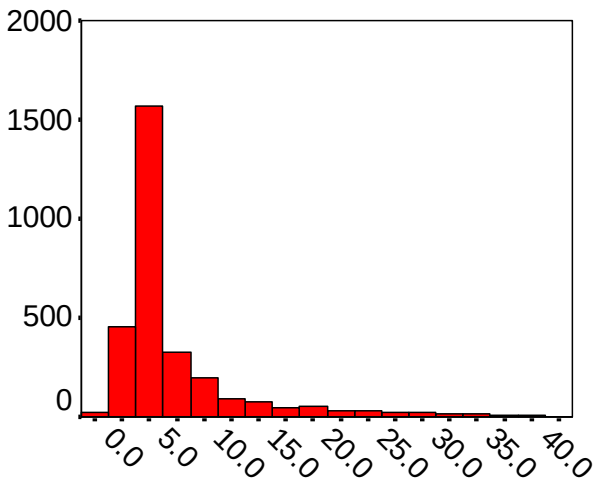

Figure 1: SYMAMD elimination row histogram

The amount of reuse between subsequent elimination rows impacts the potential efficiency of a memory hierarchy. Rows that are reused from one iteration to the next can be cached and a row not used in the previous iteration can be prefetched. Rows reused in subsequent eliminations do not allow pivot search to occur in parallel with computation during elimination since some values being calculated are necessary for the next pivot search. Table 6 shows that slightly more than half of the rows are reused between subsequent iterations.

System	COLAMD	SYMAMD
1648 Bus	64%	53%
7917 Bus	64%	54%
10279 Bus	60%	54%
26829 Bus	60%	54%

Table 6: Mean Percentage of Non-Zero Row Elements Reused between Successive Elimination Rows

2.3 Software Performance

The comparison system used in the benchmarks is a 2.60 GHz Pentium 4 computer running Mandrake Linux 9.2. Software was compiled using gcc v3.3.1 (flags –O3 –fPIC) and utilize ATLAS CBLAS libraries. Three high end sparse solvers were tested, UMFPACK (Un-Symmetric Multi-Frontal Package) [8], SuperLU (http://www.cs.berkeley.edu/~demmel/SuperLU.html), and WSMP (Watson Sparse Matrix Package: http://www-users.cs.umn.edu/~agupta/wsmp.html). Reported times are CPU time. Of the three solvers, UMFPACK gave the best results on all four systems. The numerical solve time, floating point operations per second, and floating point efficiency of the comparison system were gathered using the power system matrices. Despite the 2.6 GHz clock speed and impressive computational power of the Pentium 4 benchmark system, even the highest performance software was only able to utilize a fraction of the total available floating point resources.

Data Name	UMFPACK Numeric Factor time (s)	Mflops reported by UMFPACK	Efficiency % (obtained flop * 100 /peak flop)
1648Bus	0.03	27.42	1.05
7917Bus	0.13	34.69	1.33
10278Bus	0.16	24.9	0.96
26829Bus	0.49	89.74	3.45

Table 7: Comparison System Benchmarks

3 HARDWARE DESIGN

The proposed hardware implements the Gaussian elimination with partial pivoting LU factorization algorithm is shown below:

For each column in the matrix loop
Search column for pivot element
Pivot if necessary
Normalize pivot column
Update remaining portion of matrix:
Multiply to form sub-matrix update
Update value or create fill-in
End loop

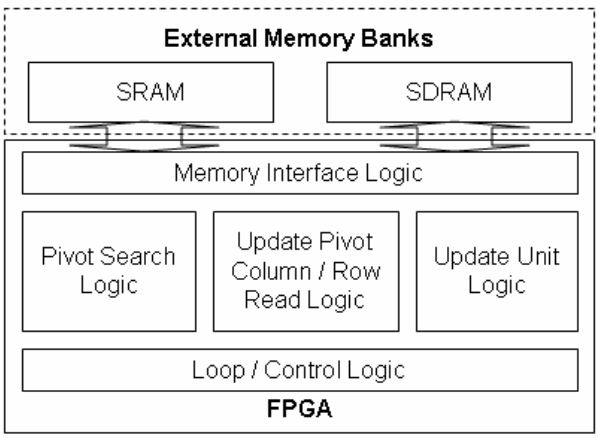

Figure 2: Architecture

In order to maximize performance, the design of the LU hardware focused on maintaining regular computation and memory access patterns that are parallelized wherever possible. Figure 2 shows a block diagram of the overall design. The Pivot Search Logic, Update Pivot Column / Row Read Logic, and Update Unit Logic blocks each perform a portion of the overall LU algorithm, with control and memory access handled by the other two blocks on the FPGA. The memory hierarchy to which the processing logic connects, consists of a write-back row-cache implemented in a fast random-access memory such as SRAM, as well as a larger block of SDRAM which is used to store the matrix rows which do not fit into the row cache. Our design makes full use of Dual-Ported SRAM memories which allow memory reads and writes to occur simultaneously. An additional bank of random-access memory is also required to store the colmap. The colmap is a redundant column-wise array of the indices of non-zero elements in the matrix. The purpose of the colmap is to increase the efficiency of the pivot search.

The minimum size of the row cache is equal to the amount of memory required to store the largest sub-matrix update, which is proportional to the product of

the maximum number of non-zero elements in an elimination row and the maximum number of non-zero elements in an elimination column. A cache of this size insures that all row writes for a given sub-matrix update will be cache hits and will occur at cache speed, since all row misses will only occur during pivot search insuring row availability during sub-matrix update. The parameters to calculate the amount of cache required was extracted previously (See Section 2.2).

For the initial design, the cache row size, sdram row size, and colmap row size exceeds the maximum counts which were extracted during the initial matrix profiling. This insures that the memories will have enough space to store the worst case, howeve this also means that there will be inefficiencies in the storage for the typical case.

Not shown in the hardware diagrams are the facilities required to maintain row pivoting information, which is implemented by lookup tables, and memory pointers which keep track of the row to memory mappings for the compressed row storage format. An additional table is used to keep track of elimination progress to avoid reading row values that are not part of the sub-matrix update, i.e. elements that have already been eliminated. The size of these lookup tables scales linearly with matrix size. Depending on the size of the matrices to be solved and the available on-chip resources, these tables can be stored in embedded memory on-chip or if they are larger, in off-chip memory.

3.1 Pivot Search Logic

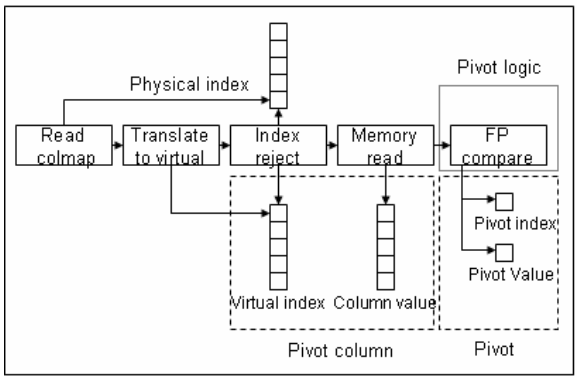

Figure 3: Pivot Search

The pivot search logic block performs the pivot search prior to each sub-matrix elimination step. A memory read of the column map followed by indexing logic creates the indices for the pivot column. The column map stores the physical address of rows in a particular column. The physical address must then be translated to the corresponding matrix row number. Since the colmap may contain rows which have already been eliminated, index rejection logic is required to select the appropriate indices. The physical addresses corresponding to these indices are then used to access the values from memory. The values are compared sequentially as they arrive to obtain the absolute maximum value and index. This value is stored in a register as the pivot element. The physical indices, virtual indices, and floating point values for the pivot column are also stored in registers to be used by the pivot column update logic. The minimum amount of memory required for these registers is equal to the product of the largest elimination column in the matrix and the amount of storage for two indices and one value. After pivoting is complete, a background update is sent to lookup tables (not shown) to maintain the correct row and address mappings for future translations.

3.2 Update Pivot Column/Row Read Logic

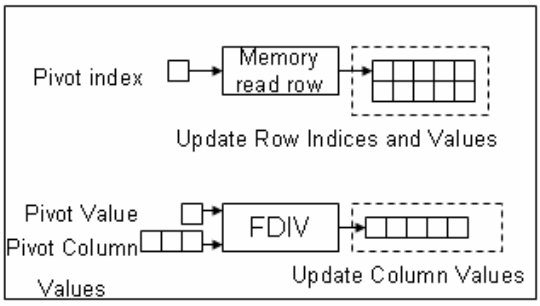

Figure 4: Update Pivot Column/Row Read

The update pivot column/row read logic block performs normalization prior to elimination in parallel with the pivot row requested from memory. The pivot index, pivot value, and pivot column are from the pivot search logic (Figure 3). After the pivot row is completely read from memory into buffers, and at least one of the update column entries has been created the next part of the elimination logic can proceed. Our floating point division unit divides by multiplication via Newton-Raphson iteration. By unrolling the iterations, our floating-point divider design allows fully pipelined operation. With a pipeline rate of 1 per clock cycle, all of the remaining divides will complete before they are requested by the update unit logic. The update row indices and values, and the normalized update column indices (physical and virtual) and values are stored in registers. The minimum size of this register space is equal to the product of the maximum elimination row in the matrix and the amount of space required to store three indices and two values.

3.3 Update Unit Logic

The update unit logic performs the remaining computations required during the elimination step. In this design, the processing logic can support multiple update units operating in parallel. Each update unit is tasked with completing the elimination for a single row of the sub-matrix to be updated. Each update unit contains a floating point multiplier and adder to perform

the required arithmetic as well as other logic that operates on the column indices. The computational logic operates in parallel with the memory and indexing logic to maximize utilization of all of the logic units. The number of update units that can fit on the FPGA is limited by the amount of on-chip embedded memory available, the number of multipliers available, and also the available logic resources. Our estimates show that a high end FPGA has enough resources to support over 16 of these units.

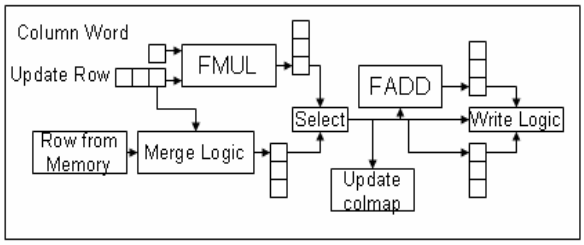

Figure 5: Update Unit

The merge logic maintains the row order and also determines if the multiplier output is a fill-in, an update, and also determines if the sub-matrix row element should be copied into the new row. If an entry is a fill-in, the colmap table is updated in the background to reflect the newly added index. Updates that result in a zero value are not pruned from the colmap and they are also stored in the row result. All logic in the update unit is pipelined for maximum throughput. Once the front end of the update unit has completed processing, it can request the next available row to be operated on in order to eliminate idle logic cycles. After all of the pending row eliminations for the current sub-matrix update have been completed, the next pivot search can begin.

4 HARDWARE PERFORMANCE MODEL

A program was written in C, to project the performance of the hardware design. This program performs the elimination in the same fashion as the hardware would, and operates on an input file that contains the power system Jacobian matrix. The number of clock cycles the hardware requires to obtain the L and U factors is accumulated based on the latency and throughput of the processing logic and memory hierarchy. This cycle count allows us to compute the computation time based on clock speed.

	Update Units Solve Time (s)				
System	**1**	**2**	**4**	**8**	**16**
1648 Bus	0.0041	0.0028	0.0021	0.0018	0.0017
7917 Bus	0.0243	0.0157	0.0116	0.0097	0.0090
10279 Bus	0.0311	0.0202	0.0149	0.0126	0.0117
26829 Bus	0.1378	0.0817	0.0543	0.0415	0.0359

Table 8: Solve time for LU Hardware

The memory model used for accessing SDRAM utilized three parameters to model the average behavior of a SDRAM memory: overall latency, burst read length, and latency between bursts. For the SRAM and embedded memories the model parameter was latency from request to data output. The only parameter for the floating-point hardware is the pipeline length, which affects the latency of results. All units were assumed to have a pipeline rate of 1 per clock cycle, even the floating-point division hardware. Extending the model to multiple instances of update unit logic was done by simulating round-robin service for each update unit by the memory hierarchy assuming a proportional increase in bandwidth. The clock rate for memory and logic was assumed to be synchronous at 200 MHz. This system clock rate was chosen because this was the maximum clock rate reported by VHDL synthesis for our floating-point units.

Table 8 shows the projected performance of the proposed LU hardware architecture based on the four test power systems using the SYMAMD pre-permutation at a clock rate of 200 MHz. The simulated hardware performance achieves approximately 80% of the performance predicted by opcount. The projected performance using COLAMD (not shown) is less than SYMAMD, regardless of the number of update units used. Figure 6 shows the relative speedup of the LU hardware by increasing the number of update units. Scaling to even four update units does not net an equivalent speedup in overall solve time. The reason for this is apparent in Figures 7 and 8.

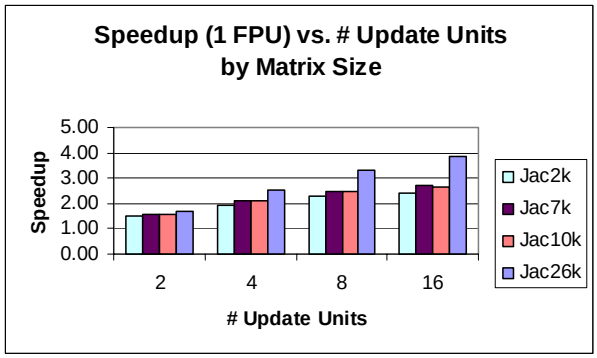

Figure 6: Speedup and Scaling of LU Hardware

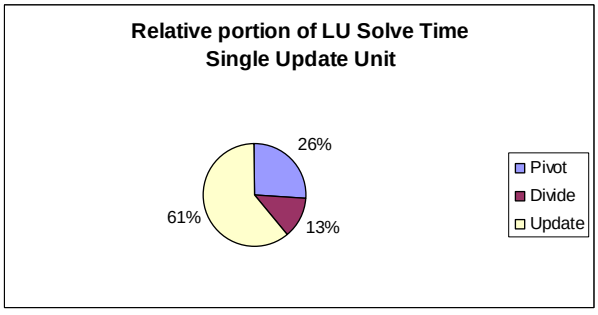

Figure 7: Relative Solve Time for Single Update Unit

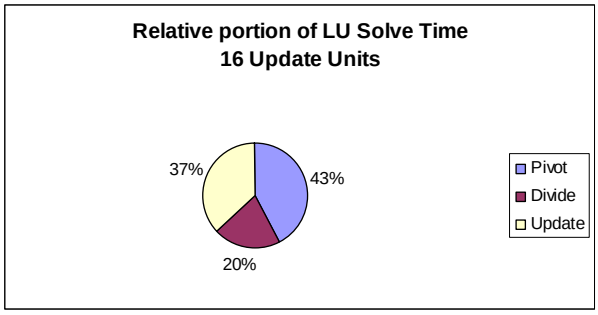

Figure 8: Relative Solve Time for 16 Update Units

Figures 7 and 8 show the relative time spent in the three portions of the LU hardware solution. For the single update unit configuration (Figure 7), the majority of the time is spent performing the matrix update. Roughly a quarter of the time is spent searching for the pivot value. Figure 8 shows that as the number of update units increases, the amount of time spent during pivot search begins the dominate the overall solve time.

Figure 9 shows the predicted performance ratio of performance of the LU hardware at 200 MHz to the Pentium 4 benchmark system. This data predicts a speedup over the Pentium 4 of a factor of 10 with only 4 update units and approximately 15 with 16 units.

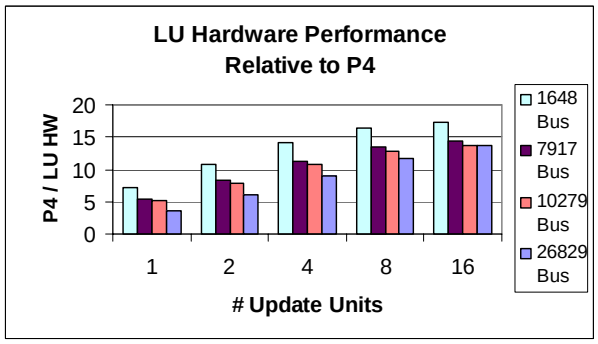

Figure 9: LU Hardware Performance Relative to Pentium 4

5 CONCLUSION

Our results show that special-purpose hardware implemented using existing FPGA technology can provide a 10x or higher performance increase, compared to software solutions using state-of-the-art sparse linear solvers, for the LU decompositions that arise in typical power flow computations. The speedup is obtained by utilizing special-purpose hardware to reduce the overhead to access and maintain data structures for sparse matrices and the use of fine-grained parallelism which performs multiple row updates in parallel. The reduction in overhead dramatically improves the utiliztion of the floating-point units as compared to software solutions on general-purpose processors. In addition to LU decomposition, the application of this approach can be used to design hardware to accelerate other computations such as linear function optimization. An FPGA implementation of the proposed design is underway in order to confirm the simulation study.

A limiting factor in the proposed design is the time for pivot search. Eliminating or reducing the pivot search time will result in additional gains in performance. Such improvement may be obtained by overlapping the next pivot search with the current update, however, this is limited by the amount of reuse in rows from one elimination step to the next. In the future we will explore this approach and other for reducing pivot time along with the use of other approaches usch as the Bordered Diagonal Block (BDB) technique which may provide additional parallelism.

REFERENCES

[1] Feng Tu and A. J. Flueck, A message-passing distributed-memory parallel power flow algorithm, Power Engineering Society Winter Meeting, 2002. IEEE, Volume 1 (2002), pp 211 – 216.

[2] X. Wang and S.G. Ziavras, Parallel Direct Solution of Linear Equations on FPGA-Based Machines, Workshop on Parallel and Distributed Real-Time Systems (17th Annual IEEE International Parallel and Distributed Processing Symposium), Nice, France, April 22-23, 2003.

[3] Keith Underwood, FPGAs vs. CPUs: Trends in peak floating point performance, ACM/SIGDA Twelfth ACM International Symposium on Field-Programmable Gate Arrays, (Monterrey, CA), February 2004.

[4] A. R. Bergen, V. Vittal, Power Systems Analysis, 2nd Edition, Prentice Hall, 2000.

[5] P. Amestoy, T. A. Davis, and I. S. Duff, An approximate minimum degree ordering algorithm, SIAM J. Matrix Anal. Applic., 17 (1996), pp. 886-905.

[6] M. Yannakakis. Computing the minimum fill-in is NP-Complete, SIAM J. Alg. Disc. Meth., 2 (1981), pp. 77-79.

[7] W. H. Press, S. A. Teukolsky, W. T. Vettering and Brian P. Flannery, Numerical Recipes in C: the Art of Scientific Computing, 2nd Edition, New York: Cambridge University Press, 1992.

[8] T. A. Davis and I. S. Duff, An unsymmetric-pattern multifrontal method for sparse lu factorization, SIAM J. Matrix Anal. Applic., 18 (1997), pp. 140-158.