A Configurable Architecture for Sparse LU Decomposition on Matrices with Arbitrary Patterns

Xinying Wang, Phillip H. Jones and Joseph Zambreno
Department of Electrical and Computer Engineering
Iowa State University, Ames, Iowa, USA
{xinying, phjones, zambreno}@iastate.edu

ABSTRACT

Sparse LU decomposition has been widely used to solve sparse linear systems of equations found in many scientific and engineering applications, such as circuit simulation, power system modeling and computer vision. However, it is considered a computationally expensive factorization tool. While parallel implementations have been explored to accelerate sparse LU decomposition, irregular sparsity patterns often limit their performance gains. Prior FPGA-based accelerators have been customized to domain-specific sparsity patterns of pre-ordered symmetric matrices. In this paper, we present an efficient architecture for sparse LU decomposition that supports both symmetric and asymmetric sparse matrices with arbitrary sparsity patterns. The control structure of our architecture parallelizes computation and pivoting operations. Also, on-chip resource utilization is configured based on properties of the matrices being processed. Our experimental results show a 1.6 to 14× speedup over an optimized software implementation for benchmarks containing a wide range of sparsity patterns.

Keywords

Architecture, FPGA, Sparse LU decomposition, Crout method

1. INTRODUCTION

Many important scientific and engineering applications (e.g. circuit simulation [3, 4], power system modeling [18], and image processing [9]) have at their core a large system of sparse linear equations that must be solved. Sparse LU decomposition has been widely used to solve such systems of equations, but it is considered a computationally expensive factorization tool.

Left-looking, Right-looking and Crout are the main direct methods for sparse LU decomposition, but are not efficient when implemented in software. Supernodal [5] and Multifrontal [10] are approaches that lend themselves well to software implementations of sparse LU decomposition. Supernodal considers the input matrix as sets of continuous columns with the same nonzero structure, while Multifrontal organizes large sparse datasets into small dense matrices. Parallel implementations of Supernodal and Multifrontal have been demonstrated on multi-core platforms and GPUs with shared or distributed memory [3, 4, 8, 11]. However, Supernodes may not exist in some applications, and it is a challenge to parallelize the Multifrontal method in a distributed memory system.

Although many FPGA-based architectures have been proposed for accelerating LU decomposition of dense matrices, only a few have been proposed for accelerating sparse matrix LU decomposition [13, 15, 16, 17]. Of these architectures, they either target domain-specific sparsity patterns [13, 15, 16], or require a pre-ordered symmetric matrix [17].

In this paper, we propose an FPGA-based architecture for sparse LU decomposition, which can efficiently process sparse matrices having arbitrary sparsity patterns. Our architecture mitigates issues associated with arbitrary sparsity patterns and extracts parallelism in two primary ways. First, it factorizes columns from the lower triangular part of a matrices in parallel with factorizing rows from the upper triangular part of the matrix. Second, our control structure performs pivoting operations in parallel with the factorizations of rows and columns. Our experimental results show a 1.6 to 14× speed up over an optimized software implementation for benchmarks containing a wide range of sparsity patterns.

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2015) Boston, MA, USA, June 1-2, 2015.

2. THEORETICAL BACKGROUND

2.1 Sparse LU decomposition with pivoting

LU decomposition factorizes a matrix A into two matrices, L and U, as shown in eq. (1)

$$A = LU \tag{1}$$

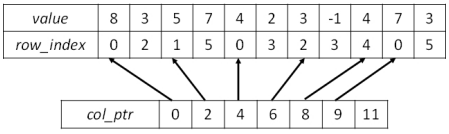
Here, A is an $m \times n$ matrix, L is an $m \times n$ lower triangular matrix, and U is an $n \times n$ upper triangular matrix. This linear process is called sparse LU decomposition when matrix A is sparse.

Sparse matrices commonly appear in a broad variety of scientific and engineering applications. These matrices are characterized by having relatively few non-zero elements. This property can be leveraged to store them in efficient formats. The two most popular of these formats are Compressed Sparse Column (CSC) format and Compressed Sparse Row (CSR). Fig. 1 illustrates the CSC and CSR format of a sparse matrix. Both CSC and CSR formats consist of three components: 1) an array of non-zero values, 2) an integer array of row or column indexes of those non-zero elements, and 3) an array of pointers where each element points the first non-zero element of a column or a row.

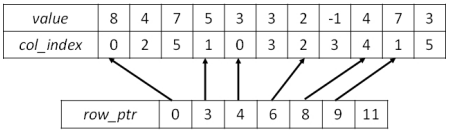
Stability and Accuracy. To ensure stability during LU decomposition, pivoting operations are performed to remove

$$\begin{pmatrix} 8 & 0 & 4 & 0 & 0 & 7 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 7 & 0 & 0 & 0 & 3 \end{pmatrix}$$

(a) Example sparse matrix



(b) Compressed Sparse Column format



(c) Compressed Sparse Row format

Figure 1: Compact storage formats for sparse matrices

zero elements from the diagonal of matrix A. This can be accomplished conceptually by applying a pivoting matrix, P, to matrix A as $PA{=}LU$. P is an $m \times m$ matrix in which each column has one element of value 1 and all other elements are 0 [12].

2.2 Algorithms for Sparse LU Decomposition

Direct methods for sparse LU decomposition include Left-looking, Right-looking, and Crout [14], which generally contain division operations and update processes (see Fig. 2).

2.2.1 *Left-looking*

The Left-looking algorithm factorizes a matrix in a column-by-column manner. Before normalizing a given column, non-zero elements of the previously factored column are used to update the current column elements $A(i,j)$ using the equation $A(i,j) = A(i,j) - L(i,k) * U(k,j)$, where $k = 1 \cdots \min(i,j)$ (see Fig. 2a).

2.2.2 *Right-looking*

The Right looking algorithm first factorizes a column from the lower triangular part of a matrix, then uses the resulting non-zero elements of that column to update the affected components in the rest of the matrix by using the equation $A(i,k) = A(i,k) - L(i,j) * U(j,k)$, where $k = j+1 \cdots N$, j is the index of current factored column, and N is the column dimension of matrix A (see Fig. 2b).

2.2.3 *Crout*

Similarly to the Left-looking algorithm, the Crout method performs updates with previously factored elements before normalizing a given vector. The difference is that the Crout method operates both on columns and rows, while the Left-looking algorithm only operates on columns (see Fig. 2c).

3. RELATED WORK

FPGA architectures have been shown to be effective in accelerating a wide range of matrix operations. However, accelerating the LU decomposition of large sparse matrices with arbitrary sparcity patterns is a challenge for hardware acceleration. In [16], the authors propose an efficient sparse LU decomposition architecture targeting the power flow analysis application domain. Their FPGA-based architecture implements the right looking algorithm and includes hardware mechanisms for pivoting operations. The performance of their architecture is primarily I/O bandwidth limited. Kapre et al., in [13, 15], introduced an FPGA implementation of sparse LU decomposition for the circuit simulation application domain. A matrix factorization compute graph is generated to capture the static sparsity pattern of the application domain, and it is exploited to distribute the explicit data flow representation of computation across processing elements. For their approach, they also illustrate that their performance gains are highly sensitive to the manner in which computations are distributed across the available processing elements. Wu et al., in [17], devised a more general hardware design to support sparse LU decomposition for a wider range of application domains. Their architecture parallelizes the left-looking algorithm to efficiently support processing symmetric positive definite or diagonally dominant matrices. One factor limiting the performance of their architecture arises from dynamically determining data dependency during their column-by-column factorization, which leads to their processing elements stalling for the purpose of synchronizing to resolve data dependency across processing elements.

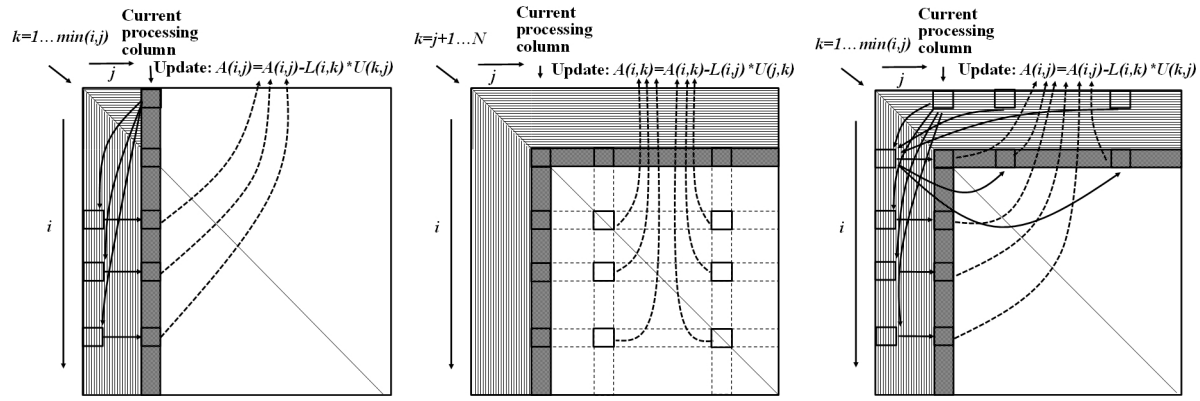
4. THE PARALLEL SPARSE LU DECOMPOSITION ALGORITHM

In general, the process of LU factorization primarily consists of pivot, division, and update operations. These operations can be performed in parallel when no data dependencies exist among them. Our architecture aims to extract this type of parallelism specifically from the Crout method of sparse LU decomposition at three different levels: 1) we leverage the fact that the structure of the Crout method naturally allows parallelization of processing columns and rows from the lower and upper triangular part of a matrix respectively (Fig. 2c), 2) we perform block partitions of the matrix to identify elements for which update operations can be performed in parallel and thus share an update processing element (see Fig. 3, the computations of matrix elements in the blocks with identical shaded pattern can be assigned to the same update processing element), and 3) the structure of the architecture control logic performs pivot operations in parallel with update and division operations.

5. THE SPARSE LU DECOMPOSITION ARCHITECTURE

Our architecture consists of four primary types of processing elements: 1) Input, 2) Update, 3) Division, and 4) Pivot. Fig. 4 provides a high-level view of how these processing elements are related within our architecture.

As a brief summary of the flow of data through our architecture, the Input processing elements are responsible for the initial processing of columns from the lower triangular part of the matrix and rows from the upper triangular part of the matrix. The output of the Input processing elements is then forwarded to the Update processing elements, which



(a) Left-looking (b) Right-looking (c) Crout

Figure 2: Popular algorithms for sparse LU decomposition

Algorithm 1 Our parallel sparse LU decomposition algorithm

```
1: [m n] ← size(A)
2: for i ← 1 to min m, n do
3:    {1.Perform pivoting operations in parallel}
4:    A(i,i) ← max(A(:,i))
5:    for j ← 1 to n do
6:       A(i,j) ↔ A(max_index, j)
7:    end for
8:    {2.Update column entries before division in parallel}
9:    for j ← i to m do
10:      if F_L(j,k) ∈ block(pid_row) then
11:         A(j,i) ← A(j,i) − F_L(j,i)
12:      end if
13:   end for
14:   {3.Parallelized division operations}
15:   for j ← i + 1 to m do
16:      if A(j,i) ≠ 0 then
17:         L(j,i) ← A(j,i)/A(i,i)
18:      end if
19:   end for
20:   {4.Update row entries after division in parallel}
21:   for k ← i + 1 to n do
22:      if F_U(j,k) ∈ block(pid_col) then
23:         U(i,k) ← A(i,k) − F_U(i,k)
24:      end if
25:   end for
26:   {5.Calculate Update factors in parallel}
27:   for z ← 1 to i do
28:      for j ← z + 1 to m(or n) do
29:         for k ← i + 1 to i + 1 + block_size do
30:            if L(j,i) ≠ 0 and U(i,k) ≠ 0 and F_L(j,k)(or
               F_U(j,k)) ∈ block(pid_row( or pid_col)) then
31:               F_L(j,k) = F_L(j,k) + L(j,i) ∗ U(i,k)
32:               (or F_U(j,k) = F_U(j,k) + L(j,i) ∗ U(i,k))
33:            end if
34:         end for
35:      end for
36:   end for
37: end for
```

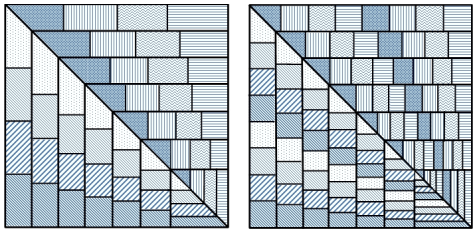


Figure 3: Two examples of block partitioning.

implement a major portion of the Crout method's computation. After a set of entries within a column of the lower triangular part of the matrix have been updated, the Division processing element normalizes these entries with respect to the matrix's diagonal element associated with that column. The Pivot processing element directs movement between Update processing elements in a manner that results in the pivoting of matrix elements. Additionally, the Pivot processing element manages the output data stream.

The number of processing elements used for our architecture is configurable based on three factors 1) amount of on-chip resources, 2) matrix block partitioning strategy used, and 3) matrix properties (e.g. matrix dimensions, sparsity rate).

5.1 Input

The Input PEs have two variations. One is for initially processing columns from the lower triangular part of the matrix, and one is for processing rows from the upper triangular part of the matrix. As shown in Fig. 5, the architecture of these two are similar, with the column version having some additional logic.

The additional logic for column processing serves three purposes: 1) it reduces the amount of processing by detecting if the update of an element involves multiplication by zero, 2) it determines if updating the current element will impact updating elements to be processed in adjacent columns from the upper triangular part of the matrix, and 3) it obtains the pivot information for each column by locating the element with the largest value.

The functionality that the two variations of the Input PE

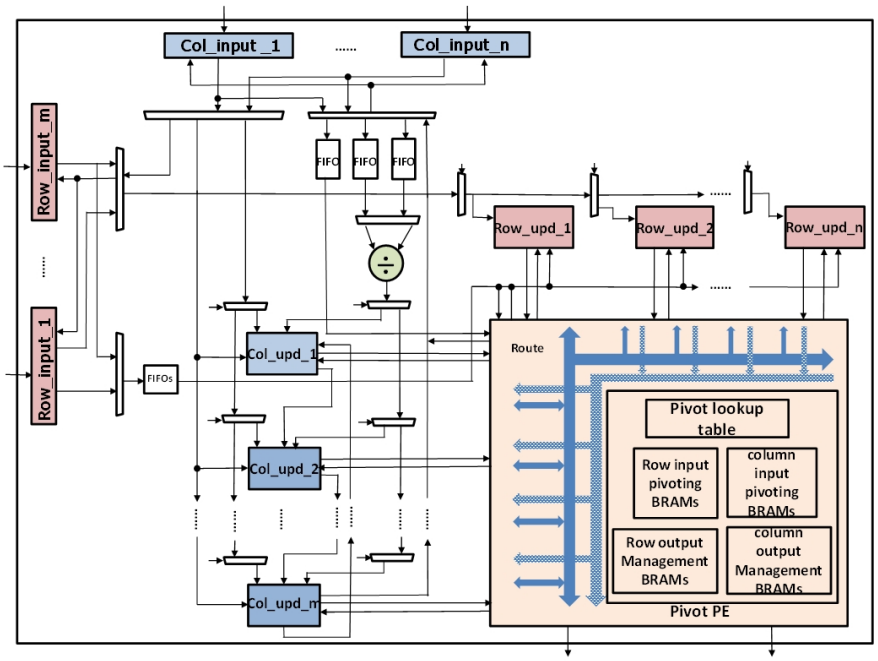


Figure 4: The block diagram of our sparse LU decomposition architecture

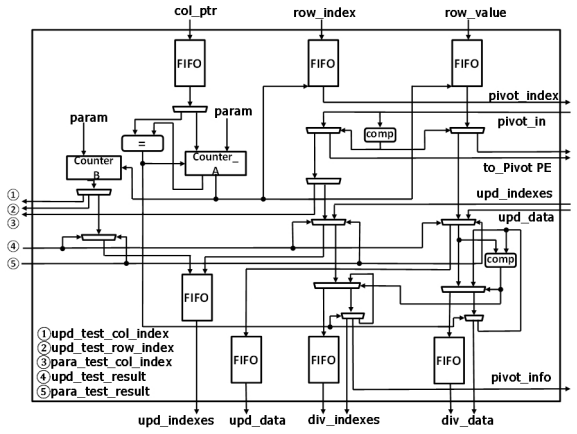


Figure 5: Input PE architecture.

share is that based on the row (column) index information received from the input, either 1) the element of the row (column) read in will be routed to a row (column) Update processing element, or 2) if the element is to be pivoted from the lower triangular part of the matrix to the upper triangular part of the matrix or visa-verse, then the element is directed to the Pivot PE, which will be responsible for forwarding the element to the proper Update PE in the future. Additionally, both Input PE types take as input matrix row (column) information formatted as CSR (CSC).

5.2 Update

The Update PEs are responsible for helping compute the core update equation of the Crout method, $A(i,j) = A(i,j) - L(i,k) * U(k,j)$, where $k = 1 \cdots \min(i,j)$. It is composed of two primary components: 1) an Update Compute Engine, and 2) an Update Input Manager. Fig. 6 gives a block-level diagram of the Update PE.

The Update Compute Engine performs multiply accumulate operations over rows (columns) that are currently being updated. It is fed appropriate values from the Update Input Manager. The Update Input Manager provides two services. Firstly, it manages what we call an "Update Check Table" to indicate if a given element of a row or table needs to be updated. Secondly, it maintains a data structure (using Address Pointer Table and *Dist_Table*) that keeps track of addresses of non-zero values that are stored in Data mem of Fig. 6. These are the values fed to the Update Compute Engine from Data mem.

Once a matrix element has been updated by the Update PE, then if it is associated with the lower triangular part of the matrix its value is normalized with respect to matrix's diagonal element associated with that element's column.

5.3 Pivot

As indicated in Section 2.1, pivot operations are required to ensure numerical stability during sparse LU decomposition. The Pivot PE (see lower right side of Fig. 4) performs pivots by acting as a router between the lower triangular and upper triangular part of the matrix. Based on an element's ⟨row, column⟩-index information, when read into an Input PE, the Pivot PE determines if that element should pivot (i.e. be transferred from the lower to upper triangular part of the matrix or visa-verse). Lookup tables within the Pivot PE are used to store and evaluate ⟨row, column⟩-index information to determine if and where an element should be pivoted. The Pivot PE is also responsible for buffering and sending elements to off-chip storage that have been completely processed. In other words, an element's value is stored off-chip when it can no longer be affected by the processing of future elements.

6. EXPERIMENTS AND EVALUATIONS

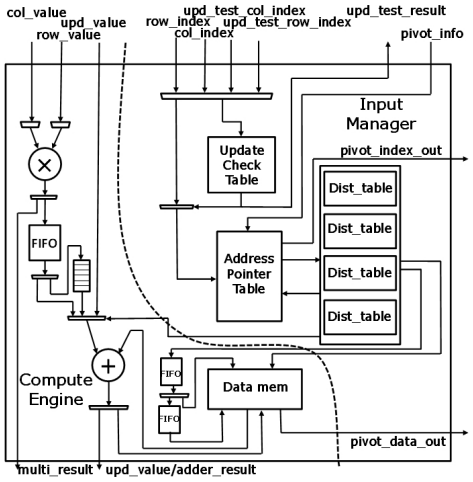
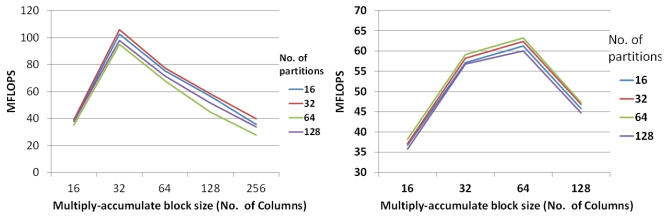


Figure 6: Update PE architecture.



(a) 494_bus (b) *photogrammetry*

Figure 7: Impact of multiply-accumulate block size and No. of Input matrix partitions on performance.

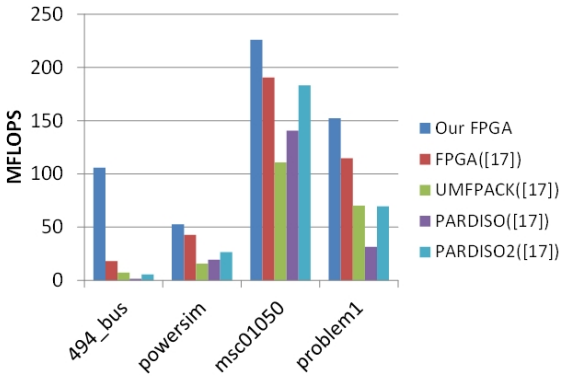


Figure 8: Throughput comparison between our architecture, and the FPGA and software implementations in [17].

6.1 Implementation and Experimental Setup

Our architecture was evaluated using a Convey Computer HC-2 system [2]. The HC-2 is a hybrid-core system that couples a standard Intel based server with a co-processor board that hosts four Xilinx Virtex-5 XC5VLX330 FPGAs and a specialized memory subsystem for high throughput random memory access. We implemented our architecture for one of the HC-2's Virtex-5 FPGAs, referred to by Convey as an Application Engine (AE).

All floating-point operations were performed using double-precision computing cores generated with Xilinx's Coregen [1]. The pipeline latency for each type of core used in our design was: 9, 14, and 57 clock cycles for multiplication, addition, and division, respectfully. The modular structure of our design allows us to easily customize the number and types of PEs implemented to best suit the properties of the matrix being processed. For example, for a "skinny" and "tall" matrix we implement more Update PEs for processing columns than rows. The largest design that fits on the Virtex-5 LX330 consisted of 64 Update PEs. The FPGA resource utilization was 76.4% LUTs, 48.4% DSPs, and 87.5% BRAMs. It could be run at a maximum frequency of 176 MHz, which easily meets the 150 MHz frequency typically used for designs run using the HC-2.

Benchmarks were selected from the University of Florida sparse matrix collection [7] as workloads for our performance evaluation. As can be seen in Table 1, the selected benchmarks cover a wide range matrix types in terms of Dimension, Element pattern (e.g. symmetric, asymmetric, rectangular), and Sparsity rate. All matrices were split into upper triangular and lower triangular matrices and stored using CSR and CSC formats respectively.

6.2 Performance Analysis

In this section, we first investigate how different architectural configuration parameters impact our design's performance. Then we evaluate the performance of our approach against an existing FPGA implementation and against several software implementations, including Matlab.

With respect to the impact of parameter settings on performance, we chose to examine two parameters: 1) the multiply-accumulate blocks size of the Update PE, where block size refers to the number of columns or rows processed as a batch while a previous column is being normalized using the division operator, and 2) the number of partitions the input matrix is divided into. As Fig. 7 illustrates, multiply-accumulate block size has a significant impact on performance. Choosing a block size that is too big or too small causes up to a 2.5× difference in performance for the representative benchmarks shown. Deeper analysis showed that when block size was too large, the Update PE spends more time searching the block space for non-zero values, thus increasing multiply-accumulate stalls. When block size was too small, the multiply-accumulate engine of the Update PE will finish before the normalization process and will need to block until normalization completes. An interesting observation is that different benchmark matrices have different optimal multiply-accumulate block sizes. With respect to the number of partitions used to split up the input matrix, we observed a much smaller impact.

Figure 8 and 9 compares the performance of our architecture against others. When compared against the FPGA architecture of [17] our throughput is 1.2 to 5× better. Additionally, while both architectures target acceleration across arbitrary application domains, our architecture does not require the input matrix to be reordered into a diagonally dominate matrix. When compared to the software approaches in [17], where optimized linear solvers UMFPACK [6] and PARDISO [11] were used on a single-core (UMFPACK [17]), single-core (PARDISO [17]), and 4-core (PARDISO2 [17])

Table 1: Experimental benchmark matrices, properties, and their performance.

Matrix	Dimensions	Sparse rate	nnz(L+U)	Pattern	Application domain	Matlab performance (ms)	Our FPGA performance (ms)
494_bus	494 × 494	0.68%	13,425	sym	Power network	1.92	0.359
Powersim	15838 × 15838	0.03%	136,472	sym	Power network	19.7	12.3
Msc01050	1050 × 1050	2.38%	61,234	sym	Structural problem	5.89	1.20
problem1	415 × 415	1.61%	12,242	sym	FEM	1.33	0.533
qc2354	2354 × 2354	7.22%	926,209	sym	Electromagnetic	652	107
West1505	1505 × 1505	0.24%	42,688	asym	Chemical process	31.3	13.7
CAG_mat1916	1916 × 1916	5.34%	2,542,599	asym	Combinatorial problem	3920	279
lpi_chemcom	288 × 744	0.74%	1,878	rec	Linear programming	0.203	0.112
Well1850	1850 × 712	0.66%	122,104	rec	Least square problem	50.2	4.90
photogrammetry	1388 × 390	2.18%	213,891	rec	Computer vision	74.3	6.28

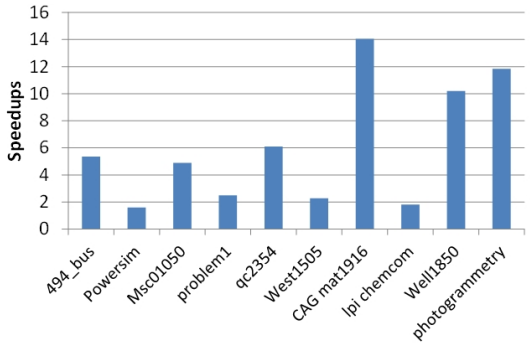


Figure 9: Speedups of our FPGA implementation normalized to Matlab's LU decomposition routine.

CPU, we show a 1.2 to 75× speedup. A comparison with Matlab's LU decomposition function (Fig. 9) run on a 2.2 GHz dual core Intel Xeon processor with 16GB of memory shows a speedup of 1.6 to 14×.

7. CONCLUSIONS

An FPGA-based architecture is presented that performs sparse LU decomposition using a modified version of the Crout method. As opposed to targeting matrices with domain-specific sparsity patterns, it efficiently processes input matrices with arbitrary sparsity patterns, without requiring pre-ordering of the input matrix. For sparse matrix benchmarks having a wide range of properties, our experimental results show a 1.6 to 14× speedup over an optimized software solution.

Acknowledgements

This work is supported in part by the National Science Foundation (NSF) under awards CNS-1116810 and CCF-1149539.

8. REFERENCES

[1] LogiCORE IP Floating-Point Xilinx Operator data sheet. March 2011.

[2] The HC-2 convey computer architecture overview. 2012.

[3] X. Chen, L. Ren, Y. Wang, and H. Yang. GPU-accelerated sparse LU factorization for circuit simulation with performance modeling. *Journal of IEEE Transactions on Parallel and Distributed Systems*, 26(3):786–795, March 2015.

[4] X. Chen, Y. Wang, and H. Yang. Nicslu: An adaptive sparse matrix solver for parallel circuit simulation. *Journal of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(2):261–274, Feb 2013.

[5] C. Cleveland Ashcraft, R. G. Grimes, J. G. Lewis, B. W. Peyton, H. D. Simon, and P. E. Bjørstad. Progress in sparse matrix methods for large linear systems on vector supercomputers. *International Journal of High Performance Computing Applications*, 1(4):10–30, 1987.

[6] T. A. Davis. Algorithm 832: UMFPACK V4.3—an Unsymmetric-pattern Multifrontal Method. *Journal of ACM Transaction on Mathematical Software*, 30(2):196–199, June 2004.

[7] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *Journal of ACM Transaction on Mathematical Software*, 38(1):1:1–1:25, Dec. 2011.

[8] J. Demmel, J. Gilbert, and X. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, 1999.

[9] D. Donoho and Y. Tsaig. Fast solution of ℓ_1 -norm minimization problems when the solution may be sparse. *Journal of IEEE Transactions on Information Theory*, 54(11):4789–4812, Nov 2008.

[10] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear. *Journal of ACM Transaction on Mathematical Software*, 9(3):302–325, Sept. 1983.

[11] K. Gårtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, 20:475–487, 2004.

[12] G. H. Golub and C. F. Van Loan. *Matrix computations*. Johns Hopkins Univ. Press, Baltimore, MD, USA, 1996.

[13] N. Kapre and A. DeHon. Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs. In *Proceedings of International Conference on Field-Programmable Technology*, pages 190–198, Dec 2009.

[14] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd ed.: The Art of Scientific Computing*. Cambridge Univ. Press, New York, NY, 2007.

[15] Siddhartha and N. Kapre. Breaking sequential dependencies in FPGA-based sparse LU factorization. In *Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines*, pages 60–63, May 2014.

[16] P. Vachranukunkiet. *Power Flow Computation Using Field Programmable Gate Arrays*. Drexel University, 2007.

[17] G. Wu, X. Xie, Y. Dou, J. Sun, D. Wu, and Y. Li. Parallelizing sparse LU decomposition on FPGAs. In *Proceedings of International Conference on Field-Programmable Technology*, pages 352–359, Dec 2012.

[18] D. Yu and H. Wang. A new parallel LU decomposition method. *Journal of IEEE Transactions on Power Systems*, 5(1):303–310, Feb 1990.