Parallel Sparse Matrix Solution for Circuit Simulation on FPGAs

Tarek Nechma and Mark Zwolinski, *Senior Member, IEEE,*

Abstract—SPICE is the de facto standard for circuit simulation. However, accurate SPICE simulations of today's sub-micron circuits can often take days or weeks on conventional processors. A SPICE simulation is an iterative process that consists of two phases per iteration: model evaluation followed by a matrix solution. The model evaluation phase has been found to be easily parallelizable, unlike the subsequent phase, which involves the solution of highly sparse and asymmetric matrices. In this paper, we present an FPGA implementation of a sparse matrix solver, geared towards matrices that arise in SPICE circuit simulations. Our approach combines static pivoting with symbolic analysis to compute an accurate task flow-graph which efficiently exploits parallelism at multiple granularities and sustains high floating-point data rates. We also present a quantitative comparison between the performance of our hardware prototype and state-of-the-art software packages running on a general-purpose PC. We report average speed-ups of 9.65×, 11.83×, and 17.21× against UMFPACK, KLU, and Kundert Sparse matrix packages, respectively.

Index Terms—Hardware acceleration, sparse matrices, SPICE, FPGA arithmetic, pipeline and parallel arithmetic and logic structures.

1 Introduction

SPICE is the de facto standard for circuit simulation. With decreasing feature sizes, the need for detailed simulations of circuits has grown in recent years. Despite the increasing performance of standard processors, there is a "verification gap" between the needs of designers and the power of simulators. Essentially, this gap has arisen because SPICE is extremely difficult to parallelize. There are two main phases within each iteration in a SPICE simulation: device evaluation and matrix solution. Device evaluation is trivially parallel; matrix solution is not, and, moreover, there are barriers between the two phases. The work described here is concerned with accelerating the matrix solution phase, by using FPGA technology.

Non-linear circuit analysis in the time domain typically requires several thousand repeated solutions of the matrix at different iterations and time-steps. Moreover, the Newton-Raphson method typically needs three to four iterations to produce the solution of each system of non-linear equations [1]. Thus, the efficient solution of the linear equations plays a critical role in the total computation time.

Extensive research has been conducted on accelerating sparse LU decomposition on general-purpose PCs and HPCs [2, 3, 4, 5, 6]. With the advent of the FPGA supercomputing paradigm, a number of researchers have investigated FPGA acceleration for LU decomposition. Most of these implementations [7, 8, 9, 10] are generally tailored towards a specific scientific problem, where the matrix to be solved is structurally symmetric and diagonally dominant. Such matrices are relatively easy to solve and parallelize, compared to asymmetric ones. In [9], *Johnson et al* presented a right-looking (i.e. sub-matrix based) LU sparse matrix decomposition on FPGAs for the symmetric Jacobian matrices that arise in power flow computations. Fine-grained parallelism is achieved by the use of a special cache designed to improve the utilization of multiple floating-point units. For matrix package UMFPACK, the authors report an order of magnitude speed-up for LU decomposition compared to a 3.2 GHz Pentium 4. Accelerating the front and back substitutions was not considered in their work.

In [7, 8, 10], *Wang et al* presented a parallel sparse LU decomposition that has been implemented using an FPGA-based shared-memory multiprocessor architecture, known as MPoPC. Each processing element (PE) consists of an Altera Nios processor attached to a single-precision floating-point unit. Coarse-grained parallelization is achieved using node tearing to partition sparse matrices into small diagonal sub-problems that can be solved in parallel. They report a considerable speed-up for power flow analysis compared to a single Nios implementation. Their results, however, were not compared to existing FPGA or software implementations. Moreover, their comparison was not made with modern and highly-optimized LU matrix kernels such as KLU and UMFPAK.

In [11], *Kapre et al* proposed an FPGA accelerator geared towards parallelizing the sparse matrix solution phase of the Spice3f5 open-source simulator. Using a 250 MHz Xilinx Virtex-5 FPGA, the authors reported speed-ups of 1.2-64 times over a KLU direct solver running on an Intel Core i7 965 processor. The KLU direct solver

• *The authors are with Electronics and Computer Science, Faculty of Physical Science and Engineering, University of Southampton, Southampton, UK, SO17 1BJ.*
E-mail: {tn06r,mz}@ecs.soton.ac.uk

reorganizes matrices into sub-blocks, using the Block Triangular Form (BTF) technique, and then factorizes them using the Gilbert-Peierls Algorithm. KLU has been written specifically to target SPICE circuit matrices that arise in the Newton-Raphson iteration. The acceleration, reported by *Kapre et al*, is achieved by leveraging the standalone symbolic analysis capabilities of the KLU solver to generate a data flow graph of the required fine-grained floating-point operations. The data flow graph is then mapped to a network of PEs interconnected by a packet-switched Bidirectional Mesh routing network. The proposed architecture, however, focuses mainly on exploiting the fine-grained data flow parallelism available in KLU, potentially overlooking the coarser-grained parallelism inherently present in sparse matrices.

More recently, *Wu et al* [12] presented a 16-PE FPGA implementation of the Gilbert-Peierls Algorithm, on an Altera Stratix III EP3SL340. Fine-grained parallelism is harnessed by sharing the computation burden to compute a given column over a number of PEs. No other levels of parallelism were explicitly considered. The reported speed-ups varied in the range 0.5-5.36$\times$ when compared to KLU runtimes on an Intel i7 930 microprocessor. However, the benchmark matrices used are relatively small in terms of their size, and also have a small number of non-zeros. The latter is the main factor that dictates the number of FLOPs needed to factorize a given matrix. Moreover results were not compared to previous FPGA implementations.

Both approaches detailed in [11, 12] employ a parallel implementation of the Gilbert-Peierls matrix factorization algorithm. They specifically apply the symbolic analysis stage of the algorithm to compute which additional matrix elements will become non-zero after the actual factorization. Using the predicated non-zero structure, a dataflow graph is generated. The approach detailed in [11] parallelizes the resulting dataflow operations by mapping them to a network of spatial floating-point operators. On the other hand, the approach followed in [12] maps the ensuing dataflow graph to a multi-PE shared-memory system. It is clear that both approaches primarily focus on extracting the fine-grained dataflow parallelism, whereas the approach we propose in this paper favors harnessing the medium-grained column parallelism without overlooking the finer-grained data operation parallelism. In effect, our approach relies on constructing a column dependency graph, which we use to parallelize column operations, rather than dataflow operations. Nevertheless, our approach also maximizes the fine-grained parallelism potential as it implicitly translates column operations into dataflow graphs, which we use to control the deep pipelines of our processing elements.

The structure of this paper is as follows: Section 2 describes previous work that has been done to factorize sparse matrices into LU form on parallel hardware, in particular the Gilbert-Peierls algorithm. In section 3, we show how this algorithm can be expressed as a scheduling algorithm. We describe how we implemented this approach on an FPGA in section 4. In section 5, we describe our experimental setup, and give our results in section 6. We conclude in section 7.

2 Parallelizing Sparse LU Factorization

One of the most important aspects of designing any parallel algorithm is identifying the appropriate level of granularity, which can be then adequately mapped to the targeted processing architecture [13]. For instance, fine-grain parallelism (i.e. at the level of individual floating point operations) is available in either dense or sparse linear systems. It can be exploited effectively by using a stream-like processing architecture, such as a vector processor or a systolic array. Medium-grain parallelism arises from the fact that many column operations can be computed concurrently across a number of processing elements. An elimination tree-like graph can be used to characterize this type of parallelism, such that columns at the same graph level can be evaluated in parallel. This level of granularity is an extremely important source of parallelism for sparse matrix factorization, as sparsity increases the number of columns that can be operated on in parallel. This may, however, cause a load imbalance in the case where an entire column operation only requires a few floating point operations.

Large-grain parallelism for sparse matrices can be also identified by means of a tree-like elimination graph. Therefore, if T_i and T_j are disjoint sections of the elimination graph, then all of the columns corresponding to nodes in T_i can be computed completely independently of the columns corresponding to nodes in T_j, and vice versa. Thus, these computations can be done concurrently on separate processing elements with no communication between them. Roughly speaking, sparsity and parallelism are largely compatible, since the large-grain parallelism is due to sparsity in the first place. As such, an ordering that increases sparsity can also increase the potential parallelism.

Many parallel sparse system solvers employ a technique called the "the multifrontal scheme" [14] to parallelize computations by rewriting the original problem into a collection of "frontal matrices" . In effect, multifrontal solvers [15, 16] rely on a Directed Acyclic Graph (DAG) to extract and organize the parallel work. Each node (i.e. frontal matrix) of the DAG represents a given computation. All leaf nodes of the DAG (i.e nodes without an offspring) can be evaluated in parallel, while internal nodes can only be computed once their children have been computed. This method involves relatively significant amounts of data exchange between the tree nodes, requiring a considerable communication bandwidth. Therefore, multifrontal solvers work best in shared memory environments.

Another approach to parallel sparse solvers revolves around evaluating many pivots in parallel [17, 18]. These

solvers typically concentrate on the medium and fine grain parallelism, and tend to be most efficient on a moderate number of processors with fairly tight synchronization [19]. An important part of any sparse solver is the algorithm controlling the number of fill-ins that are generated during the solution process. Another aspect of pivot selection is the maintenance of stability. Typically, this is done by choosing a pivot element that is within a specified multiple of the largest element in the pivot row or pivot column or the active part of the matrix, depending on the efficiency of these tests and given the assumed data structures.

The stability and sparsity requirements for pivot selection are often contradictory and most strategies involve compromise. Selecting pivots for parallelism adds a third constraint. For the medium and fine grained algorithms mentioned above, these three constraints can be considered in a reasonably straightforward way, potentially with respect to the entire active portion of the matrix. The exploitation of larger grained parallelism, however, often imposes a static decomposition on the structure of the matrix which further constrains pivot selection. The effect of these constraints, for asymmetric problems, can be seen by considering tearing techniques or nested bisection. These techniques have been proposed to expose large-grain structures, suitable for parallel execution, by reordering the matrix into a form such as the Bordered Diagonal Block (BDB) form [20].

2.1 Gilbert-Peierls' Algorithm

The aim of a sparse LU algorithm is to solve the linear system $Ax = b$ in time and space proportional to $\mathcal{O}(n) + \mathcal{O}(nnz)$, for a matrix A of order n with nnz nonzeros [21]. In practice, this is much harder to achieve as the underlying non-zero structure of the matrix may change dramatically in the course of factorization. To tackle this issue, *Gilbert and Peierls* [22] proposed a left-looking sparse LU algorithm that achieves an LU decomposition with partial pivoting, in a time proportional to the number of floating-point operations, i.e. $\mathcal{O}(flops(LU))$. It is called a left-looking algorithm because it computes the k^{th} column of L and U only by using the already computed columns 1 to $(k-1)$.

Algorithm 1 Gilbert-Peierls LU factorization of a n-by-n asymmetric matrix A

1: $L = I$
2: **for** $k = 1$ to n **do**
3: $b = (A : k)$
4: solve the lower triangular system $L_k x = b$
5: do partial pivoting on x
6: $U(1 : k, k) = x(1 : k)$
7: $L(k : n, k) = x(k : n)/U(k, k)$
8: **end for**

The core of the Gilbert-Peierls factorization algorithm is solving a lower triangular system $Lx = b$, where L is a sparse lower triangular matrix, x and b are sparse vectors [23]. It consists of a symbolic step to determine the non-zero pattern of x and a numerical step to compute the values of x. This lower triangular solution is repeated n times during the entire factorization (where n is the size of the matrix) and each solution step computes a column of the L and U matrices. The entire left-looking algorithm is described in Algorithm 1. The lower triangular solution (i.e. line 4) is the most expensive portion of the Gilbert-Peierls algorithm and includes a symbolic and a numeric factorization step.

2.1.1 Symbolic Analysis

The Gilbert-Peierls Algorithm revolves around the efficient solution of $L_k x = b$ to compute the k^{th} column, where L_k is a unit diagonal representing the already computed $(k-1)$ columns, and where the column vector b is sparse. We create a list $\mathcal{X}$ of j indices for which we know x_j will a be non-zero, $\mathcal{X} = \{j \mid x_j \neq 0\}$, in ascending order. This gives a computation time of $\mathcal{O}(f)$ for Algorithm 2, where f is the number of floating-point operations required to solve the underlying triangular system.

Algorithm 2 Sparse forward substitution

1: $x = b$
2: **for** *each* $j \in \mathcal{X}$ **do**
3: **for** *each* $i > j$ for which $l_{ij} \neq 0$ **do**
4: $x_i = x_i - l_{ij} x_j$
5: **end for**
6: **end for**

Symbolic analysis is the process whereby the set $\mathcal{X}$ is defined. From the pseudo code in Algorithm 2, it can be seen that entries in x can become non-zero in only two places, namely, lines 1 and 4. If numerical cancellation is ignored, these two statements can be written as two logical implications, Equations (1) and (2), respectively.

$$line\ 1 : [b_i \neq 0 \implies x_i \neq 0] \quad (1)$$

$$line\ 4 : [x_j \neq 0 \wedge \exists i(l_{ij} \neq 0) \implies x_i \neq 0] \quad (2)$$

These two implications can be expressed as a graph traversal problem. Let G_{L_k} be the directed graph of L_k such that $G_{L_k} = (V, E)$ with nodes $V = \{1 \ldots n\}$ and edges $E = \{(j, i) \mid l_{ij} \neq 0\}$. Thus, Equation (1) is equivalent to marking all the nodes of G_{L_k} that are non-zeros in the vector b, whereas Equation (2) implies that if a node j is marked and it has an edge to a node i, then the latter must be also marked. Figure 1 graphically highlights these two relationships.

Therefore, if we have a set $\mathcal{B} = \{i \mid b_i \neq 0\}$ that denotes the non-zeros of b, the non-zero pattern $\mathcal{X}$ can be computed by determining the vertices that are reachable from the vertices of the set $\mathcal{B}$ i.e. $\mathcal{X} = Reach_{G_L}(\mathcal{B})$. The reachability problem can be solved using a classical depth-first search in G_{L_k} from the vertices of the set $\mathcal{B}$. The depth-first search takes a time proportional to

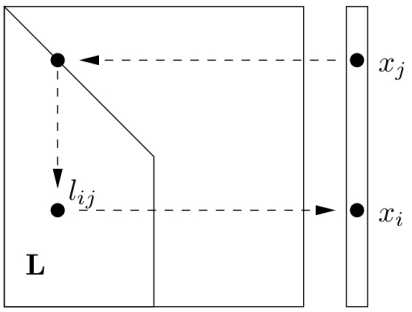


Fig. 1: Nonzero pattern for a sparse triangular solve

the number of vertices examined plus the number of edges traversed. The depth-first search does not sort the set $\mathcal{X}$, however, it computes its topological order. This topological ordering is useful to maintain the precedence relationship in the elimination process of the numerical factorization step. The computation of $\mathcal{X}$ and x both take times proportional to their floating-point operation counts [24].

2.1.2 Numerical factorization

Normally, this step consists of numerically performing the sparse triangular solution for each column k of L and U in the increasing order of the row index. The non-zero pattern computed by the symbolic analysis is, however, in a topological order. Sorting the indices would increase the time needed for the solutions. Nevertheless, the topological order is sufficient as it gives the order in which elements of the current column are dependent on each other. For instance, the depth first search would have finished traversing vertex i before it finishes traversing vertices j. Therefore, in the topological order j would appear before i.

2.1.3 Symmetric Pruning

Symmetric pruning is a technique whereby structural symmetry in matrices is exploited to reduce the time taken by the symbolic analysis [25]. The basic idea of the technique revolves around decreasing the time taken by the depth-first search by pruning unnecessary edges in the graph of a matrix (i.e. G). In effect, G can be replaced by a reduced graph, H, that has fewer edges but preserves the path structure. In fact, any graph H can be used in lieu of G if it preserves the paths between vertices of the original graph. In other words, if an edge $i \rightarrow j$ exists in G, it should also exist in H. In our work we use symmetric pruning to speed up the depth-first search in the symbolic factorization stage of the Gilbert-Peierls Algorithm.

3 Dependency-Aware Matrix Operations Scheduling

In this section, we explain one of the main contributions of this paper, which revolves around the construction of a deterministic and accurate task model for parallel LU factorization. The scheduling algorithm leverages the graph representation of a matrix, computed using symbolic factorization, to create an operation schedule that takes into account column-level dependencies. The generated static schedule can then be used to parallelize and control the dataflow of LU matrix operations on the FPGA. The main steps of the algorithm are as follows:

1. Pre-order matrix A to minimize fill-in (e.g. Approximate Minimum Degree (AMD)) and to ensure a zero-free diagonal (e.g. maximum traversal).
2. Perform symbolic factorization and determine the structure of the lower triangular matrix L and upper triangular matrix U.
3. Determine column dependencies using the structure of upper triangular matrix U.
4. Build a Directed Acyclic Graph (DAG) that represents the computed column-level dependencies.
5. Annotate nodes of the Column-Dependency DAG (CD-DAG) with their corresponding level of parallelism.
6. Derive the ASAP (As Soon As Possible) schedule for the required column operations.
7. Refine the ASAP schedule using *modulo i scheduling*, where i is the maximum number of columns that can reside at any level of the CD-DAG.

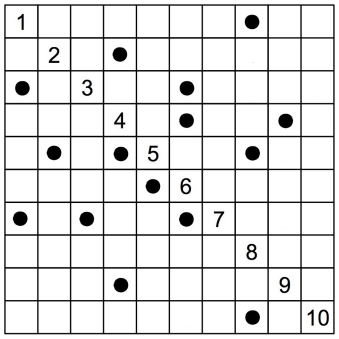


Fig. 2: Matrix A with an asymmetric nonzero pattern

To illustrate how our algorithm works, consider the matrix A shown in Figure 2. For the sake of simplicity, it is assumed that the matrix has a zero-free diagonal and it has been already pre-ordered with some fill-in minimizing heuristic. First of all, we need to carry out the Gilbert-Peierls factorization symbolically, using the principles given in Section 2.1.

For instance, to compute the nonzero pattern of column 2, we need to construct the graph of the lower components of columns to its left (i.e. Column 1 in Figure 3). The columns required at any step of the factorization process are represented by the shaded portion of the matrix in all subsequent figures of this section. The resulting graph is shown on the right-hand side of Figure 3. In column 2, there are two nonzeros at indices $\{2, 5\}$. Therefore, $Reach(2) = \{2\}$, $Reach(5) = \{5\}$ and hence $Reach(2, 5) = \{2, 5\}$. We can see that the reachability function has returned the input set itself. This

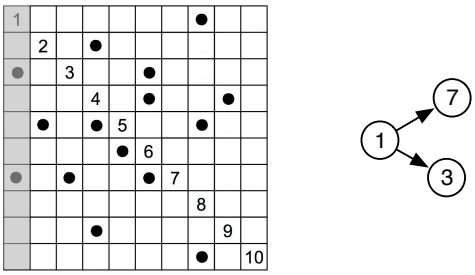


Fig. 3: Symbolic Gilbert-Peierls factorization example: step 1.

implies that column 2 structure remains unchanged and it will not suffer from any fill-in during the numerical factorization process. The structures of columns 3, 4, 5 also remain unchanged.

Starting from column 6, however, we start to see the impact of fill-ins on the nonzero structure of the matrix. As in the previous steps, we first need to construct the graph of the lower components of columns to the left of column 6. We then perform a depth-search (i.e. the reachability test) using the nonzero structure of column 6. As can be seen in Figure 4, column 6 has four nonzero elements at indices {3, 4, 6, 7}. The new nonzero pattern of column 6, including fill-ins, is given by Equations 3-5:

$$\begin{aligned} Reach(3,4,6,7) &= Reach(3) \cup Reach(4) \\ &\quad \cup Reach(6) \cup Reach(7) && (3)\\ &= \{3,7\} \cup \{4,5,6,9\} \cup \{6\} \cup \{7\} && (4)\\ &= \{3,7,4,5,6,9\} && (5)\\ Fillin(Col_6) &= Reach(3,4,6,7) - \{3,4,6,7\} && (6)\\ &= \{3,7,4,5,6,9\} - \{3,4,6,7\} && (7)\\ &= \{5,9\} && (8) \end{aligned}$$

From Equations 6-8, on the other hand, we can see that we can also expect the appearance of two fill-in elements at indices $\{5, 9\}$ in the new nonzero structure of the column 6. $Fillin(Col_k)$ is a function that returns the row indices of the new fill-ins in column k. Figure 5 shows the remaining steps of the symbolic factorization.

Now that we have computed the non-zero pattern of resulting LU factors, we need to determine the column dependencies that may arise during the numerical factorization process. In the Gilbert-Peierls algorithm, the flow of computation follows two steps, which are repeated sequentially until the entire matrix is processed. The first step is *"the sparse triangular solution"*, in which the elements of the current column are factorized by solving $L_k x = b$ for x, where L_k represents the triangular matrix of leftmost columns factorized so far, b is the current column to be decomposed, and x is the decomposed column. In the next step, the computed column is normalized by dividing all its lower off-diagonal elements by the pivot. As the column normalization operation is self-contained (i.e. does not require any other column), it is clear that any column dependencies in the overall Gilbert-Peierls algorithm only arise from the underlying dependencies in the *"the sparse triangular solution"* step. However, when computing a column k using the sparse triangular solution algorithm, not all the columns to its left are needed, as it was illustrated in Section 2.1.1. In effect, the factorization of column k only depends on the columns that satisfy the following criterion:

$$Dependency(Col_k) = \{j | a_{jk} \neq 0, j < k\} \qquad (9)$$

***Definition** 1:* We define a Directed Acyclic Graph (DAG) such that if column k depends on column i, then a directed edge exist from node i to node k (i.e. $i \rightarrow k$) where $i < k$.

***Definition** 2:* We define the following type of nodes. A "leaf node" is a node that has no incoming edges. In contract, a "parent node" is a node that has incoming edges. If a parent node has no outgoing edges, it is then called a "a root node". An "orphan node" is a node that has no incoming or outgoing edges.

***Definition** 3:* We define the level of each node as the length of the longest critical path from any "leaf node" to the node itself. In our implementation of the scheduling algorithm, we use *Liao and Wong's* algorithm [26] to find the longest path.

A parent node cannot be eliminated unless all its children have been processed. Two columns are said to be independent if they belong to two different subgraphs/trees. Moreover, all nodes at the same level can be evaluated in parallel.

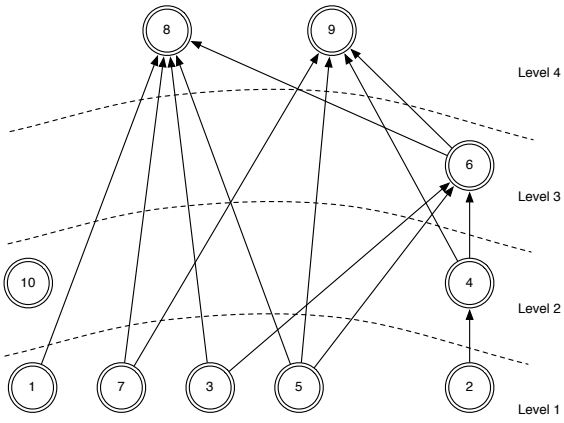


Fig. 6: Unconstrained Schedule Graph for Matrix A.

Figure 6 illustrates the column dependencies that will arise during the LU factorization of our example

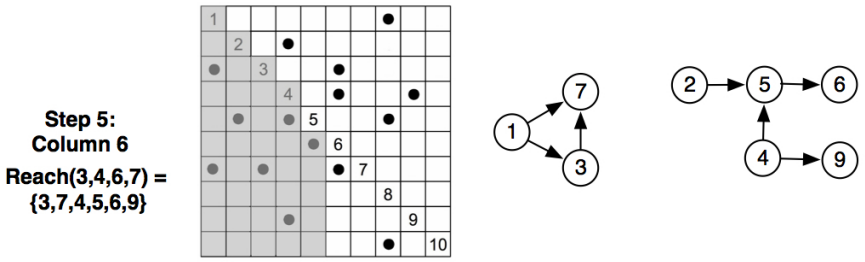


Fig. 4: Symbolic Gilbert-Peierls factorization example: step 5.

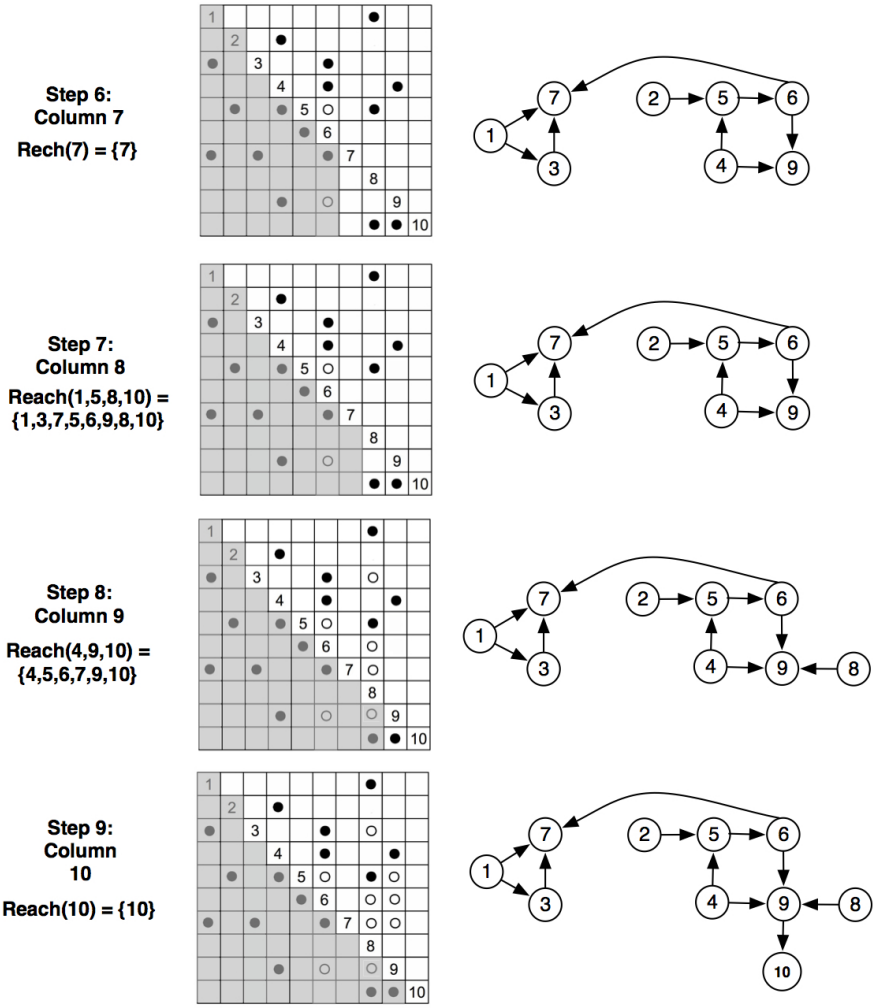


Fig. 5: Symbolic Gilbert-Peierls factorization example: step 6 - step 9.

matrix, A. The graph was computed using the predicted non-zero structure of matrix U only. All the nodes at same level can be computed independently. For instance, columns 1, 2, 3, 4, 5, 7 can be evaluated in parallel, however, column 9 cannot be processed until columns 4, 5, 6 are computed. Column 10 is represented by an orphan node, which implies that it can be placed at any given level. Generally speaking, the sparser the matrix, the fewer dependencies (i.e. fewer non-zeros) there are, and hence the node count per level also increases. Thus, pre-ordering a matrix for sparsity can dramatically increase the parallelism potential. Figure 7 visually illustrates that that pre-ordering a given matrix using the AMD heuristic produces much sparser LU factors, that is, 70% sparser for the matrix depicted. This also has the effect of reducing the overall number of FLOPs required to perform the sparse LU factorization.

Although our scheduling algorithm efficiently derives a list of columns that can be evaluated in parallel within a given time-slot, it assumes that the same time is taken to compute each column. In reality, however, columns have different non-zero structures and thus the number of floating-point operations per column will also differ, ultimately impacting the column computation

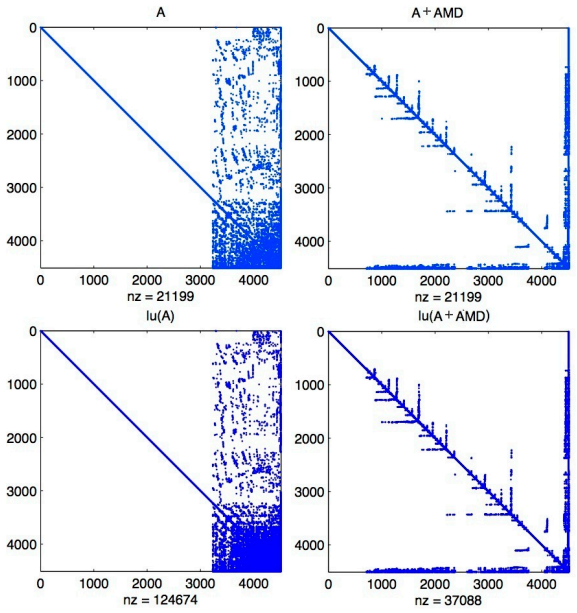


Fig. 7: Impact of the AMD ordering on Sparse LU Factorization

time. Nonetheless, pre-ordering matrices for sparsity not only reduces the overall FLOP count but also distributes the computational effort more evenly across the columns of a given matrix. Figure 8 illustrates the impact of matrix ordering on the column FLOP count associated with the Gilbert-Peierls factorization. (The properties of the test matrices used are listed in Table 2, Section 6.) The number of FLOPs associated with a given column index was acquired by profiling a purpose-written MATLAB script that performs left-looking LU decomposition with no pivoting. All the input matrices were initially permuted using maximum traversal to ensure a zero-free diagonal, and then ordered using the AMD algorithm. The script also accounts for numerical cancellations that may occur during the factorization process. These cancellations, even though very rare, lead to the appearance of zeros on the diagonal and ultimately halt the factorization algorithm during the normalization phase. We can see that AMD-ordered matrices produce much sparser LU factors, and also generate a more balanced workload across the columns. In effect, from Figure 8(a) and Figure 8(b), we can clearly see that the bulk of the FLOPs were concentrated around the left-hand side of the graph. Once the AMD ordering was applied, we see that the FLOP count is more evenly shared across the column indices. Furthermore, a lower FLOP count per column reduces the amount of resources required to compute a given column in parallel. For instance, in Figure 8(a), the highest column FLOP count recorded has dropped from almost 250,000 FLOPs to under 300 FLOPs after the AMD ordering. This is particularly attractive in a distributed computing architecture, where the columns are spread over many processing elements.

Assuming it takes roughly the same time to compute all the columns, the schedule is equivalent to the unconstrained As Soon As Possible (ASAP) schedule for the

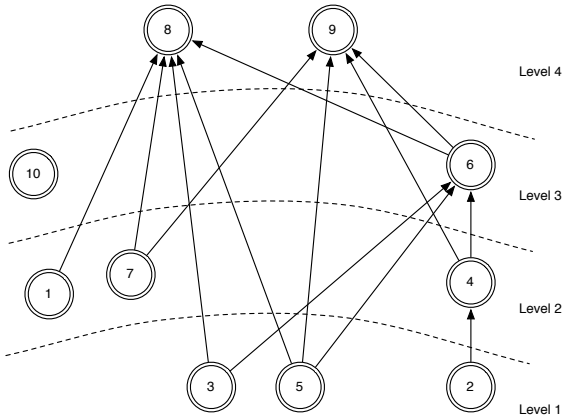


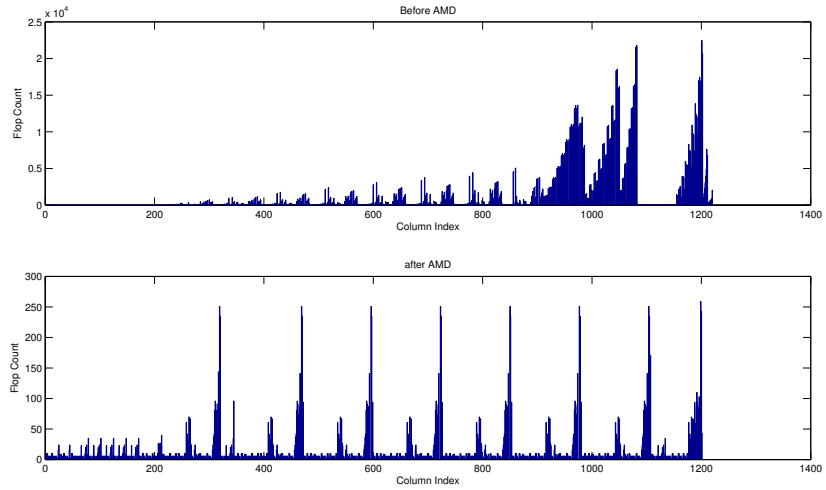
Fig. 9: Schedule Graph for Matrix A with *modulo 3*.

LU column operations [27]. The ASAP schedule unrealistically assumes that there will always be enough computational resources to concurrently process all columns within the same level. Therefore, in our algorithm, we introduce a resource-constrained scheduling algorithm we refer to as *"modulo i scheduling"*, where i refers to maximum number of nodes within any given level. For instance, a *modulo 3* schedule assumes that there are only 3 computational units, each capable of independently processing a column, and thus it limits the number of nodes per level to a maximum of 3. Figure 9 defines *"the modulo 3 schedule"* derived from the unconstrained graph depicted in Figure 6. *"Modulo i scheduling"* is particularly attractive if it is mapped to a pipelined FPGA architecture, where area is traded for latency.

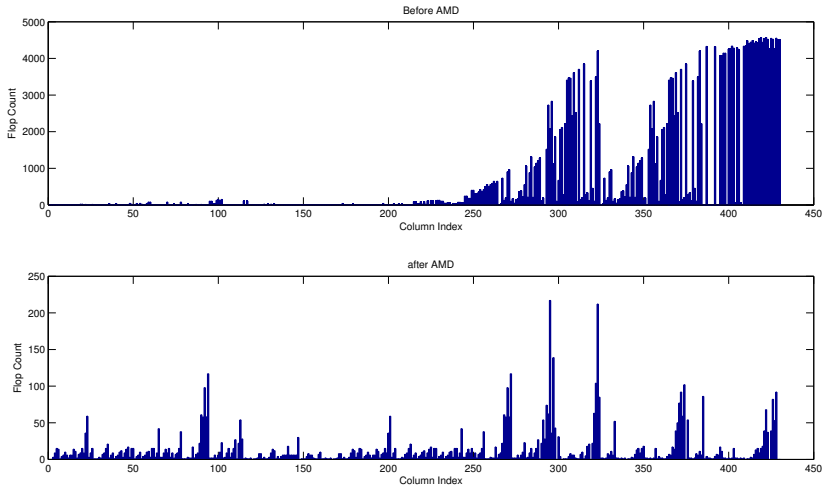
4 Parallel Sparse LU FPGA Architecture

In Section 3, we demonstrated that that the seemingly sequential flow of the Gilbert-Peierls LU factorization algorithm can be effectively parallelized by explicitly exposing column-level concurrency, by means of a scheduling graph. This graph only depends on the nonzero structure of the circuit matrix. The nonzero pattern of a circuit matrix reflects the couplings and the connections that exist in the underlying circuit, which do not change during the course of a SPICE simulation. This means that the matrix to be solved retains the same nonzero pattern over the SPICE transient iterations, and it only has changes in numerical values. Hence, the symbolic analysis cost is justifiable and is amortized over a number of iterations. Therefore, the column-level dependency graph can be cheaply computed offline (see Section 6.1) before the actual numerical factorization takes place on the FPGA accelerator.

The column-level dependency graph can be then loaded onto the FPGA and used to dictate a parallel execution flow of LU column operations. However, it may not be possible to fit the entire graph for a large matrix onto the FPGA, in which case, the column-dependency information can be also used to pre-compute a column



(a) fpga_dcop_01



(b) oscil_dcop_01

Fig. 8: The Effect of Matrix Ordering on the LU Factorization Column FLOP Count.

loading order. The latter can be then used to dynamically load columns to the FPGA such that computations and memory loads are overlapped, effectively hiding the latency associated with the external memory interface. To illustrate this concept, consider the scheduling graph shown in Figure 9 as an example. Here, columns 1, 2, 3, 4, 5, 7 can be loaded to the FPGA first. In the second stage, columns 6, 10 can be loaded in lieu of column 2, 3, 5 while columns 1, 4, 7 are being normalized. In the last stage, columns 8, 9 are loaded to replace columns 1, 7 while columns 6, 10 are being normalized. For the examples presented below, however, the built-in RAM was large enough and we did not implement dynamic loading.

4.1 Resolving Dataflow Dependencies

So far, we have established that the Gilbert-Peierls sequential column factorization process can be altered to expose column-level parallelism. Despite this exposed column-evaluation concurrency, dataflow dependencies may still exist within column-level updates themselves. In order to illustrate this, consider Figure 10, in which we show all the dataflow dependencies and operations needed to compute the LU factorization of the example matrix A, depicted in Figure 2, according to its unconstrained schedule. We note two types of dataflow dependencies: inter-column and intra-column data dependencies.

The inter-column data dependencies represent the inherent column-level dependencies that exist in the Gilbert-Peierls algorithm. This type of dependency can be naturally resolved by simply following the execution order determined by the corresponding schedule, factorizing columns in level 1 first, then columns in level 2, and so forth. The intra-column dependencies relate to the order at which the current column ele-

ment updates, in the sparse triangular solution, should be calculated. Nevertheless, in Section 2.1.1, we have established that Gilbert-Peierls' symbolic analysis of a particular column effectively computes a topological order that maintains the precedence relationship in the numerical factorization step. In effect, this computed topological order can be used to sustain a dataflow stream to the pipelined floating-point operations on the FPGA. Studying the dataflow graph more closely, we can also see that division operations associated with the column normalization stage (e.g. columns 1, 2, 3, and 5) can be performed concurrently, creating another source of parallelism that can be exploited at the hardware level.

4.2 Design Flow

Our work implements the Gilbert-Peierls LU factorization in conjunction with the static pivoting algorithm introduced by *Li and Demmel* in [28], which they showed to be as accurate as partial pivoting algorithms for a number of problems including circuit simulations. The main advantage of static pivoting is that it permits a priori optimization of static data structures and the communication pattern, effectively decoupling the symbolic and numerical factorization steps. This makes sparse LU factorization more scalable on a distributed memory architecture. The overall implemented algorithm can be summarized as follows:

1 First, we find diagonal matrices D_r, D_c and a row permutation P_r such that $P_r D_r A D_c$ is more diagonally dominant to decrease the probability of encountering small pivots during the LU factorization. To achieve this, we use the HSL MC64 routine, [29], with option 4. The latter computes a permutation of the matrix so that the sum of the diagonal entries of the permuted matrix is maximized.

2 We find a permutation P_c such that the resulting matrix in step (1) incurs less fill-in in the course of the LU factorization. We can use many heuristics such as nested dissection or minimum degree on the graph of $A + A^T$ or AA^T. However, we shall use the approximate minimum degree (AMD) as it produces the best results for circuit matrices. In order to preserve the diagonal computed in step (1), any ordering used should be applied symmetrically.

3 We perform symbolic analysis to identify the non-zero entries of L and U. We also compute the associated task-flow graph by performing a symbolic LU decomposition, i.e. using the predicted non-zero structure.

4 In this step, we perform left-looking LU factorization on the FPGA and replace any tiny pivots (i.e. $|a_{ii}| < \sqrt{\varepsilon}.\|A\|$) by $\sqrt{\varepsilon}.\|A\|$, where ε is the machine precision (e.g. 2^{-24}, 2^{-53} for single and double precision IEEE 754 formats, respectively), and $\|A\|$ is the matrix norm. This is acceptable in practical terms as the SPICE linear system solution is used as part of Newton-Raphson iteration, and an occasional small error during the iterative process does not affect the integrity of the final solution [28]. We calculate the matrix norm at the symbolic factorization phase, using the SuiteSparse API [30]. The use of the HSL MC64 routine in step (1) decreases the likelihood of encountering tiny pivots. Furthermore, selecting the diagonal as the pivot entry ensures the fill-reducing ordering from the symbolic phase is maintained.

Steps 1 to 3 form the *"matrix preconditioning phase"*, and they are conducted as part of our scheduling algorithm implementation. The scheduler takes a sparse matrix as input, applies the AMD ordering, and then symbolically generates the column-level dependencies as well as the nonzero pattern of the LU factors. For step 4, we implement the parallelized version of the Gilbert-Peierls factorization algorithm on the FPGA, using a multi-PE distributed architecture. Since we do not consider dynamic pivoting in our design, all possible fill-ins as well as column and dataflow dependencies are determined at the matrix preconditioning phase.

4.3 Top Level Design

Our parallel FPGA architecture features multiple PEs interconnected by a switch network. Figure 11 shows the top level diagram of the our sparse LU hardware implementation. Essentially, our design consists of a controller connected to n PEs. In each PE, there is a multiplier, a subtractor, a divider, and a local Block Random Access Memory (BRAM) with a reconfigurable datapath.

The maximum number of PEs, and their local memory size are limited by the available resources of the FPGA. We use the information gathered from symbolic analysis to instantiate PEs accordingly. The PEs are interconnected by high speed switches to minimize the communication overhead while increasing concurrency.

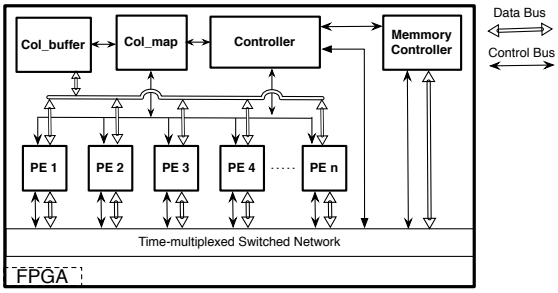


Fig. 11: Top Level Design for the LU Decomposition FPGA Hardware

The controller implements a four stage pipeline. Stage 1 consists in loading the matrix data from the off-chip DRAM to the PEs on-chip BRAM. The PEs' local BRAMs can be also preloaded with matrix data at the FPGA programming phase such that the matrix data is included in the "bitstream". Stage 2 performs a triangular sparse solve on the current column of A to compute the current columns of L and U. Stage 3 normalizes the component of L with the diagonal entry. Stages 2 and 3

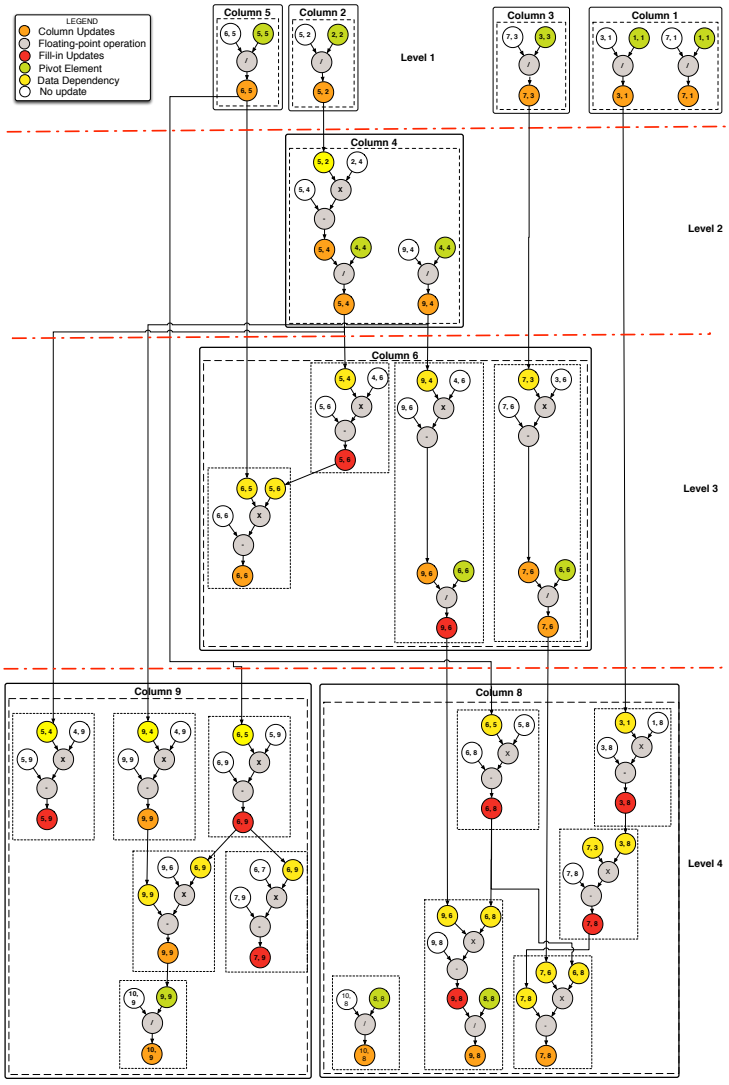


Fig. 10: Dataflow of a Gilbert-Peierls LU factorization. Columns 7 and 10 do not appear because they do not need to be processed.

are executed iteratively until all columns are evaluated. At any given time, PEs collectively perform either the sparse triangular solve or the column normalization.

In the sparse solve phase, the "Col_map" unit first performs a burst read across all PEs to form a column-wise representation of the pivot column and saves it to the column buffer. Then, elements of the column buffer are broadcast to the PEs one at a time to perform the bulk computation of the sparse triangular solution. In the normalization phase, the controller fetches the pivot entry from its corresponding PE and broadcasts it to all PEs to perform all the divisions in parallel. To fill the deep pipelines of our floating-point units, the controller uses the column-dependency graph as a task flow-graph. Data are streamed from the memory, through the arithmetic units for computation, and stored back to the memory in each stage.

5 Experimental Setup

In this section, we explain the experimental setup used to build and test our LU decomposition FPGA hardware prototype. To implement a prototype for our design, we targeted the Xilinx XUPV5-LX110T development board, which features a Virtex 5 LX110T FPGA. As mentioned in Section 4.3, the controller of our design utilises the column-dependency graph of a matrix as a task flow-graph to stream data from the memory, through the arithmetic units for computation, and stores the results back to the memory in each stage. As such, the relative placement between the memory blocks and the computational blocks is important and can significantly impact performance. The Virtex-5 FPGA benefits from the physical proximity of these blocks as they are arranged close to each other in special lanes within the fabric (i.e. BRAM and DSP48 blocks).

Therefore, in our implementation, we used the floating-point subtract, multiply/divide (DSP48 blocks), and compare units from the Xilinx Floating-Point library. The latter is readily available from Xilinx's CoreGen [31]. These units can be customized with regards to their wordlength, latency, and resource utilization. We also use Xilinx's FIFO Generator to implement

the "Col_buffer", which works in concert with the "Col_Map" unit. We used Synplify Pro 9 and Xilinx ISE 10.1 to implement our prototype on a Xilinx Virtex-5 LX100T FPGA. We limited our implementations to fit on a single FPGA and use off-chip DRAM memory resources for storing the matrix data before it is loaded onto the on-chip BRAM for processing.

Table 1 gives the resource cost for the different blocks present in our design. We can only fit a system of 8 double-precision PEs on a Virtex-5 LX110T with 88% of logic resources being used, whereas 16 single-precision PEs can be easily accommodated. Although SPICE uses double-precision arithmetic, we found that single-precision was sufficient for the examples presented here.

TABLE 1: Sparse LU Hardware Prototype Resource Utilisation on a Virtex-5 LX110T

	% of 69120 LUTs		Latency		BRAM		DSP48		Clocks (MHz)	
Precision	SP*	DP**	SP	DP	SP	DP	SP	DP	SP	DP
Adder	245	734	11	14	0	0	2	3	410	355
Multiplier	89	309(1%)	8	16	0	0	3	11	493	410
Divider	769	3206(4%)	28	57	0	0	0	0	438	410
2 PEs	2822 (7%)	16%	-	-	10	18	6	22	150	150
4 PEs	6232 (14%)	40%	-	-	20	46	12	33	150	150
8 PEs	14493 (32%)	88%	-	-	40	-	24	-	150	150
16 PEs	(71%)	-	-	-	64	-	48	-	150	-

*Single-precision **Double-precision

6 Performance Analysis

In order to evaluate the performance of our hardware design, we tested our parallel architecture with circuit simulation matrices from the University of Florida Sparse Matrix Collection (UFMC). The performance measurements are then compared to the state-of-the-art UMFPACK, KLU, and Kundert sparse LU decomposition matrix packages. In our performance evaluation, we use the CPU time reported by UMFPACK 5.4, Kundert Sparse 1.3, and KLU 1.2 on a 64-bit Linux system running on a 6-core Intel Xeon 2.6 GHz processor with 6 GB RAM, as a benchmark.

- UMFPACK [32] implements a right-looking multifrontal algorithm tuned for asymmetric matrices that makes extensive use of BLAS kernels. In our tests, we used UMFPACK's default parameters.
- Kundert Sparse [33], implements a right-looking LU factorization algorithm that preforms dynamic pivoting on the active sub-matrix using the Markowitz ordering algorithm. It is also the sparse solver used in Spice3f5, the latest version of the open-source SPICE simulator.
- KLU [23] is an LU matrix solver written in C that employs the left-looking Gilbert-Peierls LU factorization algorithm. KLU has been written specifically to target circuit simulations.

To gauge the time taken by our FPGA-based LU decomposition architecture, we used Xilinx's ChipScope Integrated Logic Analyser (ILA) to count the number of clock cycles required to perform the LU decomposition. We used the same the pre-ordering (i.e. AMD) for LU matrix packages and our Sparse LU Hardware. Table 2 contains the relevant properties of the test matrices used and the corresponding LU decomposition runtimes reported by UMFPACK, KLU, and Kundert Sparse. Table 2 also shows the execution time of LU FPGA hardware as reported by ChipScope, and the FPGA acceleration achieved using 16 single-precision PEs running at 150 MHz. The acceleration is calculated as a ratio of the CPU time taken by a given LU matrix package over the time spent by the sparse LU hardware on the same circuit matrix.

For the test matrices used, we can see that our 16-PE LU hardware outperforms KLU, UMFPACK, and Kundert Sparse on average by factors of 9.65, 11.83, 17.21, respectively. Furthermore, we note a correlation between the matrix sparsity and speedup ratio of our design. We also remark that the best acceleration results were achieved when the matrix is very sparse and has a symmetric or near-symmetric pattern (e.g. Rajat13, add32, meg4). In effect, high sparsity implies that fewer column-level dependencies will exist during the course of Gilbert-Peierls LU factorization, and thus increases the parallelism potential, as shown in Section 3. On the other hand, higher structural symmetry implies a more balanced elimination graph, which translates into a more balanced workload which minimizes the idle time of the different PEs, leading to a busier computational pipeline.

6.1 Cost of the pre-processing stage

As we mentioned earlier, KLU and our FPGA design rely on information computed in the symbolic stage to speedup subsequent factorizations. In effect, during the symbolic stage, KLU performs a one-off partial pivoting numerical factorization to determine the nonzero structure of the LU factors. In the subsequent iterations, KLU reuses the previously-computed nonzero pattern to reduce the factorization runtimes. In our work, we use the pre-processing steps described in Section 3 to perform symbolic analysis and to compute the scheduling graph. The latter is used to parallelize the actual numerical factorization on the FPGA. Therefore, we demonstrate how the cost of this symbolic stage can be amortized over a number of SPICE iterations. Table 3 tabulates the CPU runtimes for the symbolic stage of KLU as well the time taken by our pre-processing stage. From the runtime figures, we note that our pre-processing stage is on average 20% faster than KLU's symbolic analysis stage. This reflects the fact that KLU performs a one-time numerical factorization during this stage, whereas in our symbolic analysis step we only rely on the graph representation of the underlying matrix. We can also see that the time taken by the KLU symbolic stage is on average 5.1× the KLU factorization runtime on a CPU. On the other hand, the time taken by our pre-processing stage is on average 36× the factorization time on the FPGA, but which is amortized over all iterations.

TABLE 2: LU decomposition hardware acceleration achieved versus UMFPACK, Kundert Sparse, and KLU

Matrix	Matrix properties					CPU runtimes for			FPGA		FPGA speedup[6] (×) achieved versus		
Name	Order	NNZ[1]	Sparsity (%)	Str Sym[2]	Num Sym[3]	UMFPACK (ms)	Kundert Sparse (ms)	KLU (ms)	Latency[4] (Cycles)	Time[5] (ms)	UMFPACK	Kundert Sparse	KLU
Rajat11	135	665	3.600	89.10%	63%	0.003	0.002	0.019	249	0.002	2.05	1.44	11.14
Rajat14	180	1,475	0.040	100%	2%	0.020	0.011	0.029	370	0.002	8.10	4.53	11.75
oscil_dcop_11	430	1,544	0.800	97.60%	69.80%	0.583	0.793	0.329	3,397	0.023	25.74	35.02	14.54
circuit204	1,020	5,883	5.600	43.80%	37.30%	0.243	0.909	0.482	9,103	0.061	4.00	14.97	7.94
Rajat04	1,041	8,725	0.800	100%	4%	0.035	0.021	0.033	1,049	0.007	5.00	2.97	4.72
Rajat19	1,157	3,699	0.298	91%	92%	0.217	0.333	0.202	3,047	0.020	10.68	16.41	9.96
fpga_dcop_50	1,220	5,892	0.400	81.80%	33.20%	1.093	1.200	0.685	9,960	0.066	16.47	18.07	10.31
fpga_trans_01	1,220	7,382	0.500	100%	21%	0.030	0.011	0.043	1,100	0.007	4.09	1.46	5.86
fpga_trans_02	1,220	7,382	0.500	100%	21%	0.032	0.010	0.051	1,007	0.007	4.76	1.56	7.59
fpga_dcop_01	1,813	5,892	0.179	65%	1.60%	0.547	1.087	0.511	7,055	0.047	11.62	23.11	10.87
init_adder1	1,813	11,156	0.300	65.40%	1.60%	0.567	1.035	0.480	5,479	0.037	15.52	28.33	13.13
adder_ dcop_57	1,813	11,246	0.300	64.80%	0.80%	0.464	1.464	0.363	7,981	0.053	8.71	27.51	6.82
adder_trans_01	1,814	14,579	0.440	100%	3%	0.024	0.044	0.039	1,221	0.008	2.95	5.40	4.79
adder_trans_02	1,814	14,579	0.440	100%	3%	0.023	0.048	0.041	1,116	0.007	3.09	6.45	5.51
Rajat12	1,879	12,818	0.360	100%	45%	0.119	0.121	0.118	2,023	0.013	8.82	8.97	8.77
Rajat02	1,960	11,187	0.300	100%	100%	1.034	1.028	0.921	17,866	0.119	8.68	8.63	7.74
add20	2,395	13,151	0.230	100%	53%	0.861	1.021	0.460	9,710	0.065	13.30	15.77	7.11
bomhof1	2,624	35,823	0.520	100%	21%	4.550	7.181	2.675	68,651	0.458	9.94	15.69	5.84
bomhof2	4,510	21,199	0.104	81%	41%	3.944	5.974	1.950	37,081	0.247	15.95	24.17	7.89
add32	4,960	19,848	0.080	100%	31%	1.740	3.088	1.412	13,320	0.089	19.59	34.77	15.90
meg4	5,860	25,258	0.070	100%	100%	0.723	0.923	0.514	3,694	0.025	29.35	37.48	20.85
hamrle2	5,952	22,162	0.600	0.10%	0%	0.693	2.075	0.551	16,670	0.111	6.23	18.67	4.96
Rajat01	6,833	43,520	0.093	99.60%	99%	1.910	1.981	1.181	10,219	0.068	28.04	29.08	17.34
Rajat13	7,598	48,762	0.080	100%	30%	1.941	3.150	1.014	15,126	0.101	19.25	31.24	10.05
Rajat03	7,602	32,653	0.060	100%	40%	1.096	2.113	0.935	20,405	0.136	8.06	15.53	6.87
Rajat06	10,922	46,983	0.040	100%	100%	1.096	1.246	0.972	10,344	0.069	15.89	18.06	14.10
bomhof3	12,127	48,137	0.300	77%	30%	5.428	7.764	3.306	60,266	0.402	13.51	19.33	8.23
Average	-	-	-	-	-	-	-	-	-	-	11.83	17.21	9.65

[1] Number of nonzero elements.
[2] Numerical Symmetry is the fraction of nonzeros matched by equal values in symmetric locations.
[3] Structural Symmetry is the fraction of nonzeros matched by nonzeros in symmetric locations.
[4] Number of the FPGA clock cycles taken to compute the LU factorization.
[5] Time taken to complete the LU factorization on an FPGA accelerator running at 150 MHz.
[6] Using 16 single-precision PEs running at 150 MHz.

TABLE 3: Cost of the symbolic analysis in KLU and our FPGA approach

	KLU		FPGA	
Matrix	Symbolic stage (ms)	LU (ms)	Symbolic stage (ms)*	LU (ms)**
Rajat11	0.081	0.019	0.089	0.002
Rajat14	0.103	0.029	0.124	0.002
oscil_dcop_11	1.542	0.329	1.314	0.023
circuit204	1.562	0.482	1.125	0.061
Rajat04	0.158	0.033	0.147	0.007
Rajat19	1.070	0.202	0.747	0.020
fpga_dcop_50	2.425	0.685	2.724	0.066
fpga_trans_01	0.209	0.043	0.170	0.007
fpga_trans_02	0.255	0.051	0.192	0.007
fpga_dcop_01	2.559	0.511	2.150	0.047
init_adder1	2.586	0.480	1.725	0.037
adder_ dcop_57	4.642	0.363	1.376	0.053
adder_trans_01	0.212	0.039	0.150	0.008
adder_trans_02	0.190	0.041	0.161	0.007
Rajat12	0.559	0.118	0.446	0.013
Rajat02	4.275	0.921	4.018	0.119
add20	2.332	0.460	1.937	0.065
bomhof1	16.193	2.675	9.979	0.458
bomhof2	9.685	1.950	7.881	0.247
add32	7.475	1.412	5.945	0.089
meg4	2.515	0.514	0.860	0.025
hamrle2	2.983	0.551	2.419	0.111
Rajat01	5.493	1.181	4.611	0.068
Rajat13	5.098	1.014	3.918	0.101
Rajat03	4.222	0.935	3.782	0.136
Rajat06	5.745	0.972	4.027	0.069
bomhof3	24.180	3.306	12.914	0.402

* Time taken to complete the LU factorization on an FPGA accelerator running at 150 MHz.
** Using 16 single-precision PEs running at 150 MHz.

6.2 Scalability

In order to study the scalability trends of our design, we gauge the performance of our design with 2, 4, 8, and 16 PEs configurations. We use the KLU runtimes, reported in Table 2, as a benchmark to calculate the speedups achieved per design configuration. The FPGA LU factorization runtimes per PE count and their corresponding speedups are illustrated in Figure 12. In most cases, we can see that the acceleration grows almost linearly with the number of PEs, with an average 60% acceleration boost as we double the PE count. The exception to this observation are the following matrices: init_adder1, add20, and Rajat13. In effect, the maximum achievable speedup for any matrix depends on the number of columns that can be processed at once. This is determined by the number of nodes that can reside at a given level of the ASAP schedule as well as the number of PEs at our disposal. For instance, if a given matrix A has 100 ASAP schedule levels and 90 of those have 8 nodes, using 8 processors would deliver an optimal speedup-to-PE ratio. Increasing the number of PEs beyond 8 could result into a marginal speedup increase only if at least one of the remaining 10 levels has a node count higher than 8. Therefore, if we draw the ASAP schedule for the three matrices, we will most likely find that the majority of the their schedule levels are slightly higher than 8. Hence, the speedup ratios for these matrices do not scale as well when we increase the number of PEs from 8 to 16.

The acceleration potential of our design can be further improved by increasing the frequency of the overall design clock. Table 4 shows the resource utilization of our design if a Virtex-7 XC7V200T were used. As we can see, synthesis results indicate that the overall design frequency has increased from 150 MHz to 250MHz. This is mainly due to customizing the CoreGen floating-point

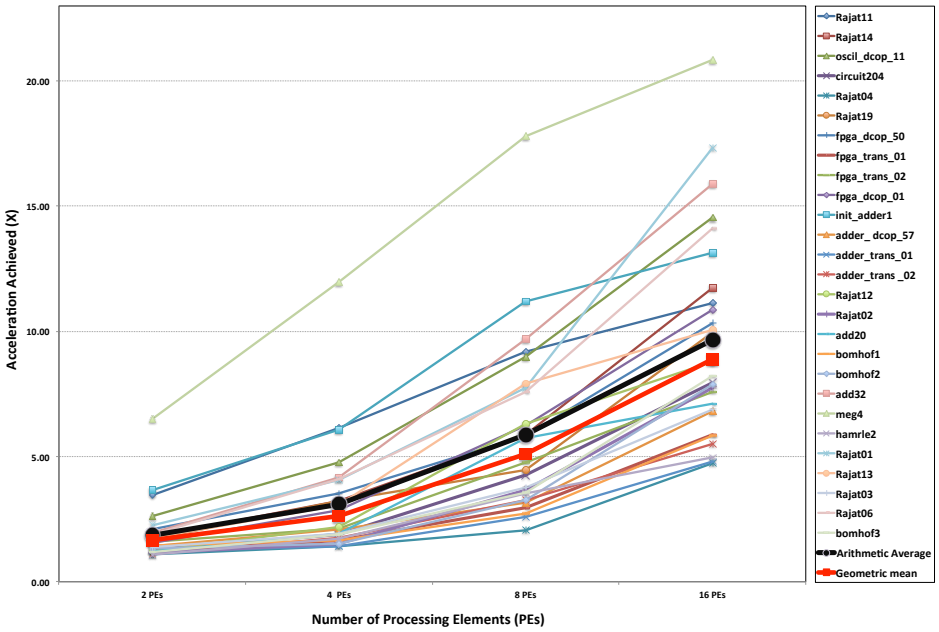


Fig. 12: Sparse LU FPGA acceleration scaling trends in terms of PEs

divider latency to 1 clock cycle as compared to 28 cycles for the same operator on the Virtex 5. This higher overall frequency indicates that we can now expect that our acceleration ratios on the Virtex 7 to increase at same rate (i.e 1.6×), as illustrated by Equation 11. In other words, changing the target FPGA from Virtex 5 to Virtex 7 improves the average 16 PEs speedup ratio from 9.65× to 15.44× (i.e 9.65 ×1.6). The overall predicted speedup that can be achieved by using a 32-PE configuration on the more modern Virtex 7 is shown in Equation 12.

$$Speedup^{32PEs} = (1.6) \cdot Speedup^{16PEs} \quad (10)$$

$$Speedup^{32PEs}_{virtex7} = Speedup^{32PEs}_{virtex5} \cdot \frac{freq_{virtex7}}{freq_{virtex5}} \quad (11)$$

$$= (1.6) \cdot Speedup^{16PEs}_{virtex5} \cdot \frac{freq_{virtex7}}{freq_{virtex5}} \quad (12)$$

TABLE 4: Sparse LU Hardware Prototype Resource Utilization on a Virtex-7 XC7V200T

	Usage of 1,954,560 LUTs		Latency		BRAM		DSP48		Clocks (MHz)	
Precision	SP*	DP**	SP	DP	SP	DP	SP	DP	SP	DP
Adder	407	794	8	8	0	0	0	0	472	436
Multiplier	103	279	6	16	0	0	3	11	463	403
Divider	1,106	3,412	1	1	0	0	0	0	482	375
1 PEs	4,931	16,080	-	-	5	10	3	11	250	250
16 PEs	17,2121 (8%)	590,576 (30%)	-	-	64	136	48	176	250	250
32 PEs	467,342 (24%)	1,456,950 (74%)	-	-	142	283	96	352	250	250

*Single-precision **Double-precision

7 Conclusions

In this paper, we have demonstrated an FPGA implementation of Sparse LU factorization, a key computational kernel of the SPICE matrix solution phase, that harnesses the parallelism of circuit matrices exposed at the pre-processing stage using specialized techniques. Using benchmark matrices from the UFMC repository, we empirically demonstrated that our 16-PE LU Virtex 5 implementation outperforms modern LU matrix software packages, running on a 6-core 12-thread Intel Xeon X5650 microprocessor, by many times. In effect, we showed that our LU FPGA implementation is on average 9.65×, 11.83×, 17.21× faster than KLU, UMFPACK, and Kundert Sparse matrix packages respectively.

References

[1] I. Naumann and H. Dirks, "Efficient reordering for direct methods in analog circuit simulation," *Electrical Engineering (Archiv fur Elektrotechnik)*, vol. 89, no. 4, pp. 333–337, 2007.

[2] A. Heinecke and M. Bader, "Towards many-core implementation of LU decomposition using Peano Curves," in *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*. ACM, 2009, pp. 21–30.

[3] I. Venetis and G. Gao, "Mapping the LU decomposition on a many-core architecture: challenges and solutions," in *Proceedings of the 6th ACM conference on Computing frontiers*. ACM, 2009, pp. 71–80.

[4] D. Maurer and C. Wieners, "A parallel block LU decomposition method for distributed finite element matrices," *Parallel Computing*, 2011.

[5] J. Dongarra, "Performance of various computers using standard linear equations software," *Rapport technique, Computer Science Department, University of Tennessee, Knoxville, Tennessee*, 2011.

[6] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *SIAM Journal on Scientific Computing*, vol. 34, p. A206, 2012.

[7] X. Wang and S. Ziavras, "Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 4, pp. 319–343, 2004.
[8] X. Wang, "Design and resource management of reconfigurable multiprocessors for data-parallel applications," Ph.D. dissertation, New Jersey Institute of Technology, 2006.
[9] J. Johnson, T. Chagnon, P. Vachranukunkiet, P. Nagvajara, and C. Nwankpa, "Sparse LU decomposition using FPGA," in *International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, 2008.
[10] X. Wang, S. Ziavras, C. Nwankpa, J. Johnson, and P. Nagvajara, "Parallel solution of Newton's power flow equations on configurable chips," *International Journal of Electrical Power & Energy Systems*, vol. 29, no. 5, pp. 422–431, 2007.
[11] N. Kapre and A. DeHon, "Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*. IEEE, 2009, pp. 190–198.
[12] W. Wu, Y. Shan, X. Chen, Y. Wang, and H. Yang, "FPGA accelerated parallel sparse matrix factorization for circuit simulations," in *Proceedings of the 7th international conference on Reconfigurable computing: architectures, tools and applications*, Springer-Verlag. Springer, 2011, pp. 302–315.
[13] C. Fu and T. Yang, "Sparse LU factorization with partial pivoting on distributed memory machines," in *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*. IEEE, 1996, pp. 31–31.
[14] J. Liu, "The multifrontal method for sparse matrix solution: Theory and practice," *Siam Review*, pp. 82–109, 1992.
[15] I. Duff, "Parallel implementation of multifrontal schemes," *Parallel computing*, vol. 3, no. 3, pp. 193–204, 1986.
[16] I. Duff and J. Reid, "The multifrontal solution of indefinite sparse symmetric linear," *ACM Transactions on Mathematical Software (TOMS)*, vol. 9, no. 3, pp. 302–325, 1983.
[17] G. Alaghband and H. Jordan, "Sparse Gaussian elimination with controlled fill-in on a shared memory multiprocessor," *Computers, IEEE Transactions on*, vol. 38, no. 11, pp. 1539–1557, 1989.
[18] T. Davis, "A parallel algorithm for sparse unsymmetric lu factorization," Illinois Univ., Urbana, IL (USA), Tech. Rep., 1989.
[19] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, H. Simon, and P. Bjørstad, "Progress in sparse matrix methods for large linear systems on vector supercomputers," *International Journal of High Performance Computing Applications*, vol. 1, no. 4, pp. 10–30, 1987.
[20] A. Erisman, R. Grimes, J. Lewis, W. Poole Jr, and H. Simon, "Evaluation of orderings for unsymmetric sparse matrices," *SIAM journal on scientific and statistical computing*, vol. 8, p. 600, 1987.
[21] I. Duff, A. Erisman, and J. Reid, *Direct methods for sparse matrices*. Clarendon Press Oxford, 1986.
[22] J. Gilbert and T. Peierls, "Sparse spatial pivoting in time proportional to arithmetic operations," *SIAM journal on scientific and statistical computing*, vol. 9, no. 5, pp. 862–874, 1988.
[23] T. Davis and E. Natarajan, "Algorithm 8xx: KLU, a direct sparse solver for circuit simulation problems," *ACM Trans. MS*, vol. 5, no. 1, pp. 1–14, 2009.
[24] T. Davis, *Direct methods for sparse linear systems*. Society for Industrial Mathematics, 2006, vol. 2.
[25] S. Eisenstat and J. Liu, "Exploiting structural symmetry in a sparse partial pivoting code," *SIAM Journal on Scientific Computing*, vol. 14, p. 253, 1993.
[26] Y. Liao and C. Wong, "An algorithm to compact a vlsi symbolic layout with mixed constraints," in *Proceedings of the 20th Design Automation Conference*. IEEE Press, 1983, pp. 107–112.
[27] G. Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.
[28] X. Li and J. Demmel, "Making sparse Gaussian elimination scalable by static pivoting," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society, 1998, pp. 1–17.
[29] HSL, "Collection of Fortran codes for large-scale scientific computation," *See http://www.hsl.rl.ac.uk*, 2007.
[30] T. Davis, I. Duff, P. Amestoy, J. Gilbert, S. Larimore, E. Natarajan, Y. Chen, W. Hager, and S. Rajamanickam, "Suite sparse: a suite of sparse matrix packages."
[31] Xilinx, *Xilinx Core Generator Floating-Point Operator v4.0*, 2010.
[32] T. Davis, "Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 2, pp. 196–199, 2004.
[33] K. Kundert and A. Sangiovanni-Vincentelli, "Sparse user's guide–a sparse linear equation solver version 1.3 a," *University of California, Berkeley*, 1988.

Tarek Nechma received the B.Eng. degree in computer engineering and the Ph.D. degree in electrical and electronic engineering from the University of Southampton, Southampton, U.K., in 2006 and 2010, respectively. He is currently an analyst developer at Barclays Bank plc, London.

Mark Zwolinski received the B.Sc. degree in electronic engineering and the Ph.D. degree in electronics from the University of Southampton, Southampton, U.K., in 1982 and 1986, respectively. He is currently a Professor in the School of Electronics and Computer Science, University of Southampton. His current research interests include high-level synthesis, fault tolerance, and behavioral modeling and simulation.