FPGA Accelerated Parallel Sparse Matrix Factorization for Circuit Simulations*

Wei Wu, Yi Shan, Xiaoming Chen, Yu Wang, and Huazhong Yang

Department of Electronic Engineering, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University
100084, Beijing, China
wwnigel@gmail.com,
{shany08,chenxm05}@mails.tsinghua.edu.cn,
{yu-wang,yanghz}@mail.tsinghua.edu.cn

Abstract. Sparse matrix factorization is a critical step for the circuit simulation problem, since it is time consuming and computed repeatedly in the flow of circuit simulation. To accelerate the factorization of sparse matrices, a parallel CPU+FPGA based architecture is proposed in this paper. While the preprocessing of the matrix is implemented on CPU, the parallelism of numeric factorization is explored by processing several columns of the sparse matrix simultaneously on a set of processing elements (PE) in FPGA. To cater for the requirements of circuit simulation, we also modified the Gilbert/Peierls (G/P) algorithm and considered the scalability of our architecture. Experimental results on circuit matrices from the University of Florida Sparse Matrix Collection show that our architecture achieves speedup of 0.5x-5.36x compared with the CPU KLU results.

Keywords: Circuit Simulation, Sparse Matrix, LU Factorization, FPGA, Parallelism.

1 Introduction

With the growing complexity of integrated circuits, manual approaches of circuit test, such as breadboard probing, are not applicable. Those approaches were substituted by circuit simulation software such as SPICE from 1975, when the SPICE2 was delivered and became really popular[1]. Currently, there are many circuit simulation software packages designed based on SPICE, such as HSPICE owned by Synopsys[3] and PSPICE owned by Cadence Design Systems[3]. In the flowchart of circuit simulation, as drafted in Figure 1, a significant and time consuming step is repeatedly solving an equation set $Ax = b$ generated from targeted circuits according to Kirchhoff's circuit laws in Newton-Raphson Iterations[4]. In the very large scale integrated circuit (VLSI), the dimension of the matrix A, in equation set $Ax = b$, can reach up to tens of thousands or even million, which brings great challenges to the computations.

There are two methods of solving the equation set $Ax = b$, iterative methods and direct methods. The former approach attempts to solve the problem by successive

* This work was supported by National Key Technological Program of China No. 2008ZX01035-001 and NSFC No. 60870001.

approximations to the solution from an initial assumption. However, this approach is not applicable for all of the matrices, since its data stability greatly depends on the convergence of the matrix A. Distinguished from the iterative methods, the direct methods, such as factorizing the matrix A into a lower matrix L and an upper matrix U, deliver the exact solution by a finite sequence of computations in the absence of rounding error. In these methods, the LU factorization are widely utilized in circuit simulation because it is universal and irrelevant to the convergence of the matrix A. There are usually two steps of the LU factorization: 1) the pre-processing, which reorders the matrix to maintain data stability and sparsity, 2) the numeric factorization to compute L and U.

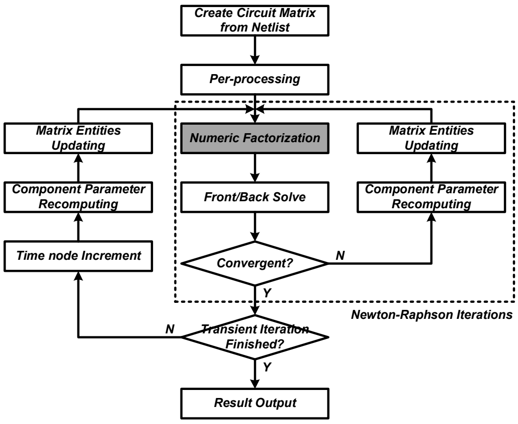
Create Circuit Matrix from Netlist
Per-processing
Matrix Entities Updating
Numeric Factorization
Matrix Entities Updating
Component Parameter Recomputing
Front/Back Solve
Component Parameter Recomputing
Time node Increment
Convergent?
N
Y
Newton-Raphson Iterations
N
Transient Iteration Finished?
Y
Result Output

Fig. 1. Flowchart of Circuit Simulation

Several characters of circuit simulation make the LU factorization of circuit matrices very different from other matrices.

Firstly, the circuit matrices are extremely sparse[13]. Therefore the algorithms designed to factorize circuit matrices shall be targeted for sparse matrix.

Secondly, during the update of nonlinear components, values of entities in matrix A vary during the simulation while the nonzero structure remains unchanged. Therefore, as shown in Figure 1, the pre-processing is only required to be carried out for once, while the numeric factorization, which is marked as gray in Figure 1, needs to be performed for many times. Consequently, the acceleration of the numeric factorization is critical to the performance of circuit simulation.

Currently, there are already many approaches to accelerate the LU factorization, by means of developing sophisticated algorithm on CPU and exploiting the parallelism on hardware, such as FPGA. Comparatively, the FPGA designs hold higher performance compared with the CPU implementations. However, most of the FPGA designs are with deficiencies on scalability and performance, and some of them can only be applied to special matrix patterns such as the *bordered diagonal block* (BDB) forms.

In this paper, a CPU+FPGA architecture for LU factorization is proposed. While the pre-processing is fulfilled on CPU, FPGA performs the numeric factorization of the matrix. The key contribution of this paper includes:

- Modified the G/P algorithm by extracting the symbolic analysis from the numeric factorization.
- Designing a CPU+FPGA architecture for LU factorization to accelerate the circuit simulation, with the pre-processing and numeric analysis implemented on CPU and the numeric factorization realized on FPGA. This architecture explore the parallelism by processing several columns of the matrix simultaneously on a set of PEs in FPGA.
- Modified the G/P algorithm again to consider the scalability to matrix size, a potential scalable version of current architecture is also introduce.
- Quantitative comparison between CPU KLU and our CPU+FPGA based implementation by a variety of circuit matrices from *the University of Florida Sparse Matrix Collection*[5].

The rest of this paper is organized in five parts. Section 2 introduces the preliminary algorithms and the related work. In section 3, the dataflow and complexity of both left-looking and right-looking algorithm are studied and our modification on G/P algorithm is introduced. Section 4 proposed our FPGA architecture, the detail implementation and its scalability. In section 5, the experimental results and its comparison with CPU implementation are provided. Section 6 concludes the paper and provides the future directions.

2 Preliminary Algorithms and Related Work

In this section, firstly, a typical data structure of sparse matrix, Compressed Column Storage (CCS) format, is introduced. Then a brief overview of the algorithms of direct LU factorization and its implementations on CPU and FPGA are provided.

2.1 Compressed Column Storage (CCS) Format [23]

A typical data structure, CCS format, is introduced to minimize the memory usage. The space requirement of the CCS format grows linearly with the number of nonzero entities (nnz) in the sparse matrix A. In the column-wise CCS format, all the row indexes and values of nnzs are stored in vector A_v column by column, while starting indexes of every column are stored in vector A_p.An example of column-wise CCS is provided as below. In A_v, (0 3) represents the nonzero entities in the *0th* row of the *1st* column, whose value is 3.

$$A = \begin{matrix} 2 & 3 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

$$A_v = [0\ 2\ \ 0\ 3\ 1\ 1\ 2\ 1], A_p = [0\ 1\ 3\ 4]$$

The algorithm and architecture in following of this paper are based on CCS format.

2.2 Algorithms of Direct LU Factorization

Pre-processing

As discussed in Section 1, the direct LU factorization includes 2 steps, the pre-processing and numeric factorization. The pre-processing part, which only needs to be executed once, performs the row/column pivoting to maintain data stability and to preserve sparsity. There are several different algorithms of pre-processing, which can be categorized into 2 groups: the static pivoting[11] and partial pivoting[12]. The static pivoting computes all the row/column permutation during pre-processing, while the partial pivoting just carries out part of the pivoting operations during pre-processing, some pivoting operations have to be carried out during the numeric factorization.

It is memory access intensive for the pivoting of rows in a column wise formatted matrix, because it is required to search for the entities of the targeted row in every column. To avoid the intensive irregular memory accesses during the numeric factorization, the static pivoting algorithm, which is used in SuperLU-DIST[10], is implemented on CPU, while the repeated numeric factorization is implemented on FPGA.

Numeric Factorization

The numeric factorization approaches are categorized into mainly two kinds of algorithms, the left-looking[6] and right-looking[6] algorithms, whose pseudo-codes are illustrated in Figure 2 and Figure 3 respectively. Other algorithms such as the G/P algorithm[24] and the left-right-looking[7] algorithm are derived from the basic left-looking algorithm. The multifrontal algorithm[8] [9] is evolved from the right-looking algorithm.

```
1 for k = 1 to n do
2     f = A(:, k)
3     for r = 1 to k - 1 do
4         $f = f - f(r) * L(r+1:n, r)$
5     end for
6     $U(1:k) = f(1:k)$
7     $L(k:n) = f(k:n) / U(k,k)$
8 end for
```

Fig. 2. Left-looking Algorithm

```
1 for k = 1 to n - 1 do
2     $L(k:n,:) = A(k:n,:)$
3     $U(:,k:n) = A(:,k:n) / L(k,k)$
4     $A(k+1:n, k+1:n)$
      $= A(k+1:n, k+1:n) - L(k+1:n,:) \times U(:,k+1:n)$
5 end for
```

Fig. 3. Right-looking Algorithm

In Figure 2, the left-looking algorithm updates matrix A column by column to calculated the L and U. While updating the current column of A, it goes leftwards to fetch the result of L. In right-looking, updating of the sub-matrix is carried out rightwards, since the sub-matrices are located at the bottom right corner.

The G/P algorithm is designed for sparse matrices factorization with the complexity proportional to arithmetic operations. The left-right-looking algorithm updates several columns are updated simultaneously in different processors or threads. In this algorithm, it is required to look leftwards to fetch data, and rightwards to notify other processors or threads that the data of current column are ready. The multifrontal algorithm is different from the former three algorithms. It reorganizes the factorization of original sparse matrix into a sequence of partial factorizations of dense smaller matrices according to *assembly tree*[8].

2.3 Related Work

CPU implementations
Currently, there are several CPU based implementations of sparse matrix factorization algorithms, such as SuperLU[10], KLU[13], PARDISO[14], UMFPACK[15] and MA41 in HSL Mathematical Software Library[16]. The sequential SuperLU[10] used the G/P algorithm with supernode involved, while the KLU does not consider supernode, but uses block triangular form (BTF) to fit circuit simulation matrices [13]. A multithread version of sequential SuperLU, SuperLU-MT, parallelizes the computation by means of multithreads[10]. For the right-looking algorithm, the SuperLU-DIST is a typical implementation on distributed memory systems[10]. PARDISO implements the left-right-looking algorithm on shared memory parallel machines[14]. In UMFPACK and MA41 of HSL Mathematical Software Library, the multifrontal algorithm is utilized[15] [16]. In this paper, we will mainly compare the performance of our implementation with KLU since it is the only CPU implementation targeted for circuit simulation problems.

FPGA implementations
The FPGA implementations of direct LU factorization appeared after 2000, which were much later than the CPU implementations. In 2004, V. Daga et al. introduced an architecture for direct LU factorization[17]. Later in 2006, they proposed a modified architecture with parameterized resource consumption[18]. However, these architectures are not designed for the sparse matrix. It is not applicable for large sparse matrices whose size may reach hundreds of thousands or even million. Xiaofang Wang et al. proposed an FPGA architecture for circuit matrices factorization in 2003[19]. Their work was limited to BDB matrix, but, actually, not all the circuits can be represented by BDB matrix. In 2008, Jeremy Johnson et al. implemented a row-wise right-looking algorithm in FPGA[20]. Their architecture depends on the data pattern, in which two copies of the matrix data are required, one in row-wise compressed format and the other in column-wise. Moreover, the performance of their architecture does not overcome the CPU implementations[21]. In 2010, Tarek Nechma et al. proposed a medium-grained paralleled architecture by means of column-dependency tree[23]. While all the data are loaded from DDR to on-chip BRAM before the computation, the problem size that can be processed is limited, since the size of on-chip memory cannot be too large. N Kapre et al. proposed a fine-grained sparse matrix solver based on a dataflow compute graph in 2009[22]. This architecture is a fine-grained one and greatly depends on the dataflow compute graph.

To be more useful for the circuit simulation, the FPGA architecture shall be scalable or at least have scalable potential, and compatible with all the form of sparse matrix and with a high performance. In the following two sections, the factorization algorithms and their suitable architecture will be analysed, and then we will propose our FPGA architecture.

3 Algorithm Complexity

In this section, the dataflow and complexity of both left-looking and right-looking algorithms for direct LU factorization are evaluated. Also a slight modification on G/P algorithm is proposed to reduce the overall complexity for circuit simulation problem.

3.1 Complexity of Left-Looking Algorithm

In this subsection, we will analyse the dataflow of both left-looking and right-looking algorithms. During studying the left-looking algorithm, we will consider the G/P algorithm since it targets for sparse matrices.

Left-looking algorithm

According to the pseudo-codes shown in Figure 2, the left-looking algorithm performs the factorization of the sparse matrix column by column. While updating the kth column, the complexity can be denoted as:

$$P_{left}(Col\ k) = n + (2 * \textstyle\sum_{i \forall\, U_{ik} \neq 0} L_i) + L_k$$

In this expression, n stands for the matrix demension, the L_i denotes the number of nnz in the ith column of matrix L. The first n in this expression stands for the complexity spends on searching for the nnzs in current column, because only nnzs are required to be processed. The second part, $2 * \sum_{i \forall\, U_{ik} \neq 0} L_i$, stands for the complexity of updating current column by *1st~(k-1)th* columns, while the third part, L_k, denotes the complexity of normalizing current column.

The second part and the third part of the complexity expression are proportional to arithmetic operations. However, the first part in the complexity grows with n^2, which will become the dominant part of the complexity when the matrix dimension n is large.

Modification on G/P algorithm

The G/P algorithm, as shown in Figure 4(a) and Figure 4(b), is designed to solve this problem[24], while Figure 4(b) is the solution for the *3rd* line in Figure 4(a). Before updating every column of A, it analysis the potential nnz structure in L and U by the information of nnz structure in the matrix L and current column of A. Moreover, the complexity of nnz analysis for a column is also proportional to the arithmetic operations rather than n.

```
(a) G/P Algorithm, Overview
1 for $k = 1$ to $n$ do
2     $b = A(:,k)$
3     $x = L \backslash b$
4     $U(1:k) = f(1:k)$
5     $L(k:n) = f(k:n) / U(k,k)$
6 end for
```

```
(b) G/P algorithm, Solve x = L \ b
1 Analysis for potential nonzero structure of L,U
2 for $i = 1$ to $k-1$ in the predicted structure do
3     $x(j+1:n) = x(j+1:n) - L(j+1:n, j)x(j)$
4 end for
```

Fig. 4. G/P Algorithm

Since the symbolic analysis of potential nonzero structure is independent to other arithmetic operations, this step can be performed before the factorization. Furthermore, as mentioned before, the nonzero structure remains unchanged during the flow of circuit simulation. the symbolic predicting can even be performed during the preprocessing before entering the inner loop of circuit simulation. After this modification on G/P algorithm, the complexity of updating a column is almost minimized and illustrated as below, with the complexity of symbolic analysis totally ruled out.

$$P_{Modified}(Col\ k) = (2 * \sum_{i \forall\ U_{ik} \neq 0} L_i) + L_k$$

It is also straightforward to conclude that the complexity of factorizing a sparse matrix with modified G/P algorithm as below:

$$P_{Modified} = \sum_{k=1}^{n}((2 * \sum_{i \forall U(i,k) \neq 0} L_i) + L_k)$$

The matrix data are stored in CCS format, which means the nnz in a column are stored one by one. During the processing of a column, we first create an index for the nnz in this column. Then, we can refer to the index to find the address of nnz in this column. The index of a column consumes n words of memory. Fortunately, the left-looking algorithm only factorizes the matrix column by column, so only the index of one column is required for this algorithm.

3.2 Complexity of Right-Looking Algorithm

When it comes to the right-looking algorithm, the matrix is processed by sub-matrix. Not like the left-looking algorithm, we cannot create the index for the entire sub-matrix since it may consume n^2 words of memory. Without the index information, it may be required to search in every column to find the data to be accessed. Although the arithmetic complexity of right-looking algorithm is the same as the left-looking algorithm, it may consume more time on data manipulation.

According to the analysis in this section, the modified G/P algorithm is the most suitable algorithm for circuit simulation problem.

4 Proposed Architecture

In this section, the architecture and implementation detail supporting modified G/P algorithm is proposed. Also the potential parallelism is analysed and corresponding parallel architecture is proposed. Then, the scalability of this architecture is also discussed.

4.1 Architecture for the Modified G/P Algorithm

To fulfill the function of modified G/P algorithm, a hardware architecture is proposed and illustrated in Figure 5.

This architecture is constructed of the following parts: the Processing Controller, the Arithmetic Logic, the Content Address Memory (CAM) and the Cache, including both the inner on and external one.

The Processing Controller controls the flow of factorizing a matrix and the state of this architecture. The CAM fulfills the function as an index. If the row ID i of $A(i,j)$ is inputted, the address of the nnz will be returned in one cycle. The Arithmetic Logic is

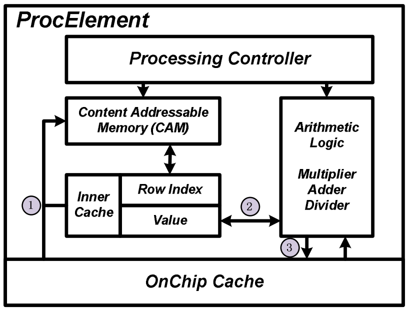

Fig. 5. Architecture for Modified G/P Algorithm

constructed of three units, a subtracter, a multiplier, and a divider. The Caches are construct of two parts, the inner part and the external part. Actually, the cache is implemented by a Tri-Port RAM (TPRAM). In these three ports, one write port and one read port are connected to local PE, while the rest read port is connected outwards for external access. The inner cache can be a virtual memory mapped on the On chip cache.

In this architecture, before the factorization, data are loaded to cache from CPU. When the factorization starts, the processing of every column in a matrix is performed in three steps, under the control of Processing Controller according to the state switching diagram in Figure 6.

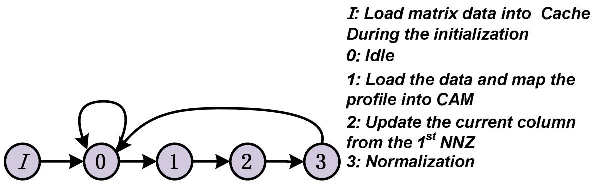
I: Load matrix data into Cache During the initialization
0: Idle
1: Load the data and map the profile into CAM
2: Update the current column from the 1st NNZ
3: Normalization
I
0
1
2
3

Fig. 6. State Switching Diagram of Processing Controller

In Figure 6, the first step is to load the data from on chip cache and to map the position of every nnz in CCS format into CAM. The second step is to update this column by the former columns of L, according to the nnz in current column of U. The third step is to normalize the entities in current column of L and dump them back to the on chip cache.

4.2 Parallelized Architecture

Potential parallelism

The architecture in the former subsection factorizes the matrix almost sequentially, except a few cycle level parallelism between the arithmetic operations. It seems that the G/P algorithm is a sequential algorithm because the processing of a column may require the data of former columns. However, three parallelism strategies can still be explored in this algorithm.

Firstly, the different column of the matrix can be processed simultaneously according to the elimination tree[7] in the parallel hardware such as multi-core GPP and FPGA. Secondly, to reduce the time on memory accessing and processing the dense block of matrix with optimized algorithm, the supernode is proposed to accelerate the processing. This approach is adopted in SuperLU[10] and PARDISO[14], but the supernode is not suitable for circuit matrices because they are extremely sparse[13]. The third parallelism is the fine-grained parallelism between the dataflow of every operations. N. Kapre et al. explored this parallelism in their FPGA architecture. However, the generation and optimization of the dataflow is required before the factorization.

Therefore, we only pursue the first parallelism by implementing a group of PEs in FPGA, while every PE process a column of the matrix independently.

Parallelized architecture

We introduce the module that factorizes a column in the sparse matrix in the former subsection. In this subsection, that module is referred as a PE. To achieve parallel processing, we implemented an architecture with several PEs. While processing a column in the matrix, all the column of matrix L might be accessed. Therefore, the data of matrix L are required to be shared to all PEs for accessing.

Our first attempt on the shared memory is an external DDR2 memory with an arbitrator to decide which PE holds the bus of the memory. That trial failed because the memory bandwidth is always the bottleneck of the system. To increase the bandwidth, we adopt a distributed shared memory to replace the original shared memory. The multi-PE architecture with distributed shared memory is shown in Figure 7.

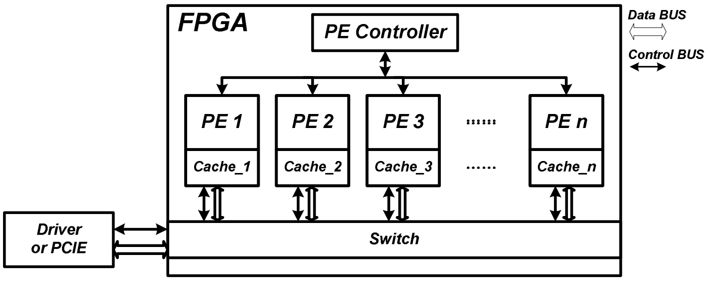

Fig. 7. Parallelized Architecture

In Figure 7, data are located at the cache distributed in every PE. To be easily realized, all the PEs are connected to a switch to construct a on chip network, in which a PE can access the data in its own cache directly and access the data stored in other cache via a switch. Since data need to be prepared in caches before the factorization, a Driver interface is also reserved on the switch for the loading matrix data from PC to FPGA, corresponding to state I in Figure 6. By replacing a single shared memory to a set of distributed shared memory, the peak bandwidth is increased by n times, in which the n stands for the number of PEs, also the number of caches in the distributed memory. In our prototype, we use 16 PEs. Under this configuration, the performance of our hardware exceeds KLU on most circuit matrices.

4.3 Scalability

As mentioned in subsection 2.2, many related designs suffer from the scalability problem[17] [18] [23]. In this subsection, we will discuss the scalability of our architecture on both the architecture and the algorithm aspects.

Firstly, we need consider how to make full use of the on-chip memory to enable our architecture to process matrices as large as possible, in other words, the scalability to matrix size. As discussed in subsection 4.1, a CAM is required in every PE to facilitate the indexing of nnzs in the column processed by PE. To index a whole column, the CAM will consume *n* words of memory, in which *n* stand for the matrix dimension. In an *m**PE system, it will take m*n words of memory for the CAM. Since there may be only average 4-5 nnzs in a column of circuit matrices, a matrix will only consume 4-5**n* words of memory to store. Comparatively, the memory consumption of CAM is very considerable.

To alleviate the memory consumption on CAM, we need to reduce the size of the CAM without influencing the performance. Therefore, we modified the G/P algorithm again by divide a column of the matrix into several sections. Then we processing a column in the matrix section by section, rather than row by row. The pseudo-code of section based G/P algorithm consideration is shown in Figure 8.

```
1  for k = 1 to n do
2      for r = 1 to j do
3          $f = A(SecStart : SecEnd, k)$
4          for $x = 1$ to $\min(SecStart - 1, r - 1)$ and $U(x, k) \neq 0$
5              $f = f - U(x,k) * L(SecStart : SecEnd, x)$
6          end for
7          if $(SecStart \leq r)$
8              for $x = SecStart$ to $r - 1$ and $U(x, k) \neq 0$
9                  $f = f - f(x) * L(SecStart : SecEnd, x)$
10             end for
11         end if
12         $U(SecStart : k) = f(SecStart : k)$
13         $L(k : SecEnd) = f(k : SecEnd) / U(k,k)$
14     end for
15 end for
```

Fig. 8. Section Based G/P Algorithm for Scalability

In the pseudo-code, if the length of a section is m, there will be j sections in current column, while $j = floor(\frac{n-1}{m}) + 1$. *SecStart* and *SecEnd* in the inner for loop refer to the first and the last row indexes in a section.

An example of section based left-looking algorithm, in which a column is divided into 2 sections, is illustrated in Figure 9. The lines in Figure 9 indicate the update steps. In the original algorithm, the k^{th} column is updated by the 3 nonzero entities from top to bottom, whereas, in the modified version, the yellow line is updated section by section. The solid lines stand for the steps of updating the first section, which

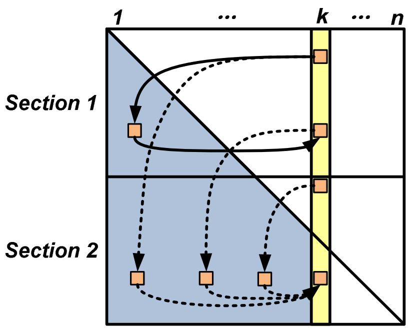

Fig. 9. Dataflow of Section Based G/P Algorithm for Scalability

is same as the original left-looking algorithm, while the steps of updating the second section are marked in dotted line. The second section is firstly updated by the nonzero entities of *U*, which are computed in the *1st* section, and then by the nonzero entities in current section until the last nnz in this column of *U* is reached.

Besides considering the scalability of matrix size, we should also consider the extension of the architecture itself. As mentioned in subsection 4.2, we increase the memory bandwidth by means of the distributed memory. In this architecture, all the inter-PE communication are performed by the fully connected switch, and the total connection in this switch is proportional to n^2. Therefore the design of the switch becomes increasingly difficult when the quantity of PE mounts up.

This scalability problem can be solved by replacing the switch based network of PEs and the mesh network is adopted to connect the PEs, which is also introduced in [22]. In the mesh network, the communication burden is not mounted on a switch anymore, but distributed to the nodes in the network. The communication overhead will be replaced by inserting more network interfaces and designing more sophisticated communication strategy using meshed network. Moreover, the communication time between two PEs should be a variant and may influence the overall performance when the mesh becomes larger. It is also possible to create a multi-FPGA version with the mesh network architecture. In this paper, we do not implement the hardware prototype of mesh network based architecture, but list it as a solution for scalability of the hardware architecture.

5 Experimental Result

In this section, we discussed the resource consumption of with parallelized architecture and the performance compared with the software implementation.

5.1 Resource Consumption

In our implementation, we used the floating point divider generated in Altera MegaWizard, the floating point subtracter and multiplier in the proposed architecture are designed by ourselves, which is faster and more efficient compared with the standard IP. The resources consumption of these arithmetic units and PEs are considered based on Altera Stratix III FPGA, EP3SL340, with Quartus II 10.1. The detail resource consumption is given in Table 1 as below.

Table 1. Resource Utilization

	Logic Utilization (%)	BRAM Utilization (%)	DSP Utilization (%)	Clocks (MHz)
Subtracter	<1%	0	0	203
Multiplier	<1%	0	4(<1%)	248
Divider	<1%	<1%	16 (3%)	125
PE	1%	3%	20(4%)	125
16PEs	36%	48%	240(56%)	83

In this Stratix III FPGA chip, only about 50% resource is consumed in the 16-PE architecture. We can notice that the resource utilization increases quickly along with PE numbers due to the resource consumed by the switch. With the 32-PE architecture, Quartus II can even not fulfill the Place & Route step. Meanwhile the maximum frequency is lower when the number of PE increase, which is also resulted from the bottleneck of the switch based PE interconnection. Fortunately, the 16-PE architecture is enough to achieve the acceleration compared with CPU implementation such as KLU. If more PE is needed, the mesh network based multi-PE architecture can be adopted to reduce the resource consumed by PE interconnection. The transmission latency of mesh network should not be a big problem because the number of PE does not need to be too large. 16~32 PE is enough to achieve acceleration on CPU implementations.

5.2 Performance

In this subsection, we compared the performance of the parallel architecture with KLU on a set of circuit matrices from *the University of Florida Sparse Matrix Collection*, as illustrated in Table 2.

Table 2. Test Sets and Results

Matrix	Matrix size	Non-Zeros	Sparsity (%)	KLU runtime (ms)	FPGA runtime (ms)	Acceleration
rajat19	1157	5399	0.403	0.466	0.208	2.24
circuit_1	2624	35823	0.520	2.269	0.719	3.15
circuit_2	4510	21199	0.104	1.868	0.694	2.69
add32	4960	23884	0.097	1.841	0.343	5.36
meg4	5860	26324	0.077	0.338	0.687	0.5

In Table 2, the KLU runtime is tested on an Intel i7 930 platform. To the FPGA platform, we test the performance by counting the cycles of factorization in Modelsim. Since it is not necessary to use switch to construct the network, we used the max frequency of PE (125MHz) to evaluate the performance. From the result acquired, we learnt that our architecture can achieve 0.5x-5.36x acceleration compared with KLU. The geometric average acceleration is about 2.19x. The acceleration varies because

KLU employ the partial pivoting during numeric factorization, while static pivoting is utilized in our system. The difference in pivoting strategy may result in different non-zero structure, which means different complexity in factorization.

In the switch based architecture, the matrix size is limited by the on-chip memory size of FPGA. Therefore we just test several small matrices in this experiment. Since the architecture can process matrices by section, large matrices can be factorized by employing larger FPGA or ASIC with more on-chip memory, especially the ASIC with 3D-stacked memory architecture.

6 Conclusions and Future Work

In this paper, a parallel CPU+FPGA based architecture for sparse matrix LU factorization is proposed. Not only the pre-processing is carried out on CPU, the symbolic analysis in G/P algorithm is also extracted from the numeric factorization of G/P algorithm and executed in CPU. The parallelism in the numeric factorization is explored by processing columns of sparse matrix simultaneously in different PEs of our architecture.

The proposed architecture is not only configurable on the scale of hardware by extending PEs, but also it is scalable to the matrix size, by dividing large matrix into small sections and processing them in PEs. The performance of our architecture exceeds the latest available KLU on a variety of circuit matrices from *the University of Florida Sparse Matrix Collection,* and achieves average 2.19x acceleration on KLU.

A few potential changes are identified as future work to improve upon current architecture. First, additional logic can be implemented in current architecture to reuse the PE for front/back solver. Second, computational burden can be further reduced by partially factorization: since only part of the entities change their value during the circuit simulation, we can only carry out computations that related to modified entities and leave constant entities alone.

References

1. Nagel, L.: Spice 2: A computer program to stimulate semiconductor circuits. University of California, Berkeley (1975)
2. Nagel, L.W., Pederson, D.O.: SPICE (Simulation Program with Integrated Circuit Emphasis), Memorandum No. ERL-M382. University of California, Berkeley (April 1973)
3. Vladimirescu, A.: SPICE – The Third Decade. In: Proc. 1990 IEEE Bipolar Circuits and Technology Meeting, Minneapolis (September 1990)
4. Warwick, C.: Everything you always wanted to know about SPICE* (*But were afraid to ask) (PDF). EMC Journal (Nutwood UK Limited) (82), 27–29 (2009)
5. Davis, T.A., Hu, Y.: University of Florida sparse matrix collection. ACM Trans. Math. Software (2010) (to be appeared), `http://www.cise.ufl.edu/sparse/matrices`
6. Sparse Gaussian Elimination on High Performance Computers, Computer Science Division, UC Berkeley, UCB//CSD-96-919 (LAPACK Working Note #127) (September 1996)
7. Schenk, O., Gartner, K., Fichtner, W.: Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors. BIT 40(1), 158–176 (2000)

8. Amestoy, P.R., Duff, I.S.: Vectorization of a multiprocessor multifrontal code. The International Journal of Supercomputer Applications 3, 41–59 (1989)
9. Rothberg, E., Gupta, A.: An evaluation of left-looking, right-looking and multifrontal approaches to sparse Cholesky factorization on hierarchical-memory machines. Int. J. High Speed Computing 5, 537–593 (1993)
10. Li, X.S.: An Overview of SuperLU: Algorithms, Implementation, and User Interface. ACM Trans. on Math. Software 31(3), 302–325 (2005)
11. Li, X.S., Demmel, J.W.: Making Sparse Gaussian Elimination Scalable by Static Pivoting. In: Proceedings of Supercomputing (1998)
12. Demmel, J.W., et al.: A Supernodal Approach to Sparse Partial Pivoting. SIAM J. Matrix Analysis and Applications 20(3), 720–755 (1999)
13. Natarajan, E.: KLU: A high performance sparse linear solver for circuit simulation problems. Master's Thesis, University of Florida (2005)
14. Schenk, O., et al.: PARDISO: a high performance serial and parallel sparse linear solver in semiconductor device simulation. Future Generation Computer Systems 789(1), 1–9 (2001)
15. Davis, T.A.: Algorithm 8xx: UMFPACK V4.1, an unsymmetric-pattern multifrontal method with a column preordering strategy. University of Florida, Tech. Rep. TR-03-007, submitted to ACM Trans. Math. Software (2003)
16. Amestoy, P.R., Puglisi, C.: An unsymmetrized multifrontal LU factorization. SIAM J. Matrix Anal. Appl. 24(2), 553–569 (2002)
17. Daga, V., Govindu, G., Gangadharpalli, S.: Efficient Floating-point based Block LU Decomposition on FPGAs. In: Proc. of ERSA 2004 (June 2004)
18. Zhuo, L., Prasanna, V.K.: High-performance and parameterized matrix factorization on FPGAs. In: International Conference on Field Programmable Logic and Applications (2006)
19. Wang, X., Ziavras, S.G.: Parallel direct solution of linear equations on FPGA-based machines. In: International Parallel and Distributed Processing Symposium (2003)
20. Johnson, J., Chagnon, T., Vachranukunkiet, P., Nagvajara, P., Nwankpa, C.: Sparse LU decomposition using FPGA. In: International Workshop on PARA (2008)
21. Chagnon, T.: Architectural support for direct sparse LU algorithms. Master Dissertation paper (March 2010)
22. Kapre, N., DeHon, A.: Parallelizing sparse Matrix Solve for SPICE circuit simulation using FPGAs. In: International Conference on Field-Programmable Technology (2009)
23. Nechma, T., et al.: Parallel Sparse Matrix Solver for Direct Circuit Simulations on FPGAs. In: Proceedings of 2010 ISCAS (2010)
24. Gilbert, J.R., Peierls, T.: Sparse partial pivoting in time proportional to arithmetic operations. SIAM J. Sci. Statist. Comput. 9, 862–874 (1988)