Nonzero Pattern Analysis and Memory Access Optimization in GPU-based Sparse LU Factorization for Circuit Simulation

Xiaoming Chen, Du Su, Yu Wang, Huazhong Yang
Department of Electronic Engineering
Tsinghua National Laboratory for Information Science and Technology
Tsinghua University, Beijing 100084, China
chenxm05@mails.tsinghua.edu.cn, sudusuperman@163.com,
yu-wang@tsinghua.edu.cn, yanghz@tsinghua.edu.cn

ABSTRACT

The sparse matrix solver is a critical component in circuit simulators. Some researches have developed GPU-based LU factorization approaches to accelerate the sparse solver. But the performance of these solvers is constrained by the irregularities of sparse matrices. This work investigates the nonzero patterns and memory access patterns in sparse LU factorization, and explores the common features to give guidelines on the improvements of the GPU solvers. We further propose a crisscross blocked implementation on GPUs. The proposed method attains average speedups of 1.68× compared with the unblocked method and 2.2× compared with 4-threaded PARDISO, for circuit matrices.

Categories and Subject Descriptors

G.1.0 [**Numerical Analysis**]: General—*Parallel algorithms*; J.6 [**Computer-aided Engineering**]: Computer-aided design (CAD)

General Terms

Algorithms

Keywords

graphics processing unit (GPU); sparse LU factorization; memory access pattern

1. INTRODUCTION

The sparse matrix solver for solving $\mathbf{Ax} = \mathbf{b}$ is a critical component in Simulation Program with Integrated Circuit Emphasis (SPICE) [1]-based circuit simulators. Circuit matrices after post-layout extraction can easily reach a dimension of several million, SPICE-based simulators may take days or even weeks to perform time-domain simulations. The sparse solver is repeated for thousands of times in a time-domain simulation, as shown in Fig. 1, so it is the most time-consuming step in post-layout simulations.

There are two important features of the sparse solver in SPICE-based simulators. First, circuit matrices are highly sparse, asymmetric, and irregular. This feature makes that Basic Linear Algebra Subroutines (BLAS) [2]-based solvers tend to perform poorly for circuit matrices [3], such as SuperLU [4], PARDISO [5], etc. So KLU [3], which is specially designed for circuit simulation problems, does not utilize BLAS. Second, during the iterations of a time-domain simulation, although the values of matrix elements change, the nonzero structure is fixed. This feature makes that only the first LU factorization is performed with pivoting, subsequent LU factorizations can use the fixed pivoting order and fixed structure of the LU factors obtained by the first factorization [3]. A subsequent factorization without pivoting is called a *re-factorization*.

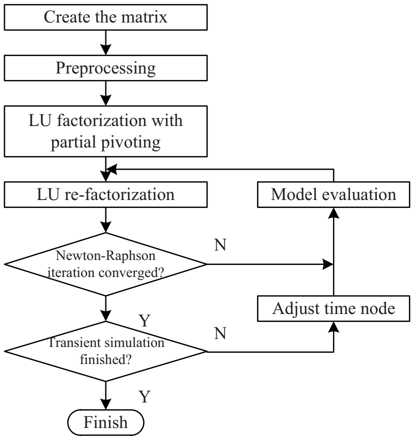

Create the matrix
Preprocessing
LU factorization with partial pivoting
LU re-factorization
Model evaluation
Newton-Raphson iteration converged?
N
Y
Adjust time node
Transient simulation finished?
N
Y
Finish

Figure 1: Flow of SPICE time-domain simulations.

Recently, a parallel sparse solver named NICSLU running on multicore CPUs is developed for circuit simulation problems [6–8], and its GPU implementation is also proposed [9]. They have been proved to be efficient for circuit matrices. However, as reported in [9], although the GPU implementation is faster than the 8-threaded CPU implementation, the peak performance achieved by NVIDIA GTX580 is only 10 Giga floating-point operations per second (Gflop/s), which is much smaller than the theoretic peak performance of NVIDIA GTX580 (200 Gflop/s for double-precision). Consequently, there is still a large potential to further accelerate the GPU solver. The challenge of GPU-based sparse LU factorization is the irregularity. The irregular nonzero distribution leads to numerous indirect and irregular memory access patterns, which greatly influences the performance on GPUs. Consequently, it is important to analyze and optimize the nonzero patterns and memory access patterns in

IA^3 2013 November 17, 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2503-5/13/11...$15.00.
http://dx.doi.org/10.1145/2535753.2535763.

GPU-based sparse LU factorization. Therefore, we make the following main contributions in this paper.

- Nonzero patterns in sparse LU factorization is investigated by a theoretical analysis. Fill-ins are not evenly distributed in the LU factors, the matrix will become denser and denser during the factorization procedure. This phenomenon prompts us to rethink the optimal implementation of the sparse factorization implementation for circuit matrices.
- Memory access patterns in sparse LU factorization are further analyzed by visualization methods. We find that sparse LU factorization should be separated into two parts to perform different optimization strategies. The former has larger stride in memory access patterns, but fewer memory requests; the latter has smaller stride but more memory requests. These are common characteristics in sparse LU factorization. Consequently, optimizing the two parts should have different focuses.
- We propose a crisscross blocked LU factorization approach which is suitable for GPUs. The proposed blocked approach combines both sparse and dense algorithms to take full advantage of the features of the two parts. It is on average 1.68× faster than the unblocked method and 2.2× faster than 4-threaded PARDISO, for circuit matrices.

2. RELATED WORK

2.1 Memory Analysis/Optimization on GPUs

Until now, there are not much work on analysis and optimization of memory access patterns on GPUs. Jang et al. proposed an analytical model to analyze and optimize memory access patterns on GPUs [10]. This work focused on regular memory access patterns but paid little attention on random patterns. Hugues et al. evaluated the performance of various sparse matrix formats for sparse matrix-vector multiplication (SpMV) on GPUs [11]. Che et al. proposed a framework to optimize the efficiency of DRAM accesses through memory layout remapping and index transformation [12], but the index transformation function was manually tried and found. A memory model for GPUs was proposed [13], which focused on 2D block-based array representations and was only suitable for dense computing kernels. An analytical method was proposed [14] to quantize the locality in memory access patterns. However, this method could only get a score for an algorithm, but the details were ignored.

2.2 Sparse Direct Solvers on GPUs

In recent years, some GPU-based direct solvers were proposed [9, 15–19]. Among them, the left-looking Gilbert-Peierls (G-P) algorithm [20] was mapped onto GPUs without pivoting, which was suitable for circuit matrices [9]. PARDISO was also mapped onto GPUs [18], but the speedup was only about 2× compared with a sequential CPU implementation. Other approaches are all based on the supernodal or multifrontal algorithm, which involve off-loading the time-consuming dense kernels to GPUs by using CUDA BLAS (CUBLAS) [21]. CUBLAS-based approaches are not suitable for circuit matrices because circuit matrices are highly sparse and so are their LU factors. Parallel sparse LU factorization on GPUs is more difficult than on CPUs, because CPUs have large caches to handle irregular memory access patterns and irregular control flows, but GPUs do not have such features so the GPU performance is significantly constrained by the irregular structure of the sparse matrices and the irregular memory access patterns.

3. BACKGROUNDS

3.1 CSR Format

The compressed sparse row (CSR) format [22] is one of the most widely used formats to store sparse matrices, as illustrated in Fig. 2. Let n be the dimension of the sparse matrix $\mathbf{A}$, and nnz be the number of nonzeros in $\mathbf{A}$. The CSR format uses three arrays to store $\mathbf{A}$: `A_x`, `A_i`, and `A_p`. `A_x` is a floating-point array of length nnz, storing the values of the nonzeros row by row; `A_i` is an integer array of length nnz, storing the column index of each nonzero; `A_p` is an integer array with length $n+1$, storing the position of the nonzero in `A_i` or `A_x` which is the first nonzero of a row. In the following contents, we use `MAT_x`, `MAT_i`, and `MAT_p` to represent the CSR arrays of the matrix named `MAT`.

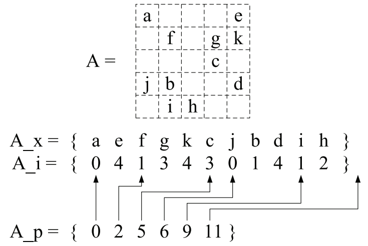

A =
a e
f g k
c
j b d
i h
A_x = { a e f g k c j b d i h }
A_i = { 0 4 1 3 4 3 0 1 4 1 2 }
A_p = { 0 2 5 6 9 11 }

Figure 2: Illustration of the CSR format.

3.2 Sparse LU Factorization for Circuit Simulation

GPU-based sparse LU factorization for circuit simulation problems consists of three steps: preprocessing, numeric re-factorization, and substitution [9]. The preprocessing step performs row/column permutations to reduce fill-ins, followed by a left-looking factorization algorithm [20] (with numeric pivoting) to obtain the structure of $\mathbf{L}$ and $\mathbf{U}$. These two steps are performed on CPUs. Numeric re-factorization is performed on GPUs and repeated for many times. As shown in Section 1, re-factorizations use the fixed nonzero structure and pivoting order obtained in preprocessing, which avoids per-iteration changes to the matrix structure and is suitable for GPU implementation.

Algorithm 1 shows the pseudo code of the row-based sparse numeric factorization algorithm. It is a transposed version (without numeric pivoting) of the left-looking G-P algorithm [20], so it can be called *up-looking* algorithm. Algorithm 2 shows the C-like code for factorizing row i. In this code, the CSR arrays of $\mathbf{L}$ and $\mathbf{U}$ do not contain the diagonals. `ldiag` is the diagonal of $\mathbf{L}$ (the diagonal values of $\mathbf{U}$ are all 1.0, which are not stored); `x` is a `double`-type vector of length n.

4. NONZERO PATTERN ANALYSIS FOR SPARSE LU FACTORIZATION

Algorithm 1 Row-based sparse LU factorization algorithm

```
for $i = 1 : n$ do
    $\mathbf{x}$=$\mathbf{A}(i, :)$;
    for $j = 1 : i - 1$ do
        $\mathbf{x}(j + 1 : n)-= \mathbf{x}(j) \times \mathbf{U}(j, j + 1 : n)$;
    end for
    $\mathbf{L}(i, 1 : i) = \mathbf{x}(1 : i)$;
    $\mathbf{U}(i, i : n) = \mathbf{x}(i : n)/\mathbf{x}(i)$;
end for
```

Algorithm 2 Detailed code for factorizing row i

```
//copy row i of A into a dense vector x
for (j=A_p[i]; j<A_p[i+1]; ++j)
{
  x[A_i[j]] = A_x[j];
}
//numeric accumulation from dependent rows
for (j=L_p[i]; j<L_p[i+1]; ++j)
{
  id = L_i[j];
   xj = x[id];
   for (k=U_p[id]; k<U_p[id+1]; ++k)
   {
     x[U_i[k]] -= xj*U_x[k];
   }
}
//storing factorization results
xj = x[i];
x[i] = 0.;
ldiag[i] = xj;
for (j=L_p[i]; j<L_p[i+1]; ++j)
{
   id = L_i[j];
   L_x[j] = x[id];
   x[id] = 0.;
}
for (j=U_p[i]; j<U_p[i+1]; ++j)
{
   id = U_i[j];
   U_x[j] = x[id] / xj;
   x[id] = 0.;
}
```

During sparse LU factorization, fill-ins are generated. The distribution of fill-ins has large impact on memory access patterns and consequently, the overall performance also strongly depends on the nonzero patterns. In this section, we theoretically analyze the characteristics of the nonzero patterns in sparse LU factorization, and explore the common characteristics, to further give guidelines on memory access improvement.

4.1 Elimination Graph

For simplicity, only symmetric matrix is considered in this section, but the method can also be extended to the asymmetric case by using the concept of bipartite graph [23]. A symmetric n-by-n matrix $\mathbf{A}$ can be represented by an undirected graph $G_0 = (V_0, E_0)$, where $V_0 = \{1, 2, \cdots, n\}$. An edge (i, j) is in E_0 if and only if $a_{ij} \neq 0 (i \neq j)$.

The LU factorization process can be represented by a series of *elimination graphs* [24]: $G_1(V_1, E_1)$, $G_2(V_2, E_2)$, $\cdots$, $G_n(V_n, E_n)$, where $G_k(V_k, E_k) (k = 1, 2, \cdots, n)$ describes the nonzero pattern of the right-bottom $(n-k)$-by-$(n-k)$ submatrix still to be factorized after the first k pivots have been chosen and eliminated. At step k, the kth pivot p_k is selected from V_{k-1}. Nodes (in V_{k-1}) adjacent to p_k become a clique (a fully connected subgraph) by adding edges between these nodes. p_k and edges that are linked to p_k are removed from G_{k-1} to generate G_k. The edges added correspond to fill-ins generated at the kth step of LU factorization. Let $ADJ_k(v)$ denote the set of nodes adjacent to node v in G_k. The kth step is described by [24]

$$V_k = V_{k-1} \backslash \{p_k\} \tag{1}$$

$$E_k = [E_{k-1} \cup (ADJ_{k-1}(p_k) \times ADJ_{k-1}(p_k))] \cap (V_k \times V_k) \tag{2}$$

An example is illustrated in Fig. 3.

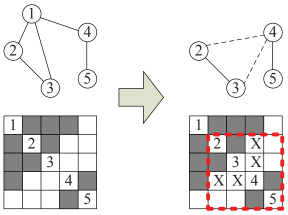


Figure 3: Eliminating node 1. The set of nodes $\{2, 3, 4\}$ becomes a clique. Filled grids represent original nonzeros."X" represents fill-ins caused by eliminating node 1. The remaining graph describes the nonzero pattern of the right-bottom 4-by-4 matrix.

4.2 Nonzero Pattern Analysis for Sparse LU Factorization

In this subsection, we analyze the nonzero patterns of sparse LU factorization based on a theoretical average analysis.

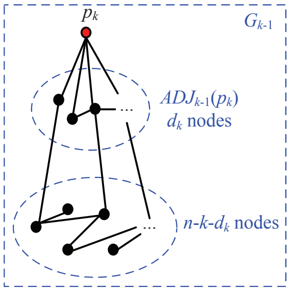


Figure 4: Illustration of the kth step.

Consider the kth step, before p_k is eliminated, the average degree of all nodes in G_{k-1} is

$$avg\_deg_{k-1} = \frac{2|E_{k-1}|}{|V_{k-1}|} = \frac{2|E_{k-1}|}{n-k+1} \tag{3}$$

When p_k is eliminated, the edges linked to p_k are also eliminated, and $ADJ_{k-1}(p_k)$ becomes a clique. We want to calculate how many edges are added in this process. Let d_k be the degree of p_k.

At step k, the nodes V_{k-1} can be partitioned into three parts: p_k, $ADJ_{k-1}(p_k)$, and other nodes, as shown in Fig. 4. Consider a node $v \in ADJ_{k-1}(p_k)$, except for the edge linked to p_k, it has on average $\frac{2|E_{k-1}|}{n-k+1} - 1$ edges linked to nodes in $ADJ_{k-1}(p_k)$ and the other $n-k-d_k$ nodes. The average number of edges that link node v and nodes in $ADJ_{k-1}(p_k)$ is

$$\left(\frac{2|E_{k-1}|}{n-k+1} - 1 \right) \times \frac{d_k - 1}{n-k} \tag{4}$$

Consequently, the average number of edges in $ADJ_{k-1}(p_k)$

is (before it becomes a clique)

$$\left(\frac{2|E_{k-1}|}{n-k+1}-1\right)\times\frac{d_k-1}{n-k}\times\frac{d_k}{2} \tag{5}$$

When $ADJ_{k-1}(p_k)$ becomes a clique, the number of edges is $\frac{d_k(d_k-1)}{2}$. Consequently, the increment of the average number of edges in $ADJ_{k-1}(p_k)$ is

$$\begin{aligned}\Delta_k &= \tfrac{d_k(d_k-1)}{2}-\left(\tfrac{2|E_{k-1}|}{n-k+1}-1\right)\times\tfrac{d_k-1}{n-k}\times\tfrac{d_k}{2}\\ &= \tfrac{d_k(d_k-1)}{2}\left[1-\tfrac{1}{n-k}\left(\tfrac{2|E_{k-1}|}{n-k+1}-1\right)\right]\end{aligned} \tag{6}$$

Consequently, when step k is finished, the average number of edges in G_k is

$$|E_k| = |E_{k-1}| - d_k + \Delta_k \tag{7}$$

Based on Eq. (7) and (6), $|E_k|$ can be represented by a function of $|E_{k-1}|$ and d_k.

In an actual factorization, a node with the minimum degree is usually selected as the pivot at each step to minimize fill-ins, this is so called the minimum degree ordering algorithm [24]. Regardless of the pivot selection strategy, all the d_k values are implementation related, and depend on the specific matrix. For the purpose of simple and theoretical analysis, we assume that each d_k is related to the average degree of all nodes at the corresponding step, i.e.

$$d_k = \alpha \times avg\_deg_{k-1} = \frac{2\alpha|E_{k-1}|}{n-k+1} \tag{8}$$

where α is a constant.

Combining Eq. (7), (6) and (8), we get

$$\begin{aligned}|E_k| &= \tfrac{-4\alpha^2|E_{k-1}|^3}{(n-k)(n-k+1)^3}+\tfrac{[2\alpha+2\alpha^2(n-k+1)]|E_{k-1}|^2}{(n-k)(n-k+1)^2}\\ &+\tfrac{[(n-k)^2-(3\alpha-1)(n-k)-\alpha]|E_{k-1}|}{(n-k)(n-k+1)}\end{aligned} \tag{9}$$

Eq. (9) is a recursion formula of $|E_k|$. It is quite difficult to deduce a general term formula of $|E_k|$, maybe impossible.

The *density* of an n-by-n sparse matrix is defined as the ratio of the nonzero count to n^2. When step k is finished, the density of the $(n-k)$-by-$(n-k)$ submatrix to be factorized is

$$density(k) = \min\left\{\frac{2|E_k|+V_k}{V_k^2},1\right\} = \min\left\{\frac{2|E_k|+n-k}{(n-k)^2},1\right\} \tag{10}$$

Fig. 5 shows the trends of the density, under different α values. For comparison, a real example of sparse LU factorization is shown in Fig. 6, which shows a similar trend compared with Fig. 5.

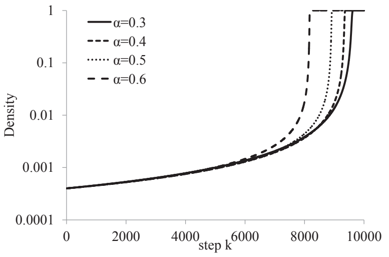


Figure 5: Trends of the density. $|V_0| = n = 10000, |E_0| = 15000$.

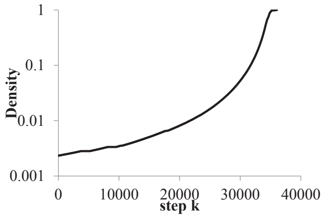


Figure 6: Trends of the density for factorizing onetone1 (36057×36057).

4.3 Summary of This Section

As a conclusion of the above discussions and results, we make an important observation that during the process of sparse LU factorization, the remaining submatrix will generally become denser and denser, and finally it becomes near fully dense or even fully dense. Thus, the nonzero distribution is irregular in sparse LU factorization.

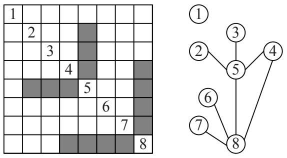


Figure 7: An example that does not fit the conclusion and its corresponding elimination graph.

Note that the above theoretical analysis is not a strict proof, since the above conclusion is not true for any matrix. One can simply verify that the matrix shown in Fig. 7 does not fit the conclusion, since it does not generate any fill-ins during LU factorization by using the natural pivoting order (i.e. $1, 2, \cdots$). Besides, matrices that can be permuted to a banded shape do not fit the conclusion either. However, our target is circuit simulation and these special cases do not exist in circuit simulation problems. The conclusion is based on a theoretical average analysis, which can stand for general cases. Anyhow, the conclusion prompts us to rethink what the optimal implementation of the sparse LU factorization algorithm is.

5. MEMORY ACCESS ANALYSIS FOR SPARSE LU FACTORIZATION

In this section, the memory access patterns in sparse LU factorization are analyzed by proposed *memory request distribution (MemReqDist)* and *memory access stride distribution (MemStrdDist)*. The two concepts are proposed to visualize the features of memory access patterns. All the following analysis is machine-independent, we attempt to show the intrinsic characteristics and expose the common characteristics.

5.1 Memory Request Distribution

We attempt to visualize the distribution of memory requests on the matrix plane, to show the number of memory requests at each position of the matrix. To achieve this goal, we calculate the number of memory requests at each nonzero position. Take `x[U_i[k]] -= xj*U_x[k]` (line 13 of Algorithm 2) as an example, there are four memory requests: one read operation for `U_x[k]` at row *id* (*id* is in line 9 of Algorithm 2) column $U\_i(k)$; one read operation for `U_i[k]` at row *id* column $U\_i(k)$; one read operation and one write operation for `x[U_i[k]]`, both at row i column $U\_i[k]$.

Using this method, the number of memory requests at each nonzero position can be calculated, then the matrix plane is partitioned into $T \times T$ tiles, the average number of memory requests in each tile is calculated. We call this distribution *MemReqDist*. An example of the *MemReqDist* is shown in Fig. 8 (for onetone1), using 60×60 tiles.

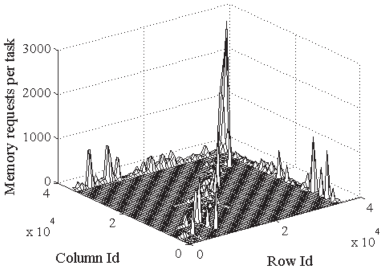


Figure 8: *MemReqDist* of onetone1.

5.2 Memory Access Stride Distribution

For GPU kernels, contiguous memory access addresses lead to more coalesced access patterns, which can greatly reduce the number of memory transactions [25]. *MemStrdDist* describes the contiguity of memory access addresses. The smaller the stride is, the more contiguous the memory access addresses are.

The memory access stride comes from the data conversion from CSR arrays to dense arrays or from dense arrays to CSR arrays. Take `x[A_i[j]] = A_x[j]` (line 4 of Algorithm 2) as an example, for the read operations of `A_x[j]` and `A_i[j]`, the memory access addresses are contiguous with index `j`. However, for the write operation of `x[A_i[j]]`, the access addresses are not contiguous, because the indexes `A_i[j]` are not contiguous.

For a general case, memory accesses can be written as `X[index(j)]`, where `X` is an array stored in the off-chip memory and `index` is a function of an induction variable `j`. We define *stride* of accesses to `X` is

$$\begin{aligned} stride(j) &= \tfrac{d}{dj}index(j) - 1 \\ &= index(j+1) - index(j) - 1 \end{aligned} \tag{11}$$

The average stride of all the memory accesses in a range r:

$$AvgStride(r) = \frac{1}{N} \sum_{loop \in LOOP} \; \sum_{j \in INDEX(loop)} stride(j) \tag{12}$$

where $LOOP$ denotes all the loops in the range r, $INDEX(loop)$ denotes the index set of loop *loop*, and N is the total number of memory requests in the range. A range can be a piece of kernel execution time, or a region on the matrix plane.

Like the *MemReqDist*, the *MemStrdDist* shows the average stride in each tile on the matrix plane. An example of the *MemStrdDist* is shown in Fig. 9 (for onetone1), using 60×60 tiles.

5.3 Summary of This Section

This section analyzes the impact of the irregular nonzero patterns on the memory access patterns. The above discussion only uses an example of onetone1, some other figures are shown at http://nicslu.weebly.com/special.html. We do see similar pictures for these sparse matrices. From the analysis, some unique characteristics of sparse LU factorization are obtained.

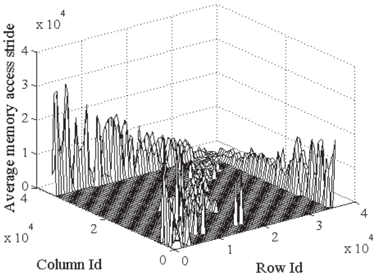


Figure 9: *MemStrdDist* of onetone1.

From Fig. 8, the positions that is near the diagonal and the bottom/right borders have many memory requests; especially at the right-bottom corner, it has abundant memory requests. In contrast, the memory access stride is very low at the right-bottom corner but higher at other positions. This situation reveals that at the right-bottom corner, the nonzeros are much denser than at other positions, which is also caused by the unevenly distributed nonzeros.

The above conclusion suggests us that sparse LU factorization should be separated into two parts to implement optimization strategies. Since the right-bottom corner has a dense block, this block should be factorized individually using dense algorithms rather than sparse algorithms; and for the rest of the matrix, sparse algorithms are still suitable.

6. MEMORY ACCESS OPTIMIZATION AND EXPERIMENTAL RESULTS

6.1 Experiment Setup

The experiments are implemented on an NVIDIA GTX580 GPU, which has a peak performance of 200 Gflop/s (for double-precision). The GPU code is programmed with CUDA (compute capability 2.0) [25]. The proposed approach is compared with PARDISO [5], which is executed on an i7-3770K CPU (4 cores). Seventeen matrices from University of Florida Sparse Matrix Collection [26] are used to evaluate our approaches.

6.2 Crisscross Blocked LU Factorization

The findings in Section 4 and Section 5 reveal that in sparse LU factorization, the right-bottom corner has a dense block, which has smaller memory access stride and more memory requests than other positions. The dense block should be factorized individually to reduce memory requests. Consequently, a blocked LU factorization algorithm [27] is used. Matrix $\mathbf{A}$ is divided into 4 blocks and the LU factors are also divided:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11} & \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} \\ & \mathbf{U}_{22} \end{bmatrix} \tag{13}$$

Since the matrix is divided into fixed 4 blocks, we call this algorithm *crisscross blocked LU factorization*. The factorization process is like this:

$$\begin{aligned} \mathbf{A}_{11} &= \mathbf{L}_{11}\mathbf{U}_{11} \\ \mathbf{L}_{21} &= \mathbf{A}_{21}\mathbf{U}_{11}^{-1} \\ \mathbf{U}_{12} &= \mathbf{L}_{11}^{-1}\mathbf{A}_{12} \\ \mathbf{S} &= \mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{U}_{12} \\ \mathbf{S} &= \mathbf{L}_{22}\mathbf{U}_{22} \end{aligned} \tag{14}$$

$\mathbf{L}_{21} = \mathbf{A}_{21}\mathbf{U}_{11}^{-1}$ and $\mathbf{U}_{12} = \mathbf{L}_{11}^{-1}\mathbf{A}_{12}$ need to explicitly calculate the sparse inverses of $\mathbf{L}$ and $\mathbf{U}$, which is expensive to be implemented on both CPUs and GPUs. We propose an equivalent algorithm that is suitable for GPU implementation, as shown in Fig. 10.

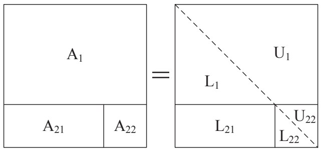


Figure 10: Blocked LU factorization that is suitable for GPU implementation.

Then the factorization process is like this:

$$\begin{aligned} \mathbf{A}_{1} &= \mathbf{L}_{1}\mathbf{U}_{1} \\ \mathbf{L}_{21} &= \mathbf{A}_{21}\mathbf{U}_{1}(:,1:p)^{-1} \\ \mathbf{S} &= \mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{U}_{1}(:,p:n) \\ \mathbf{S} &= \mathbf{L}_{22}\mathbf{U}_{22} \end{aligned} \tag{15}$$

where p is the division point, $\mathbf{U}_1(:,1:p)$ is the part of $\mathbf{U}_1$ that is with all rows and columns 1∼ p. The first step ($\mathbf{A}_1 = \mathbf{L}_1\mathbf{U}_1$) is performed by the row-based up-looking algorithm from row 1 to row p, we call it *partial factorization (ParFact)*. The second step ($\mathbf{L}_{21} = \mathbf{A}_{21}\mathbf{U}_1(:,1:p)^{-1}$) can be also calculated by a up-looking-like algorithm, as shown in Algorithm 3, we call it *complement factorization (ComFact)*. The last two steps are called *sparse multiplication (SpMul)* and *dense factorization (DenFact)*. The dense block is factorized by a CPU/GPU hybrid method which is similar to an existing method [28].

Algorithm 3 Solving $\mathbf{L}_{21} = \mathbf{A}_{21}\mathbf{U}_1(:,1:p)^{-1}$.

```
for $i = p : n$ do
    $\mathbf{x} = \mathbf{A}_{21}(i-p,:)$;
    for $j = 1 : p$ do
        $\mathbf{x}(j+1:p) -= \mathbf{x}(j) \times \mathbf{U}_1(j, j+1:p)$;
    end for
    $\mathbf{L}_{21}(i-p, 1:p) = \mathbf{x}(1:p)$;
end for
```

Please note the differences between our approach and the existing supernodal or multifrontal algorithms. Those algorithms gather nonzeros to dense subblocks according to the specific nonzero distribution. Different matrices lead to different implementations. They cannot explore common features. In contrast, our work explores the common features in sparse LU factorization, which are independent with specific matrices.

6.2.1 *Selection of the Division Point*

The performance of the crisscross blocked method greatly depends on the division point. Apparently, we should make the right-bottom block as large as possible on the condition that dense LU factorization has much larger Gflop/s than the sparse algorithm, especially for GPU implementation. On the other hand, larger right-bottom block leads to more explicit zeros filled in the block, which increases computations. So there is a tradeoff to choose the division point. In the following, we propose a simple but effective method to select the optimal division point.

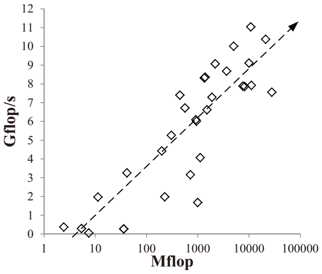


Figure 11: Performance of sparse LU factorization on NVIDIA GTX580 (double-precision). Mflop=million floating-point operations.

For sparse LU factorization, we find that the achieved GPU performance has an approximate logarithmic relation with the number of floating-point operations (flop), as shown in Fig. 11. The fitted function is

$$\text{sparse\_performance}(flop) = 2.535 \times log_{10}(Mflop) - 1.468 \tag{16}$$

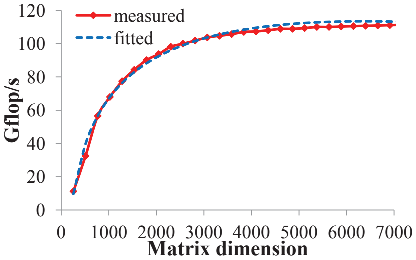


Figure 12: Performance of dense LU factorization on NVIDIA GTX580 (double-precision).

The performance of dense LU factorization on NVIDIA GTX580 is shown in Fig. 12. The performance can be fitted by a logarithmic function as follows.

$$\text{dense\_performance}(k) = 31.93 \times log_2(k) - \frac{k}{138.23} - 243.92 \tag{17}$$

where k is the dimension of the dense matrix.

The predicted factorization time can be expressed as

$$\begin{aligned} &\text{predicted factorization time} = \\ &\frac{\text{sparse\_flop}(p)}{\text{sparse\_performance(flop)}} + \frac{\text{dense\_flop}(n-p)}{\text{dense\_performance}(n-p)} \end{aligned} \tag{18}$$

where sparse_flop(p) is the number of flop of the sparse parts with division point p, which is obtained from the structure of $\mathbf{L}$ and $\mathbf{U}$, and dense_flop(k) is the number of flop of dense LU factorization with dimension k, which is calculated by

$$\text{dense\_flop}(k) = \frac{2}{3}k^3 - \frac{1}{2}k^2 - \frac{1}{6}k \tag{19}$$

A simple exhaustive method on Eq. (18) can find the optimal division point for each matrix. This method can be also implemented on other platforms by re-fitting Eq. (16) and (17).

Table 1: Comparison between unblocked and blocked LU factorization.

Matrix	factorization time		Gflop/s		bandwidth (GB/s)		speedup	speedup vs. PARDISO	dense block dimension	padding %[a]
	unblocked	blocked	unblocked	blocked	unblocked	blocked				
circuit matrices										
asic_100k	0.273	0.235	4.07	4.73	57.2	66.4	1.16	0.79	1280	23.5
asic_100ks	0.166	0.094	8.38	14.86	117.6	208.6	1.77	1.01	1280	28.0
asic_320ks	0.168	0.168	8.13	8.13	114.2	114.2	1.00	3.51	256	98.9
asic_680ks	0.150	0.079	6.11	11.65	85.9	163.7	1.91	69.80	1408	38.3
ckt11752_dc_1	0.058	0.052	5.26	5.85	74.1	82.4	1.11	1.00	768	39.9
g2_circuit	1.094	0.778	9.17	12.89	128.6	180.8	1.41	0.47	2304	32.7
onetone1	0.163	0.065	8.24	20.54	115.6	288.1	2.49	2.22	1664	35.8
onetone2	0.045	0.026	4.42	7.49	62.3	105.4	1.69	2.02	1280	63.2
twotone	0.983	0.383	11.05	28.35	154.8	397.3	2.57	2.89	3584	40.3
average							**1.68**	**2.20**[b]		
non-circuit matrices										
zhao1	0.420	0.232	8.68	15.72	121.9	220.8	1.81	0.38	2048	36.9
sme3dc	1.403	1.107	7.92	10.03	111.2	140.9	1.27	0.39	2048	29.8
xenon1	2.034	1.561	10.38	13.53	145.7	189.8	1.30	0.21	2944	37.0
denormal	0.501	0.396	10.01	12.67	140.8	178.3	1.27	0.52	2048	39.5
thermomech_dm	0.260	0.257	7.29	7.37	103.3	104.4	1.01	0.72	896	45.1
thermomech_dk	0.970	0.850	7.88	8.99	111.0	126.6	1.14	0.34	1792	44.5
thermomech_tc	0.157	0.148	6.01	6.38	85.1	90.4	1.06	0.66	896	45.1
helm2d03	3.704	2.650	7.56	10.56	106.1	148.3	1.40	0.18	2432	18.8
average							**1.28**	**0.42**		

[a] percentage of the explicit zeros filled in the right-bottom dense block

[b] this value is the geometric mean, other average values are the arithmetic mean.

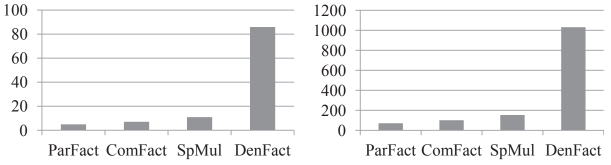
6.2.2 *Experimental Results*

Table 1 shows the performance of the proposed blocked LU factorization algorithm, for both circuit matrices and non-circuit matrices. The blocked approach is compared with the unblocked GPU implementation [9] and the multi-threaded solver PARDISO [5] (using 4 threads).

For circuit matrices, the blocked approach attains 68% performance gain compared with the unblocked approach. In addition, the blocked GPU implementation is on average 2.2× faster than 4-threaded PARDISO. Consequently, the proposed blocked LU factorization approach on GPUs is efficient for circuit matrices.

For non-circuit matrices, though the blocked approach is also faster than the unblocked approach, it is much slower than 4-threaded PARDISO. PARDISO is supernode-based, which uses BLAS to compute dense subblocks during sparse LU factorization, and non-circuit sparse matrices are generally denser than circuit matrices, so PARDISO is efficient for non-circuit sparse matrices.

The comparison between circuit matrices and non-circuit matrices reveals that for circuit matrices, since they are too sparse to form dense subblocks, dense kernel-based algorithms, such as PARDISO, are inefficient. However, we have proved there is still a dense block in the right-bottom corner of circuit matrices, and the dense block should use dense algorithms and other parts should use sparse algorithms.



(a) Gflop/s. (b) Bandwidth (GB/s).

Figure 13: Gflop/s and bandwidth of each step in blocked LU factorization, for onetone1.

Fig. 13 shows Gflop/s and bandwidth of each step in blocked LU factorization, for onetone1. It shows that the dense LU factorization step has much higher Gflop/s and bandwidth than other sparse steps on GPUs. The achieved memory bandwidth of dense LU factorization achieves more than 1000 GB/s, which greatly exceeds the theoretic peak bandwidth of the global memory of NVIDIA GTX580 (192.4 GB/s). This is mainly because of the shared memory used in dense LU factorization. The results reveal that the proposed blocked LU factorization greatly improves the memory performance and increases the overall performance on GPUs. The performance of dense LU factorization is 70∼100 Gflop/s for different matrices, which achieves about half of the peak performance of NVIDIA GTX580 (200 Gflop/s). The overall performance can be further increased if the performance of dense LU factorization can be further increased, since the performance of dense LU factorization is still far away from the peak performance of NVIDIA GTX580.

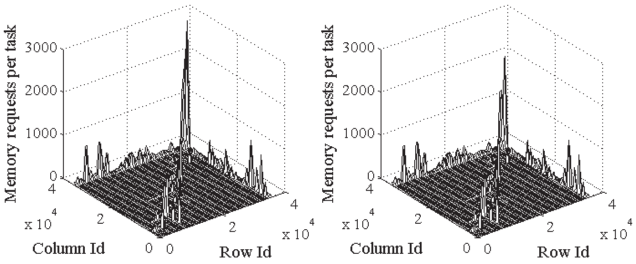


Figure 14: *MemReqDists* of unblocked and blocked LU factorization, for onetone1.

Fig. 14 shows the *MemReqDists* of unblocked and blocked LU factorization, for onetone1. It reveals that the number of memory requests is reduced in the right-bottom block when it is factorized using dense algorithm. In addition, the memory access stride is exactly 0 in the right-bottom block, since there is no CSR accesses in dense LU factorization.

7. CONCLUSIONS

The sparse matrix solver is critical in circuit simulators. The performance of existing GPU implementations is constrained by the irregularities of sparse matrices. This work analyzed the irregular nonzero patterns and irregular memory access patterns in sparse LU factorization and explored the common features. Based on the analysis conclusions, a crisscross blocked implementation was proposed on GPUs. The proposed approach is on average 1.68× faster than the unblocked method and 2.2× faster than 4-threaded PARDISO, for circuit matrices.

Acknowledgements

This work was supported by 973 project 2013CB329000, National Science and Technology Major Project (2011ZX01035-001-001-002, 2013ZX03003013-003) and National Natural Science Foundation of China (No.61373026, 61261160501, 61028006), Tsinghua University Initiative Scientific Research Program, and Tsinghua National Laboratory for Information Science and Technology.

8. REFERENCES

[1] L. W. Nagel. *SPICE 2: A computer program to stimulate semiconductor circuits*. PhD thesis, University of California, Berkeley, 1975.

[2] J. J. Dongarra, Jermey Du Cruz, Sven Hammerling, and I. S. Duff. Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Softw.*, 16(1):18–28, March 1990.

[3] Timothy A. Davis and Ekanathan Palamadai Natarajan. Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems. *ACM Trans. Math. Softw.*, 37(3):36:1–36:17, September 2010.

[4] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. *SIAM J. Matrix Anal. Appl.*, 20(4):915–952, July 1999.

[5] Olaf Schenk and Klaus Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Gener. Comput. Syst.*, 20(3):475–487, April 2004.

[6] Xiaoming Chen, Yu Wang, and Huazhong Yang. An adaptive LU factorization algorithm for parallel circuit simulation. In *ASP-DAC*, pages 359 –364, 30 2012-feb. 2 2012.

[7] Xiaoming Chen, Wei Wu, Yu Wang, Hao Yu, and Huazhong Yang. An EScheduler-Based Data Dependence Analysis and Task Scheduling for Parallel Circuit Simulation. *TCAS-II*, 58(10):702 –706, oct. 2011.

[8] X. Chen, Y. Wang, and H. Yang. NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation. *TCAD*, 32(2):261 –274, feb. 2013.

[9] Ling Ren, Xiaoming Chen, Yu Wang, Chenxi Zhang, and Huazhong Yang. Sparse LU factorization for parallel circuit simulation on GPU. In *DAC*, DAC '12, pages 1125–1130, New York, NY, USA, 2012. ACM.

[10] Byunghyun Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *TPDS*, 22(1):105 –118, jan. 2011.

[11] M.R. Hugues and S.G. Petiton. Sparse Matrix Formats Evaluation and Optimization on a GPU. In *HPCC*, pages 122 –129, sept. 2010.

[12] Shuai Che, J.W. Sheaffer, and K. Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *SC*, pages 1 –11, nov. 2011.

[13] Naga K. Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha. A memory model for scientific algorithms on graphics processors. In *SC*, SC '06, New York, NY, USA, 2006. ACM.

[14] Jonathan Weinberg, Michael O. McCracken, Erich Strohmaier, and Allan Snavely. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *SC*, SC '05, pages 50–61, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Robert F. Lucas, Gene Wagenbreth, Dan M. Davis, and Roger Grimes. Multifrontal computations on GPUs and their multi-core hosts. In *VECPAR*, VECPAR'10, pages 71–82, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] T. George, V. Saxena, A. Gupta, A. Singh, and A.R. Choudhury. Multifrontal Factorization of Sparse SPD Matrices on GPUs. In *IPDPS*, pages 372–383, 2011.

[17] Chenhan D. Yu, Weichung Wang, and Dan'l Pierce. A CPU-GPU hybrid approach for the unsymmetric multifrontal method. *Parallel Comput.*, 37(12):759–770, December 2011.

[18] M. Christen, O. Schenk, and H. Burkhart. General-purpose sparse matrix building blocks using the NVIDIA CUDA technology platform. In *First Workshop on General Purpose Processing on Graphics Processing Units*. Citeseer, 2007.

[19] Géraud P Krawezik and Gene Poole. Accelerating the ANSYS direct sparse solver with GPUs. In *SAAHPC*, july 2009.

[20] J. R. Gilbert and T Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput.*, 9(5):862–874, 1988.

[21] NVIDIA Corporation. CUBLAS.

[22] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Software, Environments, Tools. Society for Industrial and Applied Mathematics, 1987.

[23] John R. Gilbert, Esmond G. Ng, and G. Ng. Predicting Structure In Nonsymmetric Sparse Matrix Factorizations. In *Graph Theory and Sparse Matrix Computation*, pages 107–139. Springer-Verlag, 1992.

[24] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. An Approximate Minimum Degree Ordering Algorithm. *SIAM J. Matrix Anal. Appl.*, 17(4):886–905, October 1996.

[25] NVIDIA Corporation. NVIDIA CUDA C Programming Guide, 2013.

[26] Timothy A. Davis and Yifan Hu. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[27] Jürgen Garloff. Block methods for the solution of linear interval equations. *SIAM J. Matrix Anal. Appl.*, 11(1):89–106, January 1990.

[28] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *IPDPSW*, pages 1 –8, april 2010.