SPARSE LU FACTORIZATION FOR LARGE CIRCUIT MATRICES ON HETEROGENOUS PARALLEL COMPUTING PLATFORMS

A Thesis

by

ADITYA SANJAY BELSARE

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2014

Major Subject: Computer Engineering

ABSTRACT

Direct sparse solvers are traditionally known to be robust, yet difficult to parallelize. In the context of circuit simulators, they present an important bottleneck where the key steps of LU factorization and forward-backward substitution are repeatedly performed to reach the solution. Limited speedups have been obtained on multi-core CPUs as well as GPUs owing to the strong data dependency in these steps. With the advent of many-core coprocessors like the Intel Xeon Phi with fewer yet powerful cores and wider vector units, sparse LU factorization can be optimized for higher speedups compared to traditional LU decomposition methods like the Gilbert Peierl's algorithm. In this thesis, we first establish Sparse Compressed Row (CSR) as the preferred data structure amongst other popular sparse matrix representations for parallelizing sparse circuit solvers, irrespective of the architecture used. Next, we propose and implement a sparse circuit solver suited for parallelization on both the Nvidia GPU and Intel Xeon Phi platform, which is amenable to vectorization and takes advantage of hardware support, if any, for gather-scatter operations. Finally, we analyze our implementation on multi-core, SIMD and SIMT architectures namely Intel Xeon CPU, Intel Xeon Phi coprocessor and an Nvidia GPU respectively, each using different programming models suited for the respective platform to determine the architecture best suited for parallelizing direct sparse matrix solvers. Our parallel sparse LU factorization achieves an average speedup of 7.18x on the Xeon Phi and 2.75x in case of the GPU implementation on GTX 680 over an Intel 4-core i7 CPU, which is up to 13x faster than a single threaded implementation.

To my mother

ACKNOWLEDGEMENTS

I would like to thank my advisor Professor Steve Liu for his invaluable and continuous support in making this work possible. I thank him for the opportunity and freedom he gave me for working on an entirely new topic within his research group, complete with sponsoring the newest hardware for the research. I would also like to thank Dr. Desmond Kirkpatrick and Dr. Kanupriya Gulati from Strategic CAD Labs, Intel Corporation, Hillsboro, Oregon for first introducing me to the wonderful world of sparse matrices and linear systems of equations during my internship there. A special thanks to Dr. Jiang Hu for serving as the co-chair of my committee and Dr. Paul Gratz for serving on my committee.

I would like to thank my loved ones, who have supported me throughout the entire process.

NOMENCLATURE

CSR	Compressed Sparse Row
CSC	Compressed Sparse Column
COO	Coordinate Format
N,n	Order of Matrix
NNZ	Number of Non-zeros
SpMV	Sparse Matrix Vector Multiplication
BTF	Block Triangular Form
AMD	Approximate Minimum Degree

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Circuit simulators dealing with non-linear equations and models face a fundamental bottleneck of repeatedly solving $Ax = b$, a linear system of equations. Modern circuit simulators carry out more detailed simulation for a circuit than ever before. A circuit simulator is commonly used for linear and non-linear DC analysis, linear AC analysis, small signal AC analysis, linear and large signal transient analysis, as also for other functions like pole/zero analysis and noise analysis. This has resulted in long simulation time of the order of days or even weeks on modern CPUs, as the size of circuits continue to increase attempting to keep up with the Moore's law. While iterative methods present an interesting approach for circuit simulators since they can be massively parallelized, however, they suffer from the obvious disadvantages of non-convergence of solutions often resulting in variable, large number of iterations. Direct methods, in contrast, are stable and remain a popular choice for circuit simulators today where reasonable precision is expected in the solutions.

Circuit simulators operate by converting non-linear models to linear equations for a single point, and starting with an initial guess, repeatedly solve the linear models until convergence is reached. This entire process constitutes a Newton-Raphson loop [1], which is outlined in figure 1.1.

The nodal equations comprise a linear system of equations of the form -

$$\begin{bmatrix} G_{11} & G_{12} & \dots \\ G_{21} & \ddots & \\ \vdots & & G_{nn} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} I_1 \\ I_2 \\ \vdots \end{bmatrix} \tag{1.1}$$

where V_i, I_i and G_{ij} denote the node voltage, current and conductance respec-

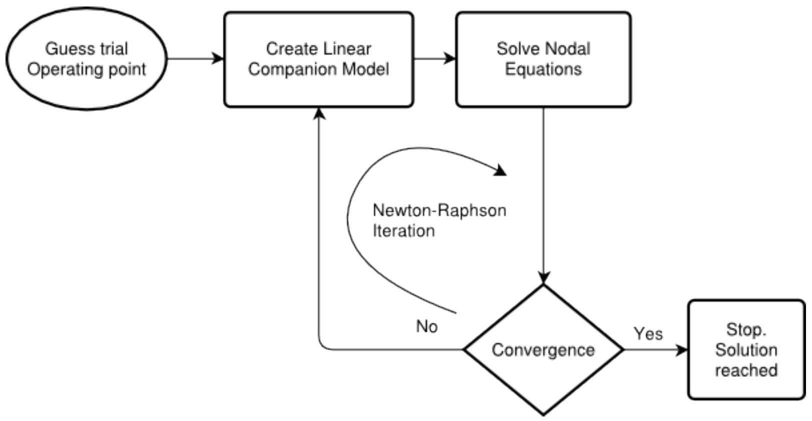

Figure 1.1: Basic steps in a circuit simulator

tively. The conductance matrix (G) and the accompanying linear system of equations is repeatedly solved in the Newton-Raphson loop. The conductance matrix is very sparse, with typically up to 10 non-zeros per row, corresponding to each node. It is also very large, with as many rows as the number of nodes in the circuit, which is of the order of millions for modern VLSI designs. Maximum entries are along the diagonal of the matrix, or can be permuted to be so using various ordering methods like AMD [2], COLAMD [3], Reverse Cuthill McKee (RCM) ordering [4] etc.

1.1 Key Steps in Direct Sparse Circuit Solvers

A typical sparse solver involves 4 or more distinct steps -

1. A permutation step that ensures zero-free diagonal and appropriate scaling
2. An ordering step that permutes the rows and columns of the matrix into a form like the Block Triangular Form (BTF) [5], which helps reduce the fill-in of the factors.
3. An analysis step called the symbolic analysis or symbolic factorization that determines the nonzero structures of the factors, and modify the existing data

structure in order to accommodate the additional fill-in.

4. Numerical factorization or LU decomposition that computes the L and U factors using some variant of Gaussian elimination.

5. A forward backward substitution (or block forward backward substitution) step that substitutes the computed values to solve the equation in a final linear step. factors.

There are a variety of algorithms for each step, suited for sparse matrices with different characteristics. For circuit matrices, KLU [5] is a widely used sparse circuit solver; we have used KLU as the baseline for comparison of our results. The key steps in KLU are as below -

1. *Scaling* ($RAx = Rb$). R is a diagonal matrix with scale factors for each row.

2. Asymmetric permutation to Block Triangular Form (BTF) [5]:

   (a) Asymmetric permutation to get zero-free diagonal using maximum traversal (Duff's algorithm [6, 7])

   (b) Symmetric permutation to BTF by finding strongly connected components of the graph. (Duff's and Reid's algorithm [8, 9])

3. Symmetric permutation of each block (fill-in reducing ordering) using Approximate Minimum Degree(AMD) [2], or Column AMD (COLAMD) [3]

4. *Symbolic analysis* - Computing the non-zero pattern of column 'k' of the factors.

5. Numerical factorization and Solve:

   (a) Sparse *LU decomposition* (Numerical factorization) of each block using Gilbert Peierl's algorithm [10] with partial pivoting ($LUx = b$)

(b) *Forward-backward substitution* (Solve) to solve the system using block back substitution and account for the off-diagonal entries.

Scaling, BTF permutation, AMD, Symbolic analysis are static steps and performed only once for circuit matrices while iterating the Newton-Raphson loop. The last step, which includes LU factorization and forward-backward substitution are performed repeatedly until convergence is reached. In essence, speeding up these two steps should form the essential crux when parallelizing and speeding up any modern circuit simulator. In this thesis, we have attempted to parallelize these two steps while using the rest of the steps from KLU to speedup the circuit simulator, as a whole.

1.2 Choice of Sparse Matrix Data Structures for Parallelizing Sparse Circuit Solver

We evaluated three popular formats for parallelizing the two key steps in our sparse circuit solver, namely *LU factorization* and *forward-backward substitution.* These formats are Sparse Compressed Row (CSR), Sparse Compressed Column (CSC), and Coordinate format (COO) [11]. CSR consists of three vectors *vals*, *cols*, and *row_ptr*. *row_ptr* consists of row pointers. It is of length $n + 1$. The start of row k of the input matrix is given by $row\_ptr[k]$. *cols* consists of column indices of the elements. This is a zero based data structure with column indices in the interval $[0, n)$. *vals* consists of the actual numerical values of the elements. As an example, the matrix

$$\begin{bmatrix} 5 & 0 & 0 \\ 4 & 2 & 0 \\ 3 & 1 & 8 \end{bmatrix} \tag{1.2}$$

represented in the CSR format will be:

$row\_ptr = [0, 1, 3, 6]$

$cols = [0, 0, 1, 0, 1, 2]$

$vals = [5, 4, 2, 3, 1, 8]$

The Compressed Sparse Column (CSC) format is analogous to the CSR format, with *col_ptr* storing the row indices of the elements. The above matrix represented in the CSC format will be:

$col\_ptr = [0, 3, 5, 6]$

$rows = [0, 1, 2, 1, 2, 2]$

$vals = [5, 4, 3, 2, 1, 8]$

The Coordinate format has the simplest representation with $(row, col, value)$ tuples listed for each non-zero in the matrix. For our chosen example, the representation in COO format will be:

$rows = [0, 1, 1, 2, 2, 2]$

$cols = [0, 0, 1, 0, 1, 2]$

$vals = [5, 4, 2, 3, 1, 8]$

The storage requirements for the 3 formats are listed in Table 1.1.

Table 1.1: Storage requirements for CSR, CSC and COO sparse matrix representations

Format	Integers	Floating Points
CSR	nnz+n	nnz
CSC	nnz+n	nnz
COO	2nnz	nnz

We use the Sparse Compressed Row (CSR) format for storing and manipulating the sparse circuit matrices in our implementation. With regards to either serial or parallel implementation of LU factorization, all the 3 formats can be implemented in a way to yield nearly identical performance. For the Sparse Matrix-Vector Multiplication (SpMV) operation however, which is a key operation in the forward-backward substitution step when using BTF, we show that the CSR format easily outperforms the other two formats. The reason behind this, as investigated later, is that the CSR format does not require any synchronization between threads while adding up multiplication results from each thread. As a result, we have used the CSR format as our preferred choice for parallelizing LU factorization and forward-backward substitution, and not CSC, which is adopted by KLU.

The remainder of this thesis is organized as follows. Section 2 describes the elementary theory used in any sparse matrix solver. Section 3 describes our parallel Sparse Compressed Row (CSR) based LU factorization, while Section 4 reports experimental results, including results from comparisons with KLU against our GPU and multi-core implementations on CPU and Intel Xeon Phi coprocessor. Section 5 surveys the existing work done on sparse linear solvers, particularly for circuit matrices. Finally in Section 6, we make concluding comments and discuss further work that needs to be done in this area.

2. COMPONENTS OF A SPARSE MATRIX SOLVER

Any sparse matrix solver invokes a number of steps before performing the actual Gaussian elimination and solve operations. This section describes in detail the key operations outlined earlier, like the Block Triangular Form (BTF) permutation and ordering methods like the Approximate Minimum Degree (AMD) ordering. Specifically, we here only describe the steps prior to Symbolic analysis in this section, while the next section describes the remaining steps that we have implemented as part of our solver.

2.1 Pivoting

In matrix computations, a pivot element is an element in each row of the matrix, which is most distant from zero. The pivot element is essential for performing varied computations in matrix algorithms like Gaussian elimination, Simplex algorithm etc. The process of finding the pivot element is called pivoting. For matrix computations, there are two important considerations for the pivot element -

(i) The pivot should not be zero, else Gaussian elimination fails.

(ii) The pivot element should generally have a large absolute value, for better numerical stability.

We illustrate these considerations with two examples. Consider a simple 2*2 matrix,

$$A = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \tag{2.1}$$

When we apply Gaussian elimination to find L and U matrices, we compute the U term $-a_{21}/a_{11}$ $(2 \div 0)$, multiply it with first row and add to second row, to reduce

a_{21} to zero. Clearly, this step fails for the above case since a_{11} is zero. The same is equivalently true when any diagonal element a_{ii} is zero.

Now consider a case when the pivot is not absolute zero, but very close.

$$\begin{bmatrix} 0.003 & 59.14 \\ 5.291 & -6.130 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 59.17 \\ 46.78 \end{bmatrix} \tag{2.2}$$

The system has exact solution of $x_1 = 10.00$ and $x_2 = 1.000$, however, when Gaussian elimination and forward-backward substitution is performed using a 4-digit decimal precision, the small value of a_{11} causes small round-off errors to be propagated. We obtain an approximate solution of $x_1 \approx 9873.3$ and $x_2 \approx 4$, which is way off the accurate solution. A simple row interchange here between the two rows fixes the problem. The row interchange makes the original a_{11} a non-pivot element. a_{21} is the new pivot element, which is sufficiently distant from zero. The problem lied with the multiplier, which earlier obtained from a small pivot element was huge, and when added to the small element a_{22}, suppressed it completely.

Pivoting can be of two forms -

(i) Partial pivoting

(ii) Complete pivoting

In *partial pivoting*, the algorithm selects the largest absolute value from each column as the pivot element. In *scaled partial pivoting*, the largest element from each row is selected as the pivot element. Partial pivoting and scaled partial pivoting both add a time complexity of $O(n^2)$ to the Gaussian elimination for dense matrices. The time cost for sparse matrices cannot be quantified in terms of n as the number of non-zeros in each row is variable, however, the cost is substantially less for sparse matrices.

In *complete pivoting*, the entire matrix is search (and subsequently permuted) to allow the largest element from the matrix to be the pivot element for first row, second largest element to be the pivot for the second row and so on. This is substantially more expensive than partial pivoting, with a time complexity of $O(n^3)$ to calculate the pivots for a dense matrix. Complete pivoting is usually not necessary to ensure numerical stability, and due to the additional computations it introduces, we use partial pivoting with diagonal preference in our our implementation.

2.2 Scaling

Pivoting, partial or complete is often insufficient to prevent small elements in the matrix from getting obscured during Gaussian Elimination. The numerical addition between the rows can still assume large differences in the values used for addition, which partial pivoting is unable to overcome. Let us illustrate this with an example.

Consider the system of equations,

$$\begin{bmatrix} 10 & 10^5 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 10^5 \\ 2 \end{bmatrix} \tag{2.3}$$

Applying Gaussian elimination with partial pivoting to the above system, we obtain (after eliminating a_{21}),

$$\begin{bmatrix} 10 & 10^5 \\ 0 & -10^4 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 10^5 \\ -10^4 \end{bmatrix} \tag{2.4}$$

which yields a solution of $x_1 = 1, x_2 = 0$, different from the true solution, $x_1 = 1, x_2 = 1$. Suppose now that we *scale* by dividing the first row by the largest element, 10^5, before performing the Gaussian elimination. The system before applying Gaussian elimination is,

$$\begin{bmatrix} 10^{-4} & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \tag{2.5}$$

This process of dividing each row or column by the largest element in the respective row or column is termed scaling. In column scaling, we divide each column by the largest element in the respective column. After applying partial pivoting, this becomes,

$$\begin{bmatrix} 1 & 1 \\ 10^{-4} & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \tag{2.6}$$

Applying Gaussian elimination, we obtain,

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 - 10^{-4} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 - 10^{-4} \end{bmatrix} \tag{2.7}$$

Solving this system yields $x_1 = 1, x_2 = 1$, which is the true solution. Scaling is optional for many problems where the values are already balanced.

2.3 Block Triangular Form (BTF)

Square sparse matrices can often be reduced to a useful form called the Block Triangular Form (BTF) by simple permutation of rows and columns. BTF allows a good saving of computation effort for problems in linear algebra like solving linear systems of equations and eigenvalue problems.

Consider the linear system $Ax = b$ shown in Eq. 2.8, which has been expressed in the BTF form.

$$\begin{bmatrix} 2 & 9 & 5 & 0 & 0 & 0 \\ 9 & 5 & -3 & 0 & 0 & 0 \\ 1 & 0 & 4 & 0 & 0 & 0 \\ -9 & 4 & 2 & 1 & 2 & -1 \\ -6 & 9 & 1 & 1 & 1 & 3 \\ 9 & 4 & 3 & 2 & 3 & -2 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{bmatrix} \tag{2.8}$$

This is a 6*6 linear system with rows and columns permuted to the BTF form. Notice the top-right 3*3 matrix block which consists of zeros exclusively. The aim of BTF permutation is to arrange blocks of such zero matrices in either the upper triangle (U) or the lower triangle (L) of any given matrix. This linear system can be equivalently written as,

$$\begin{bmatrix} 2 & 9 & 5 \\ 9 & 5 & -3 \\ 1 & 0 & 4 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{2.9}$$

$$\begin{bmatrix} -9 & 4 & 2 \\ -6 & 9 & 1 \\ 9 & 4 & 3 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 1 & 2 & -1 \\ 1 & 1 & 3 \\ 2 & 3 & -2 \end{bmatrix} \times \begin{bmatrix} x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} b_4 \\ b_5 \\ b_6 \end{bmatrix} \tag{2.10}$$

where the matrix and vector terms are defined as follows -

$$A_1 x_1 = b_1 \tag{2.11}$$

$$A_2 x_2 = b_2 - A_{21} x_1 \tag{2.12}$$

Solving the original 6*6 system in Eq. 2.8 is now reduced to solving two smaller 3*3 systems in Eqs. 2.11 and 2.12. Assuming a computation cost of N^3 for solving the original system, the cost is now reduced to roughly $2 * (N/2)^3$, a 4 fold reduction in efforts.

The variant of BTF permutation used in KLU, and for our solver permutes the zeros to the lower triangular blocks. The chief advantages of permuting into the BTF form are -

(i) The part above (or below depending on the type of BTF used) the block diagonal does not require factorization.

(ii) Even though the block diagonals can only be factorized in sequence, they are still independent of each other, meaning each factorization solves only for variables contained in the particular block.

Permutation to the BTF form can be accomplished using the Duff and Reid's algorithm [8] which employs symmetric permutation to a matrix and find the strongly connected components of the matrix graph. The matrix is first represented in an equivalent undirected graph notation with the matrix order N denoting the number vertices and the co-ordinate *(u,v)* representing an edge from vertex *u* to *v*. We introduce a few formal definitions for a directed graph relevant for finding the BTF permutation -

- **Path -** A path from node v_1 to v_k in the graph is a sequence of edges (v_1, v_2), (v_2, v_3),.. (v_{k-1}, v_k)
- **Cycle -** The path in the graph is cyclic if $v_1 = v_k$.
- **Subgraph -** A subset of nodes and all edges that are pairs of nodes belonging to a particular subset.

- **Strongly connected -** A subgraph is strongly connected if there is a path from any of its node v_i to any other v_j.
- **Strong component -** A subgraph has strong components if it is strongly connected and cannot be enlarged to another strongly connected subgraph by adding extra nodes and associated edges.

From the definition of strong component, we can deduce that a node always belongs to a unique strong component. In essence, strong components define the partition of a graph. There should be at least one strong component C_1 such that there is no path from any of its node to any node of another strong component. The remaining strong components $C_2, C_3, \ldots C_k$ may be chosen similarly in a way that there is no path from any node of one strong component to a node of a strong component later in the sequence.

Using a depth first search approach from unvisited nodes in the graph, we can identify these strong components and the corresponding ordering. Using the ordering, we may be able to label nodes of C_1 before those of C_2 and so on; the associated matrix is essentially lower block triangular with blocks corresponding to the strong components. The subgraph associated with each sub-matrix on the diagonal of the block form is strongly connected. The Duff and Reid's algorithm has a time complexity of $O(n + \tau)$, where n is the order of the matrix and τ is the number of off diagonal non-zeros in the original matrix.

2.4 Approximate Minimum Degree (AMD) Ordering

The fill-in of a matrix is defined as a non-zero position (i, j) in the factorization of the matrix which was earlier a zero, in the original matrix, i.e. a fill-in occurs if $L_{ij} \neq 0$, where $A_{ij} = 0$. The purpose of a fill reducing ordering, like the approximate minimum degree ordering (AMD) [2] used in KLU is to generate a permutation P

of the symmetric matrix A, such that the fill-in in the factorization of PAP^T is minimal. If the matrix A is unsymmetric, we can use the transformation $A + A^T$ instead for the factorization. Finding the best ordering for a matrix is an NP-complete problem, with several heuristic methods available. AMD has been known to yield better ordering for circuit matrices.

The fill-reducing ordering is carried out on each of the block diagonal matrices generated after the BTF permutation on the original matrix. Any ordering algorithm like the AMD only takes into account the non-zero positions or the structure of the matrix, without regard to the actual numerical values present in those positions. It generates a permutation in a way such that the node with minimum degree is eliminated in each step of Gaussian elimination. We illustrate this with a small example.

Figure 2.1 shows the initial configuration of non-zeros in an unordered symmetric matrix, and the resultant configuration after one step of Gaussian elimination, when a_{11} is assumed to be the pivot element. The matrices in this figure can be represented as undirected graphs $G(V, E)$ shown in figure 2.2 with vertices corresponding to row/column indices. An edge $i \to j$ exists in G if $A_{ij} \neq 0$.

$$\begin{bmatrix} * & * & & * & * \\ * & * & & * & \\ & & * & & \\ * & * & & & \\ * & & & & * \end{bmatrix} \to \begin{bmatrix} * & * & & * & * \\ & * & & * & * \\ & & * & & \\ & * & & * & * \\ & * & & * & * \end{bmatrix}$$

Figure 2.1: AMD: Non-zero pattern initially and after one step of Gaussian elimination. Adapted from [2].

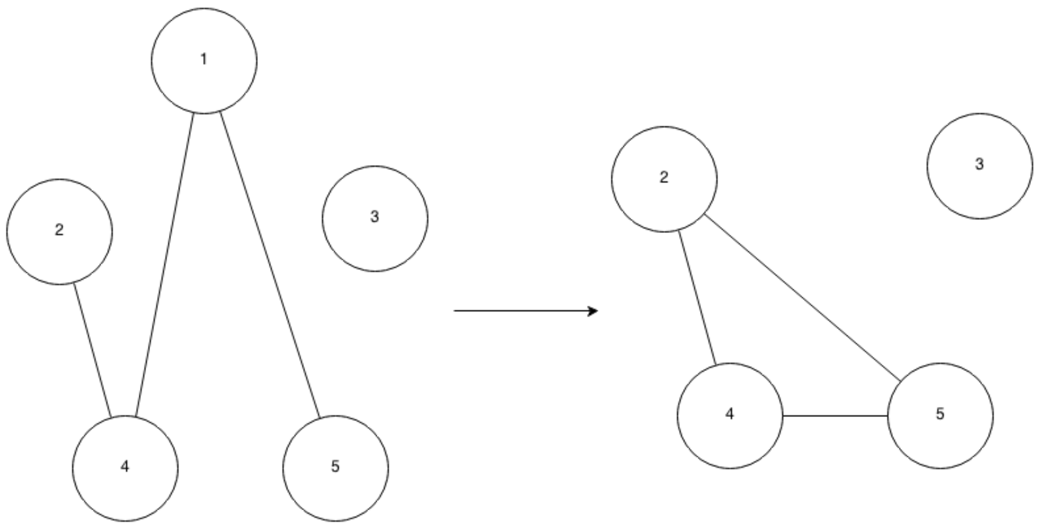

Figure 2.2: AMD: Matrices represented as undirected graphs

In graph representation, the first step of Gaussian elimination using a_{11} as the pivot element is equivalent to removing node 1 from the undirected graph and adding edges to connect all nodes adjacent to 1. Elimination creates a group of nodes adjacent to the pivot element, with the number of fill-ins in the new matrix equal to the number of edges added in the group of nodes adjacent to the node eliminated. Node 1 is not necessarily the best choice of pivot element and might not result in minimum number of nodes in the group adjacent to eliminated node. The aim of AMD is to find such node with minimum degree or pivot element in each step, which results in the node with minimum degree being eliminated.

2.5 Condition Number

Condition number measures the the variation in the output of a function for a small change in the input argument. Condition number gives an idea of the sensitivity of a function with regard to the errors in the input. In the context of linear system of equations $Ax = b$, the condition number gives a bound on how inaccurate the solution x would be after approximating. Condition number is a key factor that determines the stability of an algorithm.

A problem is **ill conditioned** if a small relative error in data results in a large relative error in solution; this is irrespective of the algorithm employed. A problem is **well conditioned** if a small relative error in data does not result in a large relative error in solution. If the problem is ill conditioned, even a very stable algorithm coupled with tricks like partial pivoting, scaling, fill reducing ordering cannot ensure a reasonably accurate solution to the problem. We now define ill conditioned and well conditioned matrix mathematically.

Suppose $X(d)$ is the solution to a problem X for an input d. Suppose now the problem is subjected to a small perturbation δd in the input d. The relative output error can be given by,

$$\frac{|X(d+\delta d) - X(d)|}{|X(d)|}$$

and the relative input error can be given by,

$$\frac{|\delta d|}{|d|}$$

The problem is ill conditioned if $\frac{|X(d+\delta d)-X(d)|}{|X(d)|} > \frac{|\delta d|}{|d|}$, well conditioned otherwise. The **condition number** for system of type $Ax = b$ is defined as,

$$Cond(A) = \|A\| \left\|A^{-1}\right\| \tag{2.13}$$

The system $Ax = b$ is ill conditioned if $Cond(A)$ is a large number; well conditioned otherwise. A problem with computing $Cond(A)$ is immediately evident from Eqn. 2.13, as calculating inverse of a matrix is an expensive operation, even more than solving the system $Ax = b$ itself. We use KLU's approach for calculating the condition number, which is based on Hager's algorithm [12]. Hager proposed an optimization approach for estimating 1-norm condition number $\|A^{-1}\|_1$ given as,

$$\|A\|_1 = max\frac{\|Ax\|_1}{\|x\|_1}$$

The computation of $\|A^{-1}\|_1$ involves computing $A^{-1}x$ and $(A^{-1})^T x$ which is equivalent to solving $Ax = b$ and $A^T x = b$; this in turn can be solved using KLU. KLU's condition number estimator is based on Higham's refinement of Hager's algorithm [13].

3. IMPLEMENTING THE SPARSE MATRIX SOLVER

In this section, we present our sparse LU factorization method in detail. LU factorization is preceded by numerous preprocessing steps like scaling, ensuring a zero-free diagonal, Block Triangular Form (BTF) permutation, Approximate Minimum Degree (AMD) ordering, which we have used from the KLU. We have implemented a Sparse Compressed Row (CSR) based symbolic analysis phase which aids repeated LU factorizations in our sparse circuit matrix solver. Figure 3.1 summarizes the overall implementation flow for both KLU as well as our solver, using a parallel LU factorization.

3.1 Preprocessing

The preprocessing consists of 4 steps:

1. **Scaling:** Scaling is the process of balancing out the numerical enormity or obscurity on each row or column. For our CSR based implementation, we divide each row of the matrix by the largest element in that row.

2. **Zero free diagonal:** Gaussian elimination fails if there are zeros along the diagonal in the matrix. Hence, we employ KLU's unsymmetric ordering to ensure a zero free diagonal, which internally uses Duff's algorithm [6, 7].

3. **Block Triangular Form (BTF) Permutation:** By appropriately permuting the rows and columns of a matrix, it can be converted into a block triangular form as shown in figure 3.2. BTF restricts factorization to only the diagonal blocks; the part of the matrix below the block diagonal is zeros and requires no factorization. The diagonal blocks which are independent of each other require

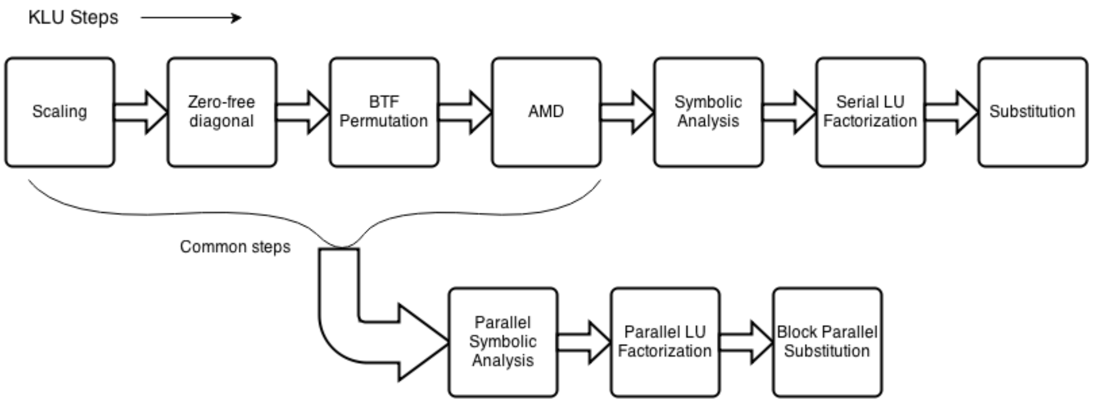

Figure 3.1: Implementation flow for KLU as well as our solver, using a parallel LU factorization

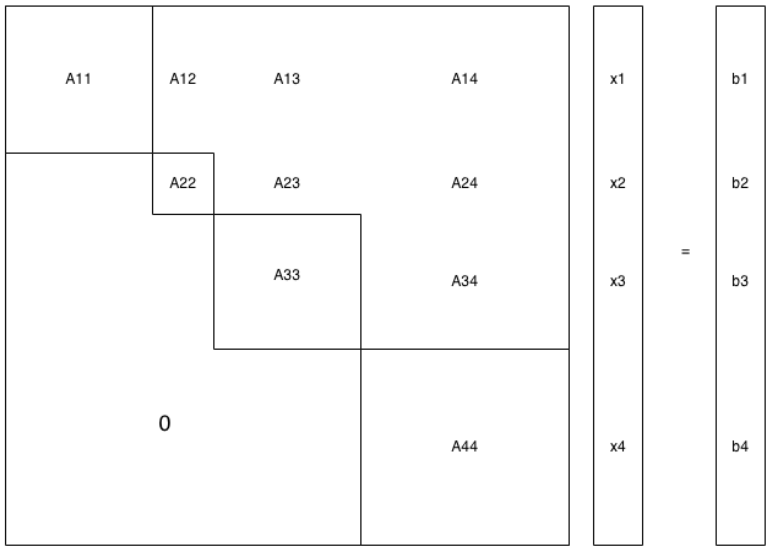

Figure 3.2: Block Triangular Form of a matrix

factorization. The system can be solved for x_i's as:

$$A_{44}x_4 = b_4 \tag{3.1}$$

$$A_{33}x_3 = b_3 - A_{34}x_4 \tag{3.2}$$

and so on. Here, $A_{34}x_4$ is a sparse matrix-vector multiplication (SpMV) operation, which is invoked multiple times for each row during the forward-backward substitution step. SpMV represent the dominant cost in the forward-backward substitutions which are repetitively performed to reach convergence in the Newton-Raphson iteration in a sparse circuit solver. Parallelizing SpMV is therefore important for speeding up block forward-backward substitutions.

4. **Fill-in reducing ordering (AMD):** Of the various ordering schemes for each block matrix resulting from the BTF permutation, AMD gives best results on circuit matrices [5]. We use KLU's Approximate Minimum Degree (AMD) for reducing the fill-in caused by sparse Gaussian Elimination.

3.2 Parallelizing Sparse Matrix Vector Multiplications

As described previously, SpMV operations form the bottleneck in block forward-backward substitutions when the matrix is permuted to the BTF form. To parallelize SpMV, we analyzed three popular formats used for representing sparse matrices - the coordinate form (COO), sparse compressed column (CSC) and sparse compressed row (CSR).

For the COO format, there is no obvious rule to divide amongst threads. Consider a simple example for the COO format where we divide work equally amongst three threads.

$$\begin{bmatrix}1&5&0&7\\4&9&0&0\\0&0&2&3\\0&6&0&8\end{bmatrix} \times \begin{bmatrix}1\\5\\3\\6\end{bmatrix} \Longrightarrow \left\{ \begin{array}{cccc} Row & Col & Val & \\ 0 & 3 & 7 & \\ & & & Thread1 \\ 2 & 2 & 2 & \\ 3 & 1 & 6 & \\ Row & Col & Val & \\ 0 & 1 & 5 & \\ & & & Thread2 \\ 2 & 3 & 3 & \\ 0 & 0 & 1 & \\ Row & Col & Val & \\ 1 & 1 & 9 & \\ & & & Thread3 \\ 3 & 3 & 8 & \\ 1 & 0 & 4 & \end{array}\right.$$

As apparent, each of the 9 individual terms calculated by the 3 threads need to be reduced to 3 terms by addition. There is a reduction across threads, requiring a synchronization operation at the end between the threads.

Consider next the CSC format. The natural way to distribute work amongst threads here is by assigning a column to each thread or a group of threads. Similar to the COO format, terms belonging to the same row are gathered across threads for reduction, necessitating an atomic operation as illustrated in the OpenMP pseudocode in Algorithm 1. The keyword *pragma simd* hints the compiler to vectorize elements from each column array.

Finally, for the CSR format, by assigning a row of the sparse matrix to each thread

Algorithm 1 Multicore SpMV Multiplication - Sparse Compressed Column

```
#pragma omp parallel for
for(i=0; i<length_cols -1; i++)
        #pragma simd
        for(j=cols[i]; j<cols[i+1]; j++)
                #pragma omp atomic
                y[rows[j]] += vals[j] * x[i];
```

or a group of workers, one does not require reduction across threads for adding the results of multiplication. This is particularly useful for a many-core architecture like the Xeon Phi, where thread communication cost is higher than a SIMT architecture like the GPU. An OpenMP pseudocode illustrating CSR based SpMV is shown in Algorithm 2.

Algorithm 2 Multicore SpMV Multiplication - Sparse Compressed Row

```
#pragma omp parallel for
for(i=0; i<n; i++)
        #pragma simd
        for(j=rows[i]; j<rows[i+1]; j++)
                y[i] += vals[j] * x[cols[j]];
```

Therefore, we have chosen a row based format (CSR) for our parallel sparse solver implementation instead of a column based format like that used in KLU.

3.3 Symbolic Analysis

Following the preprocessing step and prior to the actual LU factorization, we perform symbolic analysis to determine the existing and future non-zero pattern of the sparse matrix when the LU decomposition is performed. Symbolic analysis is a static static step performed only once and ensures that the new non-zero positions created as a result of subtracting rows during LU decomposition are accounted for in the CSR data structure. We conveniently break down symbolic analysis as a series of steps described in the following sections, which would help speed up the actual LU factorization.

3.3.1 Column Non-zero Pattern Calculation

This linear operation scans the CSR data structure and records the row position of each non-zero into a column, i.e. the element positions are recorded columnwise. Consider a 5×5 matrix as shown with non-zeros indicated with *:

$$\begin{bmatrix} (*) & (*) & (0) & (0) & (0) \\ \{*\} & * & 0 & * & * \\ \{0\} & * & * & * & 0 \\ \{*\} & * & * & * & 0 \\ \{0\} & 0 & 0 & * & * \end{bmatrix}$$

The pivot row is denoted with round braces while the pivot column is shown with curly braces. During LU factorization, we seek to avoid redundant row subtractions between the pivot row and current row, if the pivot element in the current row is a zero, for example rows 2 and 4 here. However, for a row based format like the CSR with unsorted column indices, it is difficult to directly index a particular non-zero from a column for a given row, without traversing the entire row. Hence, storing all non-zeros in a given column helps directly reference the non-zeros for which the given row should be subtracted from pivot row during LU factorization. Algorithm 3 illustrates the pseudocode for extracting the non-zeros in a columnwise fashion:

Algorithm 3 Column non-zero pattern calculation

```
k = [0] * n
for each row in (0: n) in parallel, do:
    for each col in row:
        nz_cols[ csr_column[row][col] ][ k[csr_column[row][col]]++ ] = col
```

3.3.2 Row Non-zero Pattern Calculation

This step performs an in-place symbolic analysis to determine the row non-zero pattern of the given sparse matrix. For a given pivot row and current row for which the additional non-zeros are to be found, we first find the non-zero positions are unique to the pivot row. We then simply append these blank positions to the end of the current row, and update the column non-zero pattern calculation done earlier (a constant time operation). The number of non-zeros unique for the pivot row can be given by -

$$current\_row' = pivot\_row - (current\_row \cap pivot\_row) \tag{3.3}$$

The total number of non-zero positions allocated for any row is simply $current\_row' + current\_row$. Row non-zero pattern calculation is the most time consuming step in the symbolic analysis of matrix, requiring $O(n)$ time per row to find the elements unique to pivot row and an overall time of $O(n^2)$.

3.3.3 CSR Indices for Diagonal Elements

This is trivial final step we use in symbolic analysis to get the index from which subtractions should begin for a given current and pivot row during sparse LU factorization. Since we perform an in-place sparse LU decomposition, it is important to not subtract elements to the left of diagonal or the pivot element.

3.4 Parallel Sparse LU Factorization

In this subsection, we discuss the parallel LU factorization employed by our sparse solver. In this step, we first assign a work group for each row corresponding to a given pivot row. In case of Nvidia GPU, this is simply a thread block, while for the Xeon Phi coprocessor, we assign a thread for each row. As an example, consider a 5×5 matrix as shown, with its corresponding sparse matrix representation on the right (after adding zeros in the symbolic analysis step):

$$\begin{bmatrix} [4] & 18 & 0 & 0 & 0 \\ \{13\} & 10 & 0 & 8 & 20 \\ 0 & 1 & 7 & 17 & 0 \\ \{8\} & 0 & 11 & 3 & 0 \\ 0 & 0 & 0 & 3 & 19 \end{bmatrix} \quad \begin{array}{c} row[0] = \{4, 18\} \\ row[1] = \{13, 10, 8, 20\} \\ row[2] = \{1, 7, 17, 0\} \\ row[3] = \{8, 0, 11, 3, 0\} \\ row[4] = \{3, 19\} \end{array}$$

Consider $row[0]$ for the pivot row. For this pivot, the corresponding '*current rows*' which undergo Gaussian Elimination are $row[1]$ and $row[3]$. Since we also maintain a column-wise list of non-zeros, we can access each current row in constant time which is operated upon by a worker group or threads. Figure 3.3 shows the sparse Gaussian Elimination between $row[0]$ and $row[1]$ in action. For our Xeon Phi implementation, the relevant portion of the pivot row ($row[0]$) is first scattered into a vector of appropriate length (512-bit for the Xeon Phi), and the vector is divided by the pivot element. In the second step, all the current rows are simultaneously scattered into independent vectors by the corresponding threads. In the third step, the following operation is performed on all the *current row* vectors, as part of the

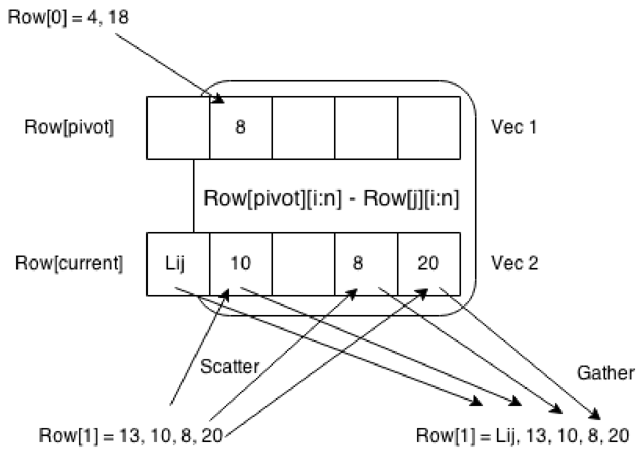

Figure 3.3: Sparse Gaussian Elimination between a *pivot row* and *current row*

standard Gaussian Elimination between rows:

$$Row_{current} = Row_{current} - Row_{pivot} \times Row_{current}[pivot] \tag{3.4}$$

The corresponding L_{ij} entry from the lower triangular matrix in LU decomposition is also calculated and replaced in place of the new zero that is created at the pivot position for the current row. In the third step, all the *current row* vectors are gathered in parallel and committed to the CSR data structure for the matrix.

The GPU implementation is marginally different. Here, each *current row* is assigned to a thread block and computations within a thread block are assigned to individual threads, instead of a vectorized Gaussian Elimination as in the case of Xeon Phi. Algorithm 4 formally summarizes the parallel sparse LU factorization described above.

Algorithm 4 Parallel Sparse LU Factorization

```
$current\_row$[0 : max_threads] = [0]*n; //initialize vectors
get $pivot\_index$ and $pivot\_element$
$Vec\_pivot\_row$ = scatter(CSR[pivot_row]);
$Vec\_pivot\_row$ /= $pivot\_element$;
for $i$ in range (0, n):
    parallel for j in range(0, $nz\_cols$[i]):
        $tid$ = get_thread_id();
        start_col_index = $nz\_cols$[i][j];
        $L\_numerator$ = CSR[$current\_row$][$start\_col\_index$];
        $Vec\_current\_row$[$tid$] = scatter(CSR[$current\_row$]);
        $Vec\_current\_row$[$tid$] -= $Vec\_pivot\_row$ * $L\_numerator$;
        CSR[$current\_row$][$start\_col\_index$+1:length_row]= gather($Vec\_current\_row$[$tid$]);
        CSR[$current\_row$][$start\_col\_index$] = $L\_numerator$ / $pivot\_element$; //Storing L
```

3.5 Block Forward Backward Substitution

As described by Equations (3.1) and (3.2), block forward backward substitution first serially solves a matrix along the diagonal, then substitutes the values obtained from its solution into equations involving SpMV operations. These two steps are repeated for all the block matrices produced as a result of permuting the matrix into the Block Triangular Form (BTF). With regards to parallelizing the SpMV operations on the coprocessor Xeon Phi, we assign one thread per row (master thread), which in turn spawns multiple threads per row. Each of the thread spawned performs a vectorized multiplication, and finally all the elements are reduced via addition to a single term per row by the master thread. The 512-bit vector unit is capable of performing 8 double-precision or 16 single-precision multiplications in a single SIMD instruction. For the GPU implementation, each CSR row is assigned to a thread block, and each block invokes $length(CSR[i])$ number of threads for the i^{th} row, to perform the multiplications in that row.

4. EXPERIMENTAL RESULTS AND DISCUSSION

4.1 Experimental Setup

We test the performance of our parallel sparse circuit solver on the following three computing platforms: Intel Xeon E5-1620 CPU, Intel Xeon Phi 3120A coprocessor, and NVIDIA GeForce GTX 680 GPU. The specifications of all these platforms are listed in Table 4.1.

Table 4.1: Specifications of devices used in experiments

Devices	Xeon E5-1620	Xeon Phi 3120A	GTX 680
# of cores	4	57	1536 (CUDA cores)
Active threads	8	228	16384
Clock Rate	3.8 GHz	1.1 GHz	1.08 GHz
Memory			
Total	8 GB	6 GB	4 GB
Bandwidth	51.2 GB/s	240 GB/s	192.2 GB/s
Speed	6 Gbps	6 Gbps	1333 MHz
Cache	10 MB	28.5 MB (total) 512 KB (L2/core)	512 KB (L2/all)
Vector unit width	64 bit	512 bit	–

For the CPU implementation, we used Centos 6.5 (64-bit) version with the Intel *icc/icpc* compiler and Advanced Vector Extensions (AVX) [14] enabled. For implementation on the Xeon Phi coprocessor, we used the same 64-bit Centos server with the *icc* compiler, but with Xeon Phi's own Vector unit (VPU) enabled. The VPU supports lane predicated execution, which helps in vectorizing short conditional branches. The VPU also supports gather and scatter instructions, which are simply non-unit stride vector memory accesses, directly in hardware. For the the GTX

680, we used PyOpenCL [15], a Python wrapper for OpenCL implementation on the GPU. The machine used for the GPU implementation featured a 64-bit Ubuntu 12.04 edition with CUDA 5.5 [16]. To test the performance of our sparse LU factorization implementations, we used a set of 15 different circuit matrices from the University of Florida Sparse Matrix Collection [17], which include circuits like cache memory and voltage regulators. We also evaluate certain random sparse matrices to analyze sparse matrix-vector multiplications.

4.2 Performance and Speedup

4.2.1 Performance of SpMV Operations

As described previously, sparse matrix vector multiplications are used in the right hand solving for block forward-backward substitution step of our sparse solver. A simple analysis showed that a row based format like the Sparse Compressed Row is suitable for parallelization as it is devoid of thread synchronizations. The results in Table 4.2 verify this analysis for performance comparison of three formats - Coordinate Format (COO), Sparse Compressed Row (CSR), and Sparse Compressed Column (CSC). The table shows GFlops numbers for our fastest parallel threaded and vectorized implementation across the three formats for the circuit matrix *IBM EDA* (N=116835, nnz=766396) from the University of Florida Sparse Matrix Collection [17]. Clearly, the CSR format yields more than an order of magnitude better performance than the COO and CSC formats for all the three architectures.

Next, we analyze the performance of SpMV operations using the CSR format as a function of the number of threads on different platforms. The aim here is to find the maximum achievable Flops as a function of number of threads for different architectures, so we choose a fairly dense random sparse matrix, with a sparsity of 0.25 and n=30,000 for our tests. Figure 4.1 shows GFlops versus the number

Table 4.2: Performance of parallel SpMV operations for COO, CSC and CSR sparse matrix formats

	COO (unsorted)	CSC	CSR
Xeon E5-1620	0.106 GFlops	0.088 GFlops	2.1 GFlops
Xeon Phi 3120A	0.089 GFlops	0.069 GFlops	5.6 GFlops
GTX 680	0.076 GFlops	0.056 GFlops	4.1 GFlops

of threads for Xeon, Xeon Phi and the GTX 680. For the Xeon and Xeon Phi, performance of the threaded and vectorized versions are shown alongside the purely threaded versions. As seen for the E5-1620, introducing vectorization for each thread yields no additional improvement over the purely threaded version. This is mostly due to small width of the vector unit (64 bits), and as a result, the GFlops peak at around 55 for both the versions. For Xeon Phi for the purely threaded version, the performance slowly saturates as the number of threads approach the maximum number of hardware threads supported by the coprocessor and peak at 84 GFlops for 228 threads. For the threaded and vectorized version however, we observe a peak performance of 237 GFlops, which is 2.8x times the performance of the purely threaded version. This can be attributed to two factors - a very wide 512-bit vector unit per thread, and hardware support for gather and scatter operations encountered in the SpMV operations. For the GTX 680, the performance variation is very smooth and peaks at 54.8 GFlops, which is about the same as the E5-1620. The GPU is compute starved as opposed to bandwidth starved since the performance increases no further when the number of CUDA threads are increased beyond what are present in the GPU.

In order to get an estimate for the performance of our SpMV kernels for very sparse matrices commonly found in circuit simulations, we also observe the variation of GFlops as a function of matrix sparsity for different architectures. Sparsity of a

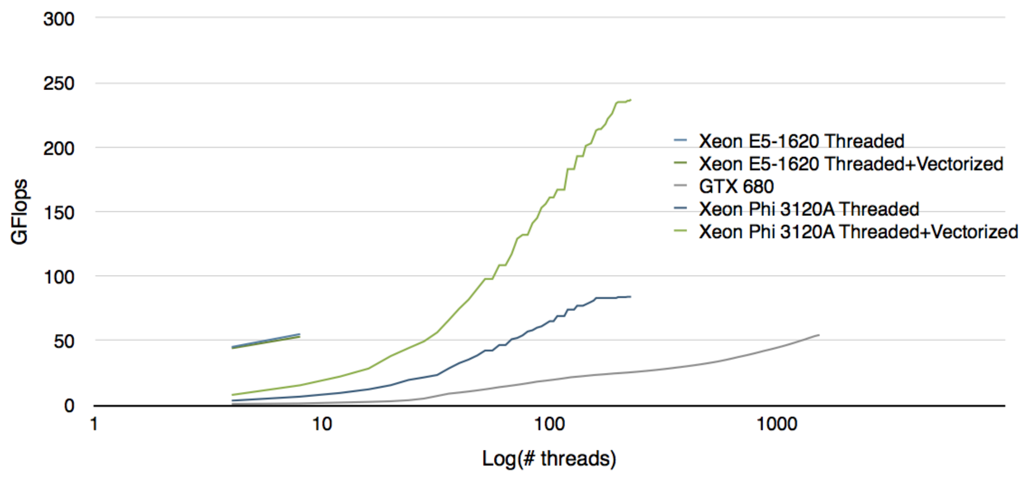

Figure 4.1: Performance of SpMV for CSR format as a function of threads

matrix is the number of non-zero elements divided by the total number of elements in the matrix. Figure 4.2 shows the performance (GFlops, logarithmic) of different architectures for a random sparse matrix with varying sparsity. A sparsity of 1 implies a dense matrix represented in CSR; a sparsity of 10^{-4} or less is typical of sparse circuit matrices. With decreasing matrix sparsity, the SpMV performance asymptotically approaches 4.5 GFlops, 4.7 GFlops and 6.7 GFlops for the Xeon, Xeon Phi and the GPU platforms respectively, for a random sparse matrix with sparsity 10^{-4}. Of course, the performance of SpMV kernels widely vary depending upon the matrix ordering employed, this analysis gives a useful estimate of the performance of SpMV kernels for unordered matrices for different platforms.

The reason for this decreasing performance with increasing matrix sparsity can be attributed to decrease in the Useful Cacheline Density (UCLD) [18] of the vector which is multiplied during the SpMV operations. Figure 4.3 shows a row of the sparse matrix fetched as a continuous 'dense' line and multiplied with a large vector, by accessing its elements from discontinuous locations. Since the entire vector cannot

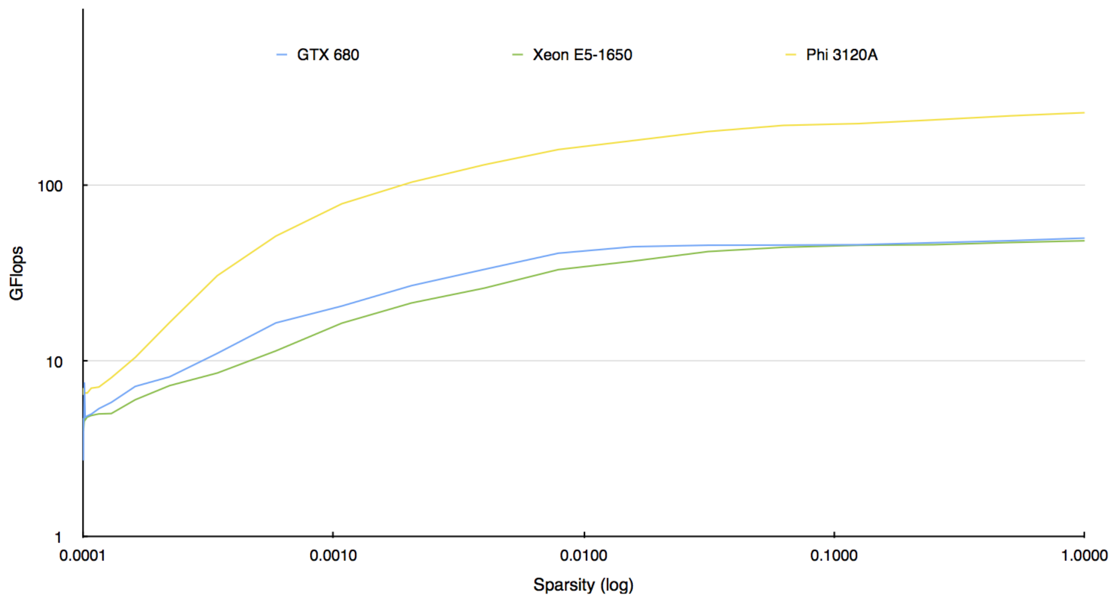

Figure 4.2: Performance of SpMV for CSR format as function of matrix sparsity

be accommodated in the cache line at the same time, entire cache line is discarded simply for accessing a single element as the distance between two accessed elements grows larger.

4.2.2 Performance of Parallel Sparse LU Factorization

Table 4.3 shows the performance of our parallel sparse LU factorization for 15 circuit matrices from the University of Florida Sparse Matrix Collection [17], evaluated for the Xeon CPU, Xeon Phi and the GTX 680 GPU. The table shows our implementation results for - CPU serial, CPU 4-core, GPU, Phi threaded (57-core) and Phi threaded+vectorized versions, along with comparison with KLU. The matrices are arranged according to the increasing number of average non-zeros per row.

Consider the GPU implementation; the listed time is only for numeric factorization, excluding preprocessing and forward-backward substitution for right hand solving. We observe a large variation in the speedups obtained for various matri-

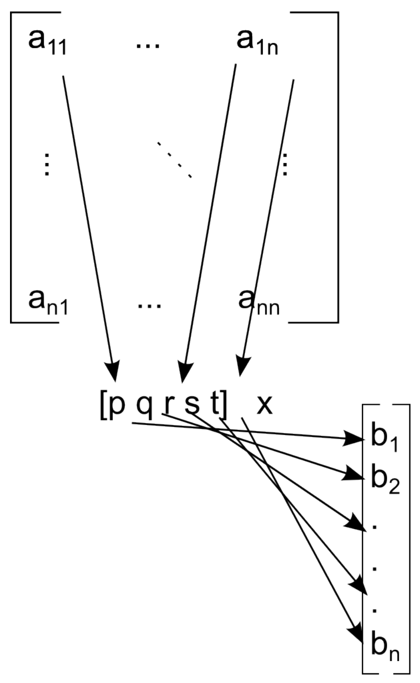

Figure 4.3: Decreasing useful cacheline density in an SpMV operation

ces. For matrices with less than 200M floating point operations overall, the average speedup is often less than 1. As explained in [19], this can be attributed to the inability to explore the high memory bandwidth of GPU when the problem scale is small. Overheads like data transfer and launching kernels account for most of the runtime. For certain circuits like *majorbasis*, *ASIC_320k* and *ASIC_320ks*, the GPU yields unusually high speedups. This is because many denormal floating point numbers, which are used to represent extremely small real numbers, occur for these matrices. CPUs perform poorly for denormal numbers [20] compared to GPUs, which can handle denormal numbers at the same speed as normal numbers. The average GPU speedup is calculated to be 2.75x, excluding the matrices with fewer floating point operations and ones with denormal numbers.

The threaded only Xeon Phi implementation is predictably related with the CPU implementation, with a consistent speedup of between 3-4x compared to the 4-core CPU implementation. The threaded+vectorized implementation in Phi reveals an increasing speedup over the threaded only version, as the average number of non-zeros/row of the matrix increase. This is illustrated in figure 4.4, where the speedup of threaded+vectorized implementation in Xeon-Phi over a purely threaded implementation is plotted against the increasing non-zeros/row of our matrices.

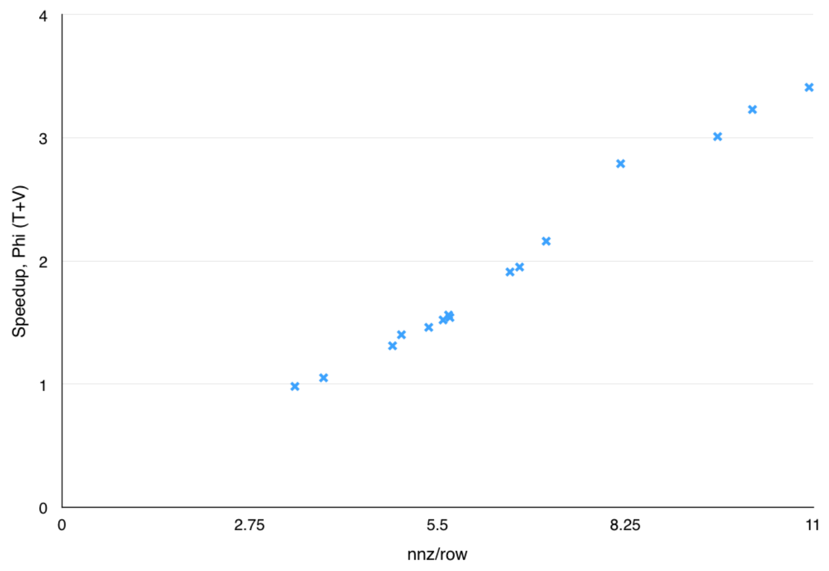

Figure 4.4: Speedup, Xeon Phi (threaded+vectorized, over threaded only) vs. non-zeros/row

Clearly, the large width of the vector unit in Xeon Phi is at play here. With the 512-bit wide VPU in the Phi, up to 8 double-precision operations can be performed simultaneously, yielding a maximum speedup of 8x theoretically, compared with the equivalent version with vectorization turned off. The average speedup with the threaded and vectorized implementation on Xeon Phi is calculated from the table to be 7.18x over the 4-core CPU version.

Table 4.3: Performance of parallel sparse LU factorization on CPU (Xeon E5-1620), GTX 680 (GPU) and Xeon Phi 3120A (threaded [T], threaded + vectorized [T+V])

Matrix	N ($\times 10^3$)	Non-zeros ($\times 10^3$)	Avg. nnz/row	# of fl. ops. ($\times 10^6$)	Time(s), CPU Serial	Time(s), CPU 4-core	Time(s), GPU
ASIC_680ks	682.7	2329.2	3.41	436.5	1.194	0.40679	0.120
circuit_4	80.2	307.6	3.83	2.5	0.014	0.00589	0.014
G2_circuit	150.1	726.7	4.84	4780.0	9.475	3.31097	0.966
hcircuit	103.2	513.1	4.97	1.0	0.015	0.00650	0.004
transient	178.9	961.8	5.37	107.8	0.261	0.09901	0.105
bcircuit	67.3	375.6	5.58	5.1	0.025	0.01169	0.007
ASIC_680k	682.9	3871.8	5.66	474.8	1.265	0.45606	0.462
ASIC_320ks	321.7	1827.8	5.68	651.6	24.384	6.60863	0.132
dc1	116.8	766.4	6.56	16.9	0.041	0.01858	0.094
ckt11752_dc_1	49.7	333.0	6.70	114.6	0.249	0.09905	0.044
hvdc2	189.9	1347.3	7.09	19.2	0.059	0.03840	0.012
ASIC_320k	321.8	2635.4	8.18	584.1	21.008	5.60974	0.310
ASIC_100k	99.3	954.2	9.60	529.6	0.923	0.37088	0.213
twotone	120.8	1222.4	10.11	5245.0	10.209	3.26702	0.769
majorbasis	160.0	1750.4	10.94	3933.0	21.390	6.97008	0.893

Matrix	Speedup, GPU vs. 4-core CPU	Time(s), Phi T	Time(s), Phi T+V	Speedup, Phi T+V vs. 4-core CPU	Speedup, Phi T+V vs. Phi T	Speedup, Phi vs. KLU
ASIC_680ks	3.38	0.1308	0.1327	3.06	0.98	14.96
circuit_4	0.42	0.0020	0.0019	3.10	1.05	13.65
G2_circuit	3.42	0.9197	0.6975	4.74	1.31	15.61
hcircuit	1.62	0.0021	0.0015	4.33	1.40	19.35
transient	0.94	0.0267	0.0182	5.44	1.46	15.12
bcircuit	1.67	0.0032	0.0021	5.56	1.52	7.54
ASIC_680k	0.98	0.1222	0.0783	5.82	1.56	8.69
ASIC_320ks	50.06	1.6929	1.0936	6.04	1.54	8.68
dc1	0.19	0.0057	0.0030	6.19	1.91	9.67
ckt11752_dc_1	2.25	0.0286	0.0146	6.78	1.95	10.87
hvdc2	3.20	0.0106	0.0049	7.83	2.16	25.87
ASIC_320k	18.09	1.1736	0.4205	13.34	2.79	26.45
ASIC_100k	1.74	0.1050	0.0350	10.59	3.01	12.70
twotone	4.23	0.8455	0.2617	12.48	3.23	17.09
majorbasis	7.80	1.9101	0.5593	12.46	3.41	15.76
Average	**2.75**			**7.18**	**1.95**	**14.80**

5. PREVIOUS WORK

Extensive work has been done previously for direct linear solvers and linear solvers in general. The survey in [21] gives a comprehensive list of algorithms and software available to solve sparse linear systems using direct methods for serial platforms, shared memory parallel machines, and distributed memory parallel machines. A subset of these papers are directed towards sparse circuit matrices, but mostly for serial implementations [5, 22, 23]. Because of the high data dependency during the LU factorization, forward-backward substitution and the irregular structure of circuit matrices, limited work has been done on heterogenous platforms like the GPU. Previous work on GPU LU factorization has mostly focused on dense matrices, which cannot be extended for implementation for sparse matrices. Consider the *IBM_EDA* (116k by 116k) circuit matrix as an example. The LU factorization of this matrix assuming dense data representation is difficult on a GPU owing to large memory requirements for even storing the matrix (107 GB).

CPU based approaches like the SuperLU [23], Gilbert Peierl's algorithm [10] and PARDISO [24] use *supernodes* to enhance computing capability with dense blocks. It is however, difficult to obtain supernodes in extremely sparse matrices like circuit matrices, making these supernode based algorithms less efficient for circuit matrices, even for parallel implementations like GPU based PARADISO [24]. UMFPACK [25], and MUMPS [26] are based on multifrontal algorithm [27]. In PARADISO [24], the left-right looking algorithm [28] is developed.

KLU [5] implements column based Gilbert Peierl's algorithm without using supernodes and is specially efficient for circuit matrices. KLU also employs the Approximate Minimum Degree (AMD) [2] ordering to permute the matrix for reduced fill-in as a result of sparse LU factorization, and works particularly well with sparse circuit matrices. KLU however, is sequential. There have been parallel shared memory implementations for multi-core CPUs [24, 29], however, the number of CPUs sharing the same memory is often limited. GPUs provide a possible solution with large number of SIMT cores and has been explored in [19]. Their implementation involves sorting the non-zeros to improve the data locality for more coalesced accesses to global memory. The speedup obtained in comparison with an 8-core Intel Xeon CPU is still limited to an average of 1.49x with this approach.

Our implementation is an extension of KLU with two important differences:

(i) Use of the Sparse Compressed Row (CSR) instead of the Sparse Compressed Column (CSC) format.

(ii) A multithreaded, vectorized implementation of the sparse LU factorization phase and parallel sparse matrix-vector multiplication operations in the forward-backward substitution phase.

To our knowledge, this is the first work on parallel sparse solver for circuit matrices on the many-core Intel Xeon Phi platform. The performance evaluation of sparse matrix vector multiplications on the Xeon Phi is introduced in [18], however, the analysis does not extend to sparse solvers used in linear systems of equations.

6. CONCLUSION

We present a parallel sparse linear circuit matrix solver, which extends KLU for parallelization on modern heterogenous platforms like the GPU and Intel Xeon Phi coprocessor. We have shown a row based sparse matrix representation format like the CSR to be more amenable to parallelization on our architectures. With our row based CSR representation instead of the typical column based sparse data structures used in serial sparse solvers, we are able to obtain reasonable speedups with both the GPU and coprocessor implementation. Our proposed LU factorization algorithm is particularly suited for vectorization on machines with wider vector units.

One limitation of our parallel sparse LU factorization implementation is the inability to accommodate large upper and lower triangular matrices (L and U) in GPU/co-processor memory for cases when the fill-in introduced by symbolic analysis is high. The fill-in, in turn depends on the ordering scheme employed during the preprocessing step. Hence, for very large matrices, the implementation is influenced by the ordering algorithm used in the preprocessing step, to ensure minimal fill-in during factorization. As GPUs and co-processors employ higher memories, the scalability to matrices with more non-zeros and fill-ins can be improved.

REFERENCES

[1] Atkinson, K. *An introduction to numerical analysis.* Iowa City: John Wiley & Sons, 1987.

[2] Amestoy, P., Davis, T. and Duff I.S. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, volume 17, no. 4, pages 886-905, December 1996.

[3] Davis, T., Gilbert, G., Larimore, S. and Ng, E. A column approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, volume 30, no. 3, pages 353-376, September 2004.

[4] Chan, W., and George, A. A linear time implementation of the reverse Cuthill-McKee algorithm. *BIT Numerical Mathematics*, Volume 20, Issue 1, pages 8-14, 1980.

[5] Davis, T. and Palamadai, E. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Transactions on Mathematical Software*, volume 37, no. 3, 2010.

[6] Duff, I.S. On algorithms for obtaining a maximum transversal. *ACM Transactions on Mathematical Software*, volume 7, no. 3, pages 315-330, 1981.

[7] Duff, I.S. Algorithm 575: Permutations for a zero-free diagonal. *ACM Transactions on Mathematical Software*, volume 7, no. 3, pages 387-390, 1981.

[8] Duff, I.S. and Reid, J.K. Algorithm 529: permutations to block triangular form. *ACM Transactions on Mathematical Software*, volume 4, no. 2, pages 189-192, 1978.

[9] Duff, I.S. and Reid, J.K. An implementation of Tarjan's algorithm for the block triangular form of a matrix. *ACM Transactions on Mathematical Software*, volume 4, no. 2, pages 137-147, 1978.

[10] Gilbert, J.R. and Peierls, T. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM Journal on Matrix Analysis and Applications*, volume 9, no. 5, pages 862-873, 1988.

[11] Golub, G.H., Loan, V. and Charles, F. *Matrix computations, 3rd edition*. Baltimore: Johns Hopkins, 1996.

[12] Hager, W.W. Condition estimates. *SIAM Journal of Scientific Statistical Computations*, volume 5, no. 2, pages 311-316, 1984.

[13] Higham, N.J. Fortran codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation. *ACM Transactions on Mathematical Software*, volume 14, no. 4, pages 381-396, 1988.

[14] Architecture instruction set extensions programming reference. https://software.intel.com/sites/default/files/m/9/2/3/41604. [Online; Accessed: January, 2014].

[15] Klckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P. and Fasih, A. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing, Elsevier*, volume 38, no. 3, pages 157-174, March 2012.

[16] Nvidia CUDA C programming guide v5.5. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. [Online; Accessed: July 2013].

[17] Davis, T. and Hu, Y. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, volume 38, no. 1, pages 1-25, December 2011.

[18] Saule, E., Kaya, K. and Catalyurek, U.V. Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi. In *Proceedings of the 10th International Conference on Parallel Processing and Applied Mathematics (PPAM)*, September 2013.

[19] Ren, L., Chen, X., Wang, Y., Zhang, C. and Yang, H. Sparse LU factorization for parallel circuit simulation on GPU. In *Proceedings of the 49th Annual ACM/EDAC/IEEE Design Automation Conference (DAC '12)*, pages 1125-1130, June 2012.

[20] Soras, L. Denormal numbers in floating point signal processing applications. http://musicdsp.org/files/denormal.pdf. [Online; Accessed: April, 2014].

[21] Li, X. Direct solvers for sparse matrices. http://crd-legacy.lbl.gov/~xiaoye/SuperLU/SparseDirectSurvey.pdf. [Online; Accessed: July, 2013].

[22] Chen, Y., Davis, T., Hager, W.W. and Rajamanickam, S. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software*, volume 35, no. 3, pages 22.1-22.14, October 2008.

[23] Demmel, J.W., Eisenstat, S.C., Gilbert, J., Li, X.S., and Liu, J.W.H. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, volume 20, no. 3, pages 720-755, 1999.

[24] Schenk, O. and Gartner, K. Solving unsymmetric sparse systems of linear equations with PARDISO. In *Proceedings of the International Conference on Computational Science (ICCS 2002)*, volume 2330, pages 355-363, 2002.

[25] Davis, T. Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, volume 30, pages 196-199, June 2004.

[26] Amestoy, P.R., Guermouche, A., L'Excellent, J.Y. and Pralet, S. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing, Elsevier*, volume 32, no. 2, pages 136-156, 2006.

[27] Liu, J.W.H. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, volume 34, no. 1, pages 82-109, 1992.

[28] Schenk, O., Gartner, K. and Fichtner, W. Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors. *BIT Numerical Mathematics*, volume 40, pages 158-176, 2000.

[29] Ashcraft, C. and Grimes, R. SPOOLES: An object-oriented sparse matrix library. In *Proceedings of the Ninth SIAM Conference on Parallel Processing*, San Antonio, 1999.