New Sparse Matrix Ordering Techniques for Computer Simulation Of Electronics Circuits

DAVID CERNY
Czech Technical University in Prague
Department of Radio Engineering
Technická 2, 166 27 Praha 6
Czech Republic
cernyd1@fel.cvut.cz

JOSEF DOBES
Czech Technical University in Prague
Department of Radio Engineering
Technická 2, 166 27 Praha 6
Czech Republic
dobes@fel.cvut.cz

Abstract: Engineers and technicians rely on many computer tools, which help them in the development of new devices. Especially in the electronic circuit design, there is required advantageous and complex equipment. This paper makes a brief introduction to the basic problematic of electronic circuit’s simulation. It proposes the key principles for developing modern simulators and possible further steps in computer simulation and circuit design. Main part of the article presents a novel sparse matrix ordering techniques specially developed for solving LU factorization. LU factorization is critical part of any electronic circuit simulation. This paper also presents complete description of backward and forward conversion of the new sparse matrix ordering techniques. The comparison of performance of standard matrix storages and novel sparse matrix ordering techniques is summarized at the end of this article.

Key–Words: Computer simulation, SPICE, Sparse Matrix Ordering Techniques, LU factorization

1 Introduction

Simulation program SPICE (Simulation Program with Integrated Circuit Emphasis) [1, 2] has dominated in the field of electronics simulation for several decades. An enormous impact of significance of SPICE simulator can be easily seen on division of simulation programs for electronic circuit into two groups: Spice-like simulators which use for simulation same principles as SPICE and others based on a different programming paradigm. The first version of the program was introduced to public in 70’s in the University of California in Berkeley. Although countless number of other programs for computer-aided design (CAD) and analyzing have been developed, SPICE is the most-used program for circuit simulation. SPICE computation core can be found in many professional programs as PSPICE (Cadence), Spectre (Cadence), HSPICE (Synopsys), ELDO (Mentor Graphics), SmartSPICE (Simucad), NgSpice and many others.

2 Future of Electronics Simulation

Nearly 40 years have passed since SPICE simulator was developed, but still we are facing very difficult decision. Will it be the most-used computer simulator for the next 40 years or the time for a change has finally come? To better understanding the problematic we must a little investigate SPICE core algorithms. The entire program and especially its simulation core algorithms were written in Fortran, program core algorithms were lately converted to C language [3]. Despite its age Fortran (and especially its modern version) allows extremely precise memory allocation and produces very efficient machine code. It is a very difficult task (even impossible) to make a program with competitive computation core in recent programming languages as C++, Java, C#.

In the paper [4], a new program architecture is proposed for simulators of electronics circuits, which will possibly overcome SPICE. The idea is based on a use of functional programming language LISP [5]. Selection of functional language is not a coincidence. Functional languages are getting more on their significance nowadays. As an example we can mention that new version of C++11 (approved by ISO in August 2011) comes with the ability of creation of unnamed function so-called lambda function principle. This is absolutely fundamental principle to functional languages.

The efficiency and modularity of written code have started to be taken very seriously recently. A huge drawback of SPICE simulator (and practically every simulator based on same architecture) is its models. It is very difficult and requires high level pro-
gramming skills of the users to create own models. It is possible to modify various parameters of existing models, but it is time-consuming and difficult to create own one from scratch. Electronics engineers are forced to deal with memory management, complex algorithms and many redundant optimization parameters if they need to do so. Moreover, this is possible only in a case when the users can recompile simulator source codes. Otherwise it is very difficult to change anything or check, whether computation algorithms are correct.

3 Power of Functionals

Turning simulation program to functional style gives an opportunity to create environment, which could change itself during its run [6]. Functions could be created and mapped to variables at any time of the program run. This gives us very powerful tool to create special input syntax, which turns, for example, mathematical equations to functions. It is not parsing. Specific property of functional languages is a possibility of language syntax redefinition to special one [7]. This allows, for example, in electronics simulation to specify arbitrary complex device model just by definition by its mathematical circuit equations. To clear this statement, the implementation of a diode model in the functional language Common LISP (CL) will be illustrated.

We are going to rewrite well known Shockley’s equation of a diode to LISP code:

\[ I_D = I_S \left( e^{V_D/(nV_T)} - 1 \right), \]  

where \( I_D \) is diode current, \( I_S \) is saturation current, \( V_D = V_p - V_n \) is voltage across the diode, \( V_T \) is thermal voltage, and \( n \) is emission coefficient. Definition of this mathematical equation in functional language Common LISP will look as

```lisp
(defun diode-current (vp vn)
  ;; (lambda ()
  (* 1e-12
    (- (exp (/ (- (eval vp) (eval vn))
             26e-3))
       1)))
```

where \( I_S = 10^{-12} \) A, \( V_T = 26x10^{-3} \) V and \( n = 1 \), e.g.

This function (DIODE-CURRENT) with parameters (VP VN) returns another function (it is actually functional), defined by LAMBDA operator, which computes current \( I_D \) through the diode. It could be mentioned that it is only trivial redefinition of mathematical equation in LISP code. It is only introduction for real magic which comes with mapping function. Following (very self explanatory) LISP code is complete mapping function for previously defined Zener diode.

```lisp
(defmethod map-device ((d class:diode) (m class:matrix-system))
  (let ((v+ (make-var-node 'v (node+ d)))
        (v- (make-var-node 'v (node- d)))
        (i (make-var-name 'i (name d))))

    (set-value m v+ i '#+ 1)
    (set-value m v- i '#+ -1)
    (set-value m i i '#+ -1)

    (set-equations-value m i
diode-current d v+ v-))

    (set-value m v+ (diode-current-dva d v+ v-))
    (set-value m v- (diode-current-dvc d v+ v-)))
```

One of the main advantages of functionals is that they can be handled in a same way as values. They can be dynamically created with different parameters, stored in variable or array. This allows to them to be used as circuit device templates and so significantly enhance device redefinition speed. There is no more need to incorporate device definition into simulation algorithms. Simulation program should be able to automatically assign defined device functionals to correct positions in simulation chain and invoke them in the right order and time when they are needed.

4 Performance of Computer Simulations

One important fact must be pointed out about functional languages. And actually about any other programming language higher than C. No matter, how many optimization algorithms given compiler will use, resulting binary will be in any case slower than C or Fortran compilation. The result can differ only in several instructions, but this will get on significance when it is performed thousand times over. To compete in performance with other simulators, all its computation demanding tasks must delegated to C language, Fortran or even lower language.

In CL, it is possible though so-called Foreign Array Interface mechanism. It allows to CL to use, create and call functions and variables from other programming languages, for example C. All demanding tasks
can be passed and evaluated by code written in C language. It proved to be good approach to use GNU Scientific Library (GSL) [8]. It offers many mathematical functions and is very fast. Disadvantage of this solution is that GSL does not support sparse matrix storage. This fact could be very problematic in electronic circuit simulation because high sparsity is characteristic property of circuit equations.

5 Sparse Matrix Storage

In the classical electronic network application, we can be sure that system equations of the simulated circuit will produce a highly sparse matrix. The number of operations necessary to solve a system of equations by Gaussian elimination or by LU factorization is approximately \( \frac{n^3}{3} \), where \( n \) is number of rows. This is valid if all operations are actually performed, even those where we multiply or add zero. The computation time, especially in the circuit simulation, can be reduced by not performing these multiplications and additions (subtractions). In fact, the zeros do not need to be stored.

Sparse matrix algorithms neither store nor perform operations on zero. Programming of these methods is usually complicated due to the difficult process of handling entries. Basic sparse matrix techniques decompose a two-dimensional matrix to several vectors. The various techniques have been published in literature, for instance, Yale matrix, Skyline storage, Profile storage matrix, Diagonal storage matrix, Compressed sparse column, Compressed sparse row etc. [9, 10].

Known techniques for the sparse matrix manipulation showed to be impractical, when they are applied on the electronic circuit simulation (ECS). To solve this problem two new sparse matrix ordering techniques optimized for ECS has been developed.

6 New Sparse Matrix Storage

The first method is based on Profile-In Skyline Storage Mode defining index vectors in different style. In this article, the method will be referred as Modified Profile-In Skyline Storage (MPSLS). The second technique is new and is based on an order of LU factorization procedures. In the article, this method will be referred as Full Profile Skyline Storage (FPSS).

7 Modified Profile-In Skyline Storage

This method decomposes a matrix into several vectors. Three of them hold matrix values, and two index vectors. Vectors \( u \) and \( l \) are both value vectors, which are accompanied by two indexes vector \( i_u \), \( i_l \). Index vectors \( i_u \) and \( i_l \) combine the best of the row and column sparse matrix techniques. The index vector in \( i_u \) uses up down, left to right organization scheme, vector \( i_l \) works with a left to right, up to down scheme. Vector \( d \) holds diagonal matrix values and therefore, non-zero values. After successful pivoting there should be no zero values in vector \( d \). Therefore, it will have full length in any case, and does not need to be accompanied by any index vector. Visual example of the transformation of the matrix \( M \) to MPSLS sparse matrix ordering technique follows:

\[
M = \begin{bmatrix}
d & 0 & 1 & 3 \\
0 & d & 2 & 4 \\
1 & 2 & d & 5 \\
3 & 4 & 5 & d \\
\end{bmatrix}, \tag{2}
\]

where vector decomposition of matrix (2) will result in set of value vectors \( u \), \( l \) and \( d \):

\[
u = [0 \ 1 \ 2 \ 3 \ 4 \ 5],
\]

\[
l = [0 \ 1 \ 2 \ 3 \ 4 \ 5],
\]

\[
d = [d \ d \ d \ d],
\]

and set of index vectors \( i_u \) and \( i_l \):

\[
i_u = [0 \ 1 \ 2 \ 3 \ 4 \ 5],
\]

\[
i_l = [0 \ 1 \ 2 \ 3 \ 4 \ 5]. \tag{4}
\]

To be able actively and fast convert column and row indexing to MPLS, and back we developed conversion algorithms.

7.1 Forward conversion of MPLS

The forward conversion algorithm first check, whether value is on a diagonal of the matrix. If so, the conversion is easy and value is inserted into the value vector \( d \). Otherwise it must be converted by the following algorithms:

\[
d = x \quad \text{for } x = y,
\]

\[
i_u = y(y + 1)/2 + x \quad \text{for } x < y, \tag{5}
\]

\[
i_l = x(x + 1)/2 + y \quad \text{for } x > y,
\]

where \( x \) and \( y \) are zero based index pairs of original matrix.

Values in the value vectors do make pairs with index vector values and together can be in any position. It is also very convenient in the cases when new values are added to the matrix. This principle allows to resize matrix without affecting indexation values. It is important to point out that presented algorithms (FPSS and MPLS) have zero-based indexation (indexation vectors start with 0).
7.2 Backward conversion of MPLS

The backward conversion of MPLS requires a little more operations than the forward one. To obtain original matrix indexation values \( x \) and \( y \) the values from vectors \( i_l \) and \( i_u \) must be converted to intermediate indexes \( m \) and \( n \).

\[
\begin{align*}
  n &= \text{isqrt}(z), \\
  m &= z - \left((n + 1) \times n/2\right),
\end{align*}
\]

where \( z \in \{i_u;i_l\} \). It should be noted a usage of the Integer Square Root function (ISQRT). In number theory, the ISQRT of a positive integer \( n \) is the positive integer \( m \), which is the greatest integer less than or equal to the square root. This function is a standard part of every mathematical library of common programming languages:

The need of this intermediate step arises from the fact that this conversion is a minimization of quadratic form on integer-valued domain. This is usually very problematic and ISQRT together with special constraints of resulting domain helps effectively solve this situation. After obtaining \( n \) and \( m \) variables, we can decode correct \( x \) and \( y \) values simply from index vector \( i_u \) as

\[
[x, y] = [n - 1, n + m],
\]

and from index vector \( i_l \) as

\[
[x, y] = [n + m, n - 1].
\]

The vector \( d \) is sorted in same order as values on diagonal of original matrix thus position of values equal to their indexes. Conversion from \( d \) is simple assignment:

\[
[x, y] = [i_d, i_d].
\]

8 Full Profile Skyline Storage

The idea of FPSS arises from a fact that only operation performed on a sparse matrix is LU factorization. Therefore, values in vector decomposed from the original sparse matrix are ordered in a same way as LU factorization is performed. It also reduces a number of index vectors to only one. It is the most important advantage of this storage and must be pointed out that any value in matrix can be obtained by only one key index. Another benefit of this storage is possibility of matrix division to parts with equal memory sizes. This parts can be stored in different places in memory. It is very convenient when simulation needs to deal with more than \( 1 \times 10^6 \) entries.

FPSS index vector is composed directly from position of values in sparse matrix. For illustration we start with small example. Let the values in matrix \( M \) represent in this situation special indexing order of FPSS:

\[
M = \begin{bmatrix}
  0 & 2 & 6 & 12 \\
  1 & 3 & 7 & 13 \\
  4 & 5 & 8 & 14 \\
  9 & 10 & 11 & 15
\end{bmatrix}.
\]

Then its decomposition to indexation and value vectors will be trivial:

\[
v = \begin{bmatrix}
  0 & 1 & 2 & 3 & 4 & 5 & 6 \ldots & 15
\end{bmatrix},
\]

\[
i = \begin{bmatrix}
  0 & 1 & 2 & 3 & 4 & 5 & 6 \ldots & 15
\end{bmatrix}.
\]

FPSS is based on row, column indexation. Values in matrix are indexed from left top corner along diagonal in row column order to right bottom. This is special order allowed to develop algorithm for very efficient conversion of values from standard \([x, y]\) (two dimensional) way to one dimensional \([z]\). Definition of developed FPSS conversion algorithms follows.

8.1 Forward conversion of FPSS

The forward conversion of FPSS is trivial and therefore can be performed very fast. Only difficult part is evaluation of “\(i\)” condition. However, it can be also easily solved by the use of templates, functions or even better functionals:

\[
f(x, y) = \begin{cases} 
  x + y^2 & \text{for } x > y, \\
  (x + 1)^2 - 1 & \text{for } x < y, \\
  y(y + 1) + x & \text{for } x = y.
\end{cases}
\]

8.2 Backward conversion of FPSS

The backward conversion as well as the one in MPLS still requires usage of ISQRT function. Nevertheless, the algorithm is still extremely simple and can be implemented in the very efficient way:

\[
\begin{align*}
  n &= \text{isqrt}(z), \\
  d &= n^2, \\
  m &= z - d,
\end{align*}
\]

and

\[
[x, y] = \begin{cases} 
  [n, n] & \text{for } n/2 = m, \\
  [n, m] & \text{for } n > m, \\
  [m - n, n] & \text{for } n < m.
\end{cases}
\]

As it has been told FPSS allows to store values in different but equally big memory places. This could be performed for example by very simple algorithm. Lets say that we have matrix with 16 values. Then we decide to slice every 4 values to different memory places (files, databases . . . ). The FPSS algorithm first computes index value \( z \) and then by simple modulo division it will obtain right memory place index.
Table 1: Capability of Different Storage Systems

<table>
<thead>
<tr>
<th>Number of Matrix Entries</th>
<th>Capability of storages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Array</td>
</tr>
<tr>
<td>100 000</td>
<td>yes</td>
</tr>
<tr>
<td>1 000 000</td>
<td>yes</td>
</tr>
<tr>
<td>10 000 000</td>
<td>no</td>
</tr>
<tr>
<td>100 000 000</td>
<td>no</td>
</tr>
</tbody>
</table>

* Storage is capable to hold this number of entries.
no Storage is not capable to hold this number of entries.

* Total time in sec. to iterate through all values.

Table 2: Performance of Sparse Matrix LUF

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Matlab</th>
<th>GLLU*</th>
<th>Sparse CLU*</th>
<th>LISP*</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 × 50</td>
<td>-</td>
<td>0.33</td>
<td>0.004</td>
<td>0.43</td>
</tr>
<tr>
<td>100 × 100</td>
<td>0.01</td>
<td>0.43</td>
<td>0.027</td>
<td>12.73</td>
</tr>
<tr>
<td>200 × 200</td>
<td>0.02</td>
<td>0.96</td>
<td>0.21</td>
<td>-</td>
</tr>
<tr>
<td>500 × 500</td>
<td>0.20</td>
<td>3.20</td>
<td>2.42</td>
<td>-</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1.7</td>
<td>20.2</td>
<td>19.25</td>
<td>-</td>
</tr>
</tbody>
</table>

* No additional optimization algorithms were performed.

9 Results

A capability of different storage systems holds huge amount of data is shown in Table 1. First column represents number of matrix entries, which goes from $1 \times 10^5$ to $1 \times 10^8$ entries. Three following columns show three different matrix storage systems. First one, is Array which is simple memory allocated array. It represents biggest allocable continual space on disk. MPLS and FPSS are our implemented matrix storage systems.

Table 2 compare performance of different LU solvers. First column in the table represents size of tested matrix. The second one holds values from Matlab solver. The third one represents sparse matrix storage accompanied with GNU GSL library. The fourth one is LU solver with FPSS and the last one shows results from the LISP sparse matrix implementation. Last column was included only to illustrate difference between LU implementation in C (Fortran) code and LISP.

All tests were run on a computer with Intel Core 2 Duo, with the CPU frequency 2.26 GHz and 3.9 GB RAM. Operating system was Ubuntu with Linux Kernel 2.6.38-10-generic-pae.

10 Conclusion

In the article, we presented modern aspects of circuit simulation and possible further development in this field. From the results, we can conclude that MPLS is comparable with normal array, but in contrast to a normal array it can handle sparse matrix indexation. FPSS reduces a number of indexation vectors and is capable of holding a huge amount of data. Implementation of proposed sparse matrix storages to LU solver in terms of performance proved to be comparable with GNU GSL LU solver. Commercial Matlab LU solver was used only as a reference value. It can not be clearly compared with our implementation, because it includes many additional optimization algorithms.

Acknowledgements: This work is supported by the Grant Agency of the Czech Republic under grant No. P102/10/1665, and by the Grant Agency of the Czech Technical University in Prague, grant No. SGS10/286/OHK3/3T/13.

References: