2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

Parallel Circuit Simulation on Multi/Many-core Systems

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

***Abstract*—SPICE is widely used for transistor-level circuit simulation. However, with the growing complexity of the VLSI at nano-scale, the traditional SPICE simulator has become inefficient to provide accurate verifications. This thesis tries to accelerate transistor-level simulation on multi/many-core systems, and we will solve 3 problems: 1) develop a parallel sparse LU factorization algorithm for circuit simulation; 2) implement the matrix solver on GPU to further accelerate the solver; 3) develop a circuit partitioning based parallel simulation approach on distributed machines to obtain better scalability. The experimental results show that the proposed parallel LU factorization algorithm effectively accelerates the matrix solver for circuit simulation on both CPU and GPU.**

***Keywords*-Parallel Circuit Simulation, Parallel LU Factorization, GPU Acceleration**

I. Introduction

SPICE (Simulation Program with Integrated Circuit Emphasis) [1] developed by UC-Berkeley is a circuit simulator widely used in IC design. With the growing complexity of the VLSI, accurate post-layout simulation can often take days or even weeks of runtime on modern processors. Fig. 1 shows a typical flow of SPICE transient simulation. SPICE runtime is dominated by 2 steps: model evaluation and sparse matrix solver. Many parallel approaches have been proposed to accelerate SPICE simulation (see Section IV). The parallelism of model evaluation is straightforward, but the matrix solver is difficult to be parallelized because of the high data-dependency during LU factorization and the irregular structure of the circuit matrices. So the matrix solver has become the bottleneck in the SPICE flow.

With these motivations, our research goal is to accelerate SPICE simulation on multi/many-core systems. We will contribute to solve 3 problems.

The first is to parallelize the sparse matrix solver on shared-memory systems, which is the most severe bottleneck in the SPICE flow. This work has been published by TCAS-II [2] and ASPDAC-2012 [3]. The parallel solver can be downloaded from ***http://nicslu.weebly.com***. The features of the solver are: 1) column-level parallelism, which is suitable for circuit simulation [4]; 2) automatically decide whether a matrix is suitable for sequential or parallel algorithm; 3) two parallel modes are dynamically scheduled to fit the different

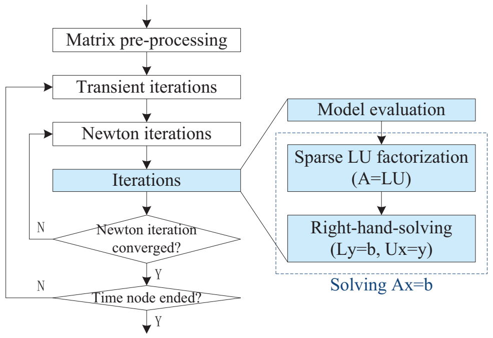


Figure 1. A typical SPICE simulation flow.

data-dependency and reduce scheduling overhead; 4) it's based on shared-memory machines and can be conveniently used on multi-core computers.

The second is to implement the parallel solver on GPU (Graphic Processing Units) to further accelerate it. The number of CPU cores sharing the same memory is often limited. The state of the art GPU provides a possible solution to this problem: a GPU naturally has numerous cores sharing the same device memory. This is the first work on GPU-based sparse LU factorization intended for circuit simulation. This work is accepted by DAC-2012 [5].

The last is to develop a circuit partitioning based simulation approach on distributed-memory machines, since the traditional parallel simulation approaches and the parallel matrix solver can easily reach bottlenecks on shared-memory computers when the number of cores increases, which means the scalability of the traditional parallel approaches is poor. The distributed methods will greatly improve the parallel scalability.

The first two problems have been already solved, and currently we are developing solutions for the last one.

II. Parallel LU Factorization Algorithm

A typical LU factorization has three steps: 1) pre-processing (performs column/row permutations to increase numeric stability and reduce fill-ins), 2) numeric factorization (factorizes matrix A into the product of L and U), and 3) right-hand-solving (solves $L\vec{y} = \vec{b}$ and $U\vec{x} = \vec{y}$). In circuit simulation, the pre-processing step is performed only once, the last two steps are repeated for many times during the iterations.

Fig. 2 shows the flow of our proposed parallel solver. After the pre-processing step, a symbolic factorization [6] is performed to predict the structure of L and U, which is

* Xiaoming Chen is a 1st-year PhD student, working with advisor Dr. Huazhong Yang and co-advisor Dr. Yu Wang.

This work was supported by National Science and Technology Major Project (2011ZX01035-001-001-002, 2010ZX01030-001) and National Natural Science Foundation of China (No.60870001, 61028006).

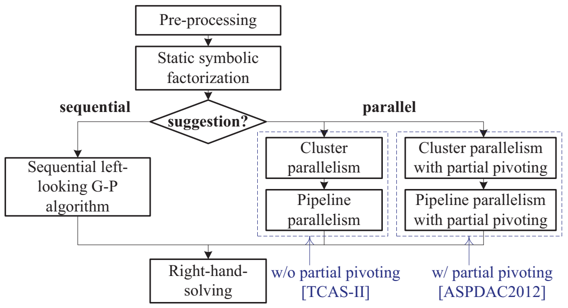

Pre-processing
Static symbolic factorization
sequential
suggestion?
parallel
Sequential left-looking G-P algorithm
Cluster parallelism
Pipeline parallelism
Cluster parallelism with partial pivoting
Pipeline parallelism with partial pivoting
Right-hand-solving
w/o partial pivoting [TCAS-II]
w/ partial pivoting [ASPDAC2012]

Figure 2. The flow of our parallel LU factorization algorithm.

used to give a suggestion that whether the matrix should use parallel or sequential algorithm. In the following, we will describe the parallel strategies about 3 points: task description, task partitioning and task scheduling.

Task description: The basic LU factorization algorithm is the left-looking Gilbert-Peierls (G-P) algorithm [4], [7], as shown in Fig. 3. It factorizes matrix A by sequentially processing each column (k) in 3 primary steps: 1) symbolic factorization, 2) numeric factorization, 3) partial pivoting. Numeric factorization (Fig. 3(b)) is the most time-consuming step. It indicates that the numeric factorization of column k refers to the data in some previous (left side) columns $\{j|U(j,k) \neq 0, j < k\}$. In other words, **column k depends on column j, iff $U(j,k) \neq 0 (j < k)$, which means the column-level dependency is determined by the structure of U**. The parallel strategy is to first determine the column-level dependency in the pre-processing step (performed once), and then the parallel factorization is scheduled by the column-level dependency (performed many times in circuit simulation).

Task partitioning: We have realized 2 parallel algorithms: with and without partial pivoting (see Fig. 2). The difference is that when adopting partial pivoting, the structure of L and U depends on pivot choices (partial pivoting can interchange the rows), so the exact dependency cannot be determined in the pre-processing step.

When adopting partial pivoting, we use *ETree (Elimination Tree)* [8] to represent the dependency (Fig. 4). It's not an exact representation, but contains all potential dependency regardless of the actual pivoting choices, so it overestimates the exact dependency. Node k in the *ETree* corresponds to column k in the matrix, so "node" and "column" are equivalent. Then we calculate *level* of each node. *level* is an intuitive concept that indicates the longest distance from each node to leaf nodes of the *ETree*. Finally, all the n nodes are categorized into different levels to obtain the *EScheduler (Elimination Scheduler)*.

Task scheduling: Take Fig. 4(d) as an example to illustrate the task scheduling strategies. The nodes in the same level are independent, so these nodes can be calculated in parallel effectively, with little scheduling time. It's *cluster mode* (level 0 and 1). However, when the nodes in one level become fewer (level $2 \sim 4$), *cluster mode* is ineffective.

```
Algorithm: Left-looking G-P algorithm
for k = 1:N do
    Symbolic factorization: determine the structure of
    column k, and which columns will update column k;
    Numeric factorization on column k;
    Partial pivoting on column k;
end for
```
(a)

```
Algorithm: Numeric factorization on column k
x = A(:, k);
for j<k where U(j, k)!=0 do
    x(j+1:n) = x(j+1:n) - x(j)*L(j+1:n, j);
end for
```
(b)

Figure 3. The left-looking G-P algorithm [4], [7].

Table I
THE RESULTS OF THE PARALLEL LU FACTORIZATION.

	w/ partial pivoting		w/o partial pivoting
	[1]group 1	[2]group 2	
#benckmarks	24	11	26
[3]speedup (4-core)	5.60	0.67	2.97
speedup (8-core)	8.38	0.46	4.55
[4]relative speedup (4-core)	2.66	0.60	2.88
relative speedup (8-core)	4.34	0.41	4.78

[1] **group 1**: the matrices which are suitable for parallel algorithm
[2] **group 2**: the matrices which are suitable for sequential algorithm
[3] **speedup**: the speedups over KLU
[4] **relative speedup**: the speedups over our 1-core solver

In this case, *pipeline mode* which **exploits parallelism between dependent levels** will be used. The 2 modes work coordinately and achieve a high level of concurrency. Because of space limitation, we omit the details here, please refer to [3].

If partial pivoting is not adopted, the exact dependency can be determined by the structure of U, which is obtained in the pre-processing step and forms an *EGraph (Elimination Graph)*. So *ETree* is not used, and the *EScheduler* is directly built from *level* categorization of the *EGraph*. The factorization is also scheduled by *cluster mode* and *pipeline mode*, but the detailed algorithm is different since here symbolic factorization is not needed during numeric factorization. The details can be found in [2].

The experiments are implemented by C on a server with 2 Xeon5670 CPUs (12 cores in total). The test benchmarks are from University of Florida Sparse Matrix Collection [9] (the largest matrix is 5.5M by 5.5M). We also test KLU [4] as a baseline for comparison. The results are summarized in Table I. All the speedup numbers listed are **geometric-average**, some labels are explained in footnotes. For the algorithm with partial pivoting, we test 35 matrices, and they can be separated into 2 groups. For the group which is suitable for parallel algorithm, our proposed solver can achieve effective speedups over KLU; For the algorithm without partial pivoting, we test 26 matrices, and our solver can achieve speedups for all the 26 matrices. The detailed results can be found in [2], [3].

III. GPU ACCELERATION

Although the CPU-version parallel solver can achieve effective speedups over KLU, the number of CPU cores is often limited, because of the bandwidth limitation of

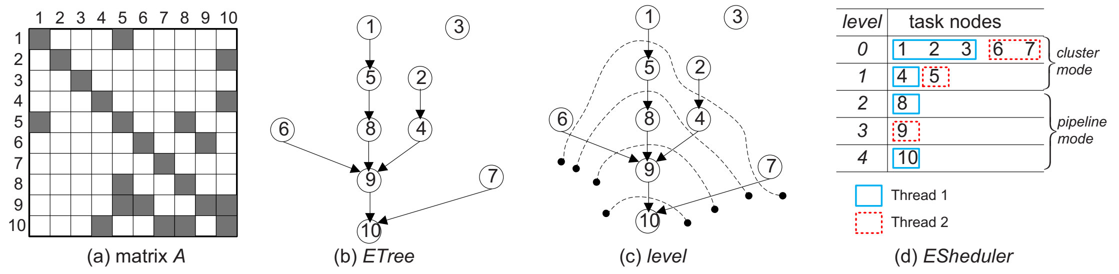


Figure 4. An example to illustrate *ETree*, *level*, and *EScheduler*.

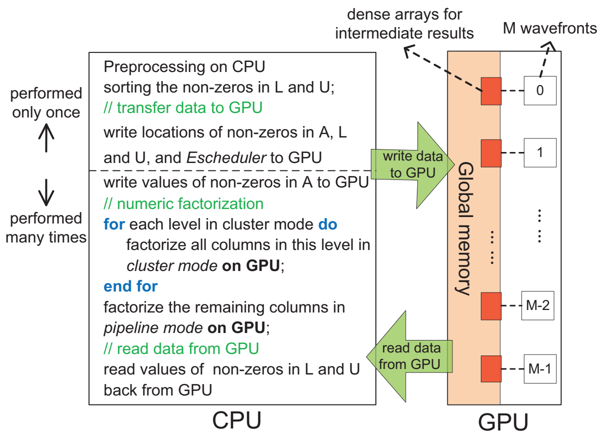

Preprocessing on CPU
sorting the non-zeros in L and U;
// transfer data to GPU
write locations of non-zeros in A, L and U, and *Escheduler* to GPU
write values of non-zeros in A to GPU
// numeric factorization
for each level in cluster mode do
factorize all columns in this level in *cluster mode* on GPU;
end for
factorize the remaining columns in *pipeline mode* on GPU;
// read data from GPU
read values of non-zeros in L and U back from GPU
performed only once
performed many times
write data to GPU
read data from GPU
dense arrays for intermediate results
M wavefronts
Global memory
0
1
M-2
M-1
CPU
GPU

Figure 5. The workflow of sparse LU factorization on GPU.

the memory bus. The state of the art GPU naturally has numerous cores sharing the same device memory, which provides a potential solution to improve the parallel scalability and obtain more speedups. In this work, we for the first time propose a GPU-based sparse LU solver for circuit simulation.

Fig. 5 is the workflow of GPU-based sparse LU factorization. The pre-processing is performed only once on CPU. For numeric factorization, based on GPU architecture, we expose more parallelism in G-P algorithm by partitioning one single vector multiple-and-add operation (see Fig. 3(b)) and mapping the partitioned tasks onto several GPU threads, since GPU schedules threads in a SIMD (single-instruction-multiple-data) manner. In addition, we propose sorting the nonzeros of L and U to improve the data locality for more coalesced access to global memory on GPU, which increases the GPU bandwidth by 2.3× (from 37.69 GB/s to 87.75 GB/s).

The experiments are implemented on NVIDIA GTX580 with CUDA 4.0. The results are shown in Fig. 6. It reveals that GPU LU factorization is efficient with matrices whose factorization involves massive (more than 200M with our platforms) flops (floating-point operations). On these matrices, the GPU implementation can achieve on average 7.6× speedup over 1-core CPU and 1.4× over 8-core CPU. In addition, we can see the GPU speedups are approximatively proportional to the flops. This indicates that the speedups will increase when the matrices become larger.

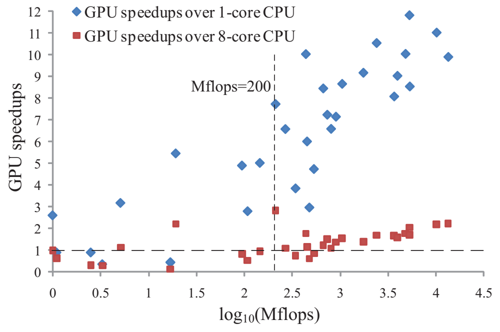


Figure 6. Relation between Mflops and GPU speedups.

IV. Related Work

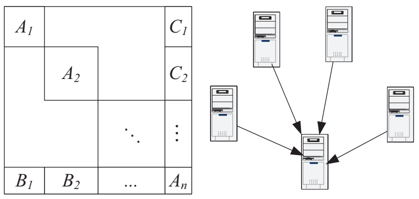
Research on parallel circuit simulation has been active for decades, and there are some classes of parallel approaches.

The traditional parallel strategy is to parallelize the sparse matrix solver. There are some popular software implementations for LU factorization, such as SuperLU [8], [10], KLU [4], UMFPACK [11], PARDISO [12], and so on. Among all the software implementations, only KLU is specially optimized for circuit simulation, however, KLU has no parallel version.

To improve the scalability of the traditional parallel approaches, some circuit partitioning based methods have been proposed [13]–[15]. Most of these methods partition circuits into several small sub-circuits to build the BBD (bordered-block-diagonal) matrices, as shown in Fig. 7(a). However, although the solution of the diagonal blocks can be parallelized independently, the right-bottom block is a serious bottleneck, as BBD-based approach is in a master-slave mode, all the slave machines send data to the master machine (Fig. 7(b)).

There are also some other parallel approaches, such as waveform relaxation [16]. However, it usually requires a grounded capacitor at each node to guarantee convergence and often performs poorly for feedback circuits. Another approach that explores parallelism among different algorithms, called multi-algorithm parallelism [17], however, it needs many redundant computation resources.

Recently, GPU proves useful in many scientific computation fields. Previous works on GPU LU factorization mainly focus on dense matrices [18]–[20]. The performance of these works is promising, however, since circuit matrices are



(a) BBD structure (b) master-slave

Figure 7. The BBD structure and the master-slave parallel mode.

extremely sparse, dense algorithms are not suited. Christen *et al.* mapped PARDISO to GPU [21]. Their work still follows the idea of GPU-based dense LU factorization: they compute dense blocks on GPU and the rest is done on CPU. To our knowledge, there has been no work on GPU-based sparse solver for circuit simulation.

V. Conclusions and Future Work

It is important to accelerate SPICE simulation for modern VLSI design. In our work to date, we have proposed a parallel sparse LU factorization algorithm on shared-memory systems to accelerate the SPICE simulator. The experimental results show that the proposed solver achieves effective speedups over KLU. In addition, the GPU implementation can also achieve more speedups than the CPU code.

Our next step is to develop a circuit partitioning based parallel simulation approach on distributed machines. The traditional SPICE flow can easily reach a bottleneck even by parallel matrix solvers, since the strong data-dependency during numeric LU factorization, memory and cache conflicts, and the inter-processor communication will drastically degrade the parallel performance. Unlike the existing BBD-based approaches, we wish our partitioning based method is a distributed method, and doesn't need to build the matrix for the whole circuit but only needs sub-matrices for each sub-circuit, which is solved by the proposed shared-memory based parallel solver on one computer. The global convergence is achieved by iterations among all the computers. This method has 2 levels of parallelism: intra-computer and inter-computer, which will contribute to achieve more speedup, and the distributed simulation approach will reduce communication among sub-matrices and improve the parallel scalability.

In addition, we will also use CPU/GPU hybrid systems to implement the partitioning based simulation approach, so a scheduling algorithm will be developed to perform a good task assignment scenario to obtain the best performance.

References

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

[2] X. Chen, W. Wu, Y. Wang, H. Yu, and H. Yang, "An escheduler-based data dependence analysis and task scheduling for parallel circuit simulation," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 58, no. 10, pp. 702 –706, oct. 2011.

[3] X. Chen, Y. Wang, and H. Yang, "An adaptive LU factorization algorithm for parallel circuit simulation," *17th Asia and South Pacific Design Automation Conference*, pp. 359–364, 2012.

[4] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems," *ACM Trans. Math. Softw.*, vol. 37, pp. 36:1–36:17, September 2010.

[5] L. Ren, X. Chen, Y. Wang, C. Zhang, and H. Yang, "Sparse LU factorization for circuit simulation on GPU," *to appear in DAC 2012*, 2012.

[6] J. R. Gilbert, "Predicting structure in sparse matrix computations," *SIAM J. Matrix Anal. Appl.*, vol. 15, pp. 62–79, January 1994.

[7] J. R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," *SIAM J. Sci. Statist. Comput.*, vol. 9, pp. 862–874, 1988.

[8] J. W. Demmel, J. R. Gilbert, and X. S. Li, "An asynchronous parallel supernodal algorithm for sparse gaussian elimination," *SIAM J. Matrix Analysis and Applications*, vol. 20, no. 4, pp. 915–952, 1999.

[9] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 3, 2011.

[10] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, "A supernodal approach to sparse partial pivoting," *SIAM J. Matrix Analysis and Applications*, vol. 20, no. 3, pp. 720–755, 1999.

[11] T. A. Davis, "Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method," *ACM Trans. Math. Softw.*, vol. 30, pp. 196–199, June 2004.

[12] O. Schenk and K. Gartner, "Solving unsymmetric sparse systems of linear equations with pardiso," *Computational Science - ICCS 2002*, vol. 2330, pp. 355–363, 2002.

[13] N. Frohlich, B. Riess, U. Wever, and Q. Zheng, "A new approach for parallel simulation of VLSI circuits on a transistor level," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 45, no. 6, pp. 601–613, 1998.

[14] Q. Zhou, K. Sun, K. Mohanram, and D. Sorensen, "Large power grid analysis using domain decomposition," *Design, Automation and Test in Europe, 2006.*, pp. 1–6, 2006.

[15] H. Peng and C.-K. Cheng, "Parallel transistor level circuit simulation using domain decomposition methods," *Asia and South Pacific Design Automation Conference. 2009*, pp. 397–402, 2009.

[16] E. Lelarasmee, A. Ruehli, and A. Sangiovanni-Vincentelli, "The waveform relaxation method for time-domain analysis of large scale integrated circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 1, no. 3, pp. 131–145, 1982.

[17] X. Ye, W. Dong, P. Li, and S. Nassif, "Maps: Multi-algorithm parallel circuit simulation," *Computer-Aided Design, 2008.*, pp. 73–78, 2008.

[18] V. Volkov and J. Demmel, "Lu, qr and cholesky factorizations using vector capabilities of gpus," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-49, May 2008.

[19] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid gpu accelerated manycore systems," *Parallel Comput.*, vol. 36, pp. 232–240, June 2010.

[20] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with gpu accelerators," *IEEE International Symposium on Parallel Distributed Processing Workshops and Phd Forum IPDPSW*, pp. 1–8, 2010.

[21] M. Christen, O. Schenk, and H. Burkhart, "General-purpose sparse matrix building blocks using the nvidia cuda technology platform," 2007.