GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling

Xiaoming Chen, *Student Member, IEEE*, Ling Ren, Yu Wang, *Member, IEEE*, and Huazhong Yang, *Senior Member, IEEE*

Abstract—The sparse matrix solver by LU factorization is a serious bottleneck in Simulation Program with Integrated Circuit Emphasis (SPICE)-based circuit simulators. The state-of-the-art Graphics Processing Units (GPU) have numerous cores sharing the same memory, provide attractive memory bandwidth and compute capability, and support massive thread-level parallelism, so GPUs can potentially accelerate the sparse solver in circuit simulators. In this paper, an efficient GPU-based sparse solver for circuit problems is proposed. We develop a hybrid parallel LU factorization approach combining task-level and data-level parallelism on GPUs. Work partitioning, number of active thread groups, and memory access patterns are optimized based on the GPU architecture. Experiments show that the proposed LU factorization approach on NVIDIA GTX580 attains an average speedup of 7.02× (geometric mean) compared with sequential PARDISO, and 1.55× compared with 16-threaded PARDISO. We also investigate bottlenecks of the proposed approach by a parametric performance model. The performance of the sparse LU factorization on GPUs is constrained by the global memory bandwidth, so the performance can be further improved by future GPUs with larger memory bandwidth.

Index Terms—Graphics processing unit, parallel sparse LU factorization, circuit simulation, performance model

1 Introduction

The Simulation Program with Integrated Circuit Emphasis (SPICE) [1] is the most widely used circuit simulation kernel for transistor-level simulation in IC design and verification. In recent years, the rapid development of very large scale integrations (VLSI) presents great challenges to the performance of SPICE-based simulators. In modern VLSIs, circuit matrices after post-layout extraction can easily reach a dimension of several million. Traditional SPICE-based simulators may take days or even weeks to perform transient simulations. There are two bottlenecks in a SPICE-based simulator: the sparse matrix solver by LU factorization and model evaluation. The two steps are repeated for thousands of rounds in a transient simulation, as shown in Fig. 1.

Parallelization of model evaluation is straightforward, but the sparse solver is difficult to parallelize because of the strong data dependence during LU factorization, the irregular structure of circuit matrices, and the low computation-to-memory ratio in sparse LU factorization. To accelerate the sparse solver in SPICE-based simulators, some parallel software solvers, such as ShyLU [2] and NICSLU [3], [4], [5], are developed. They have been proved efficient for circuit matrices on multicore CPUs.

- *X. Chen, Y. Wang and H. Yang are with the Department of Electronic Engineering, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing, China. E-mail: chenxm05@mails.tsinghua.edu.cn, yu-wang@tsinghua.edu.cn, yanghz@tsinghua.edu.cn.*
- *L. Ren is with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. E-mail: renling@mit.edu.*

Manuscript received 16 Sept. 2013; revised 15 Jan. 2014; accepted 26 Feb. 2014. date of publication 17 Mar. 2014; date of current version 6 Feb. 2015. Recommended for acceptance by F. Mueller.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2312199

In the last decade, state-of-the-art Graphics Processing Units (GPU) have been proved useful in many scientific computing fields. GPUs have hundreds of cores, large register files, and high memory bandwidth, etc. Collectively, these computing and memory resources provide a massive thread-level parallelism. Thus, GPUs potentially provide a better solution than multicore CPUs to accelerate SPICE-based circuit simulators. There have been some studies on GPU-based model evaluation [6], [7], [8]. Although some direct sparse solvers which are based on multifrontal [9] or supernodal algorithms have been developed on GPUs [10], [11], [12], [13], [14], [15], currently there is no work on GPU-based direct solver targeted at circuit matrices. Circuit matrices are quite different from matrices from other applications. Circuit matrices are highly asymmetric and sparse, so multifrontal and supernodal solvers are proved to perform poorly [16]. Therefore, there raises a natural question whether solvers targeted at circuit matrices can be accelerated by GPUs, and how we can map a sparse solver from multicore architectures to manycore architectures. In this paper we develop a parallel sparse solver on GPUs and experiments prove that it is efficient for circuit problems. This paper is extended from our conference paper [17]. This paper presents more results and develops a performance model compared with [17]. We make the following contributions in this paper.

- A GPU-based sparse solver for circuit problems is developed, which is based on parallel LU numeric factorization without pivoting. Different from the existing GPU-based direct solvers, our GPU solver does not use dense kernels. This feature makes our solver be suitable for circuit matrices. More parallelism is explored for the manycore architecture of GPUs. Task-level parallelism is proposed in CPU-version NICSLU [3], [4], [5], to perform an

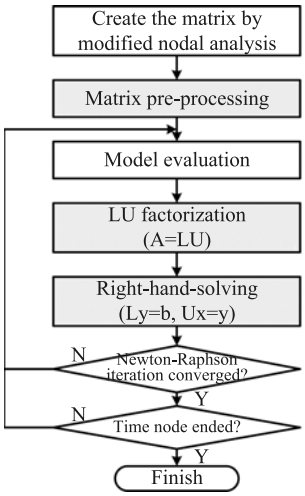


Fig. 1. Flow of SPICE-based transient simulation.

inter-vector parallel algorithm, which is not sufficient for the thousands of threads running concurrently on a GPU, so intra-vector parallelism is also exposed on GPUs. Therefore, the GPU implementation is a hybrid approach combining both task-level and data-level parallelism. The implementation is optimized based on the GPU architecture.

- We present a parametric performance model to analyze the bottlenecks of the proposed GPU-based LU factorization. We investigate the relation between the achieved GPU performance and the GPU parameters using the proposed model. Experimental results indicate that the performance of the proposed sparse LU factorization on GPUs is constrained by the global memory bandwidth of GPUs. The performance model is universal and can also be used for other GPU applications.

The rest of the paper is organized as follows. Section 2 shows some related work. Some backgrounds are introduced in Section 3. Section 4 presents the proposed GPU-based sparse LU factorization approach and optimization strategies in detail. We show the experimental results and analysis in Section 5. A parametric performance model is developed in Section 6. Finally Section 7 concludes the paper.

2 Related Work

There are many LU factorization-based sparse solvers developed on CPUs. SuperLU [18] incorporates supernodes in the Gilbert-Peierls (G-P) left-looking algorithm [19], and the Basic Linear Algebra Subroutines (BLAS) [20] library is used to compute dense subblocks in supernodes to enhance the compute capability. Another solver PARDISO [21] also utilizes supernodes. However, it is hard to form large supernodes in highly sparse matrices such as circuit matrices. Hence KLU [16] which is specially targeted at circuit matrices, directly adopts the G-P left-looking algorithm without supernodes. KLU is generally faster than BLAS-based sequential solvers for circuit matrices [16]. SuperLU and PARDISO have parallel packages, but KLU doesn't. NICSLU is a parallel version of the G-P left-looking algorithm, and it works efficiently for circuit matrices on multicore machines [3], [4], [5].

TABLE 1
Summary of Existing GPU-Based Direct Sparse Solvers

ref.	matrix type	precision	average speedup
[10]	asymmetric	single	about 2× vs. sequential PARDISO [21]
[11]	symmetric	double	2.9× vs. 2-threaded ANSYS [22]
[12]	asymmetric	double	about 2.5× vs. sequential UMFPACK [23]
[13]	symmetric	single[a]	7× vs. sequential WSMP [24] 15× by 2 GPUs vs. sequential WSMP
[14]	symmetric	single	1.97× vs. sequential CPU code[b]
[15]	symmetric	single	5.91× vs. sequential CPU code[b] 1.34× vs. 8-threaded CPU code[b]

[a] This work compared single-precision GPU implementation with double-precision WSMP
[b] The CPU code means their inhouse CPU solver

GPU-based dense LU factorization has been studied in [25], [26], [27], [28], which all belong to the MAGMA project [29]. The performance of these approaches is very promising, up to 80 percent of the GPU's theoretic peak performance [26]. NVIDIA has developed Compute Unified Device Architecture (CUDA)-based BLAS library for accelerating dense linear algebra computing [30].

In recent years, some researches have developed sparse direct solvers on GPUs [10], [11], [12], [13], [14], [15]. PARDISO is mapped onto NVIDIA GPUs using single-precision floating-point kernels [10]. It uses the CUBLAS kernels to compute dense subblocks on GPUs. Other approaches [11], [12], [13], [14], [15] are all based on the multifrontal algorithm [9], and they all use dense kernels to compute dense subblocks during sparse LU factorization. We summarize these GPU-based direct solvers in Table 1. CHOLMOD [31], which is a Cholesky factorization-based direct solver, now can be accelerated by CUDA BLAS too. These GPU solvers all involve offloading the time-consuming dense kernels to GPUs in order to improve the overall performance, so they are suitable for matrices which are denser than circuit matrices but may not be suitable for highly sparse matrices. To our knowledge, currently there is no published work on GPU-based direct solver for circuit matrices without using BLAS.

3 Backgrounds

This section briefly introduces some backgrounds about the left-looking algorithm, the two-mode scheduling algorithm in NICSLU, the NVIDIA GPU architecture, and the CUDA runtime model.

3.1 Left-Looking Algorithm

The pseudo code of the sparse left-looking algorithm [19] (without pivoting) is shown in Algorithm 1. This algorithm factorizes a matrix by processing columns in sequence. When factorizing column i, some columns on the left side will be used to update column i (i.e., data dependence). The dependence is determined by the structure of U. The columns that column i depends on are $\{j : U(j,i) \neq 0, j < i\}$. The left-looking algorithm is well suited for cache-based and shared-memory machines

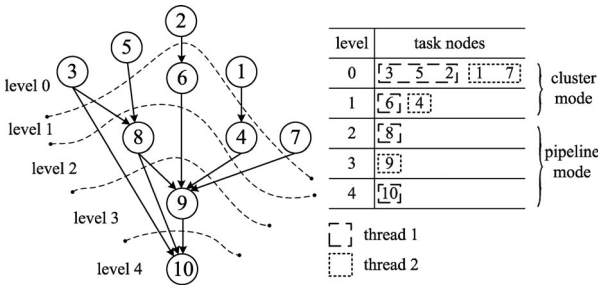


Fig. 2. Two-mode scheduling in NICSLU.

because it writes less data to main memory, compared with the right-looking algorithm.

Algorithm 1 Sparse left-looking algorithm.

```
1: //x is a column vector of length n
2: for i = 1 : n do
3:     x = A(:, i);
4:     for j = 1 : i − 1 where U(j, i) ≠ 0 do
5:         //vector multiplication-and-add
6:         x(j + 1 : n) − = x(j) × L(j + 1 : n, j);
7:     end for
8:     U(1 : i, i) = x(1 : i);
9:     L(i : n, i) = x(i : n)/x(i);
10: end for
```

3.2 Two-Mode Scheduling in NICSLU

Based on the dependence analysis in the previous section, the dependence between columns are described by a direct acyclic graph (DAG) [4], as shown in Fig. 2. The DAG is further partitioned into different levels. A two-mode scheduling algorithm combining cluster mode and pipeline mode is proposed in NICSLU [3], [4], [5] to perform a parallel LU factorization algorithm. Cluster mode is used for levels that have many columns. These levels are processed in sequence but columns in one level are processed in parallel. For each level, columns are evenly assigned to work threads. Pipeline mode is used for the rest of levels that have fewer columns in each level. These levels with dependence are computed concurrently by a pipeline-like algorithm. The proposed GPU-based sparse LU factorization approach is based on the two-mode scheduling algorithm. The details of the two modes can be found in [4].

3.3 GPU Architecture and CUDA Runtime Model

A rough diagram of the NVIDIA GPU architecture [32] is shown in Fig. 3. A GPU consists of some streaming multiprocessors (SM), and each SM is composed of many streaming processors (SP), L1 cache, shared memory, and some special functional units. In NVIDIA Fermi architecture,

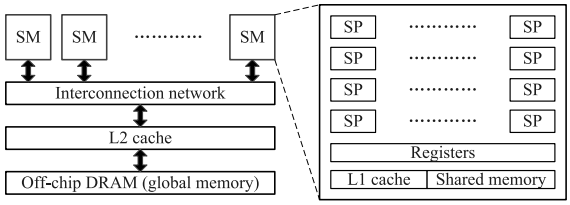


Fig. 3. Rough diagram of the NVIDIA GPU architecture.

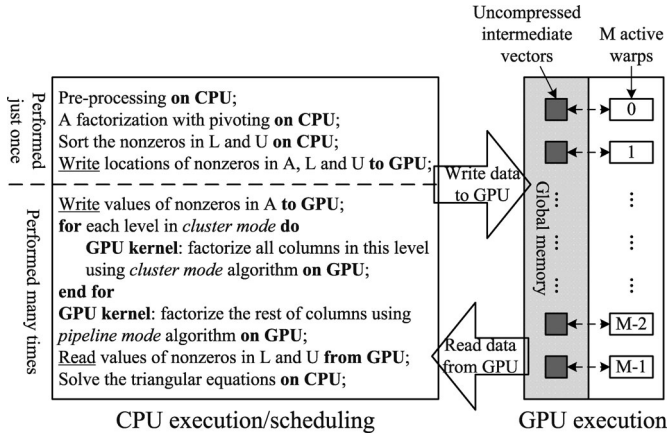


Fig. 4. Framework of the sparse solver on GPU.

each SM has 32 SPs; in the latest Kepler architecture, each SM has 192 SPs. The off-chip memory is called global memory, which has much longer access latency than on-chip storages.

NVIDIA GPUs execute programs in a single-instruction-multiple-thread (SIMT) manner. SMs execute threads in groups of 32 concurrent threads called a warp. Several warps are grouped into a thread block. One thread block is executed on one SM, and one SM can hold several concurrent thread blocks. Data can be easily shared within a thread block through the on-chip shared memory.

4 Sparse LU Factorization on GPUs

In this section, we present the GPU-based sparse LU factorization approach and the optimization strategies in detail. First, an overall framework is presented, and then optimization strategies are discussed.

4.1 Overall Framework

The three shaded rectangles in Fig. 1 are called a sparse solver. Fig. 4 shows the overall framework of the proposed GPU-based sparse solver. Actually it is a CPU-GPU hybrid solver, in which the most time-consuming numeric LU factorization is implemented on the GPU, the host CPU performs the scheduling for the GPU and executes the other steps. The proposed solver focuses on GPU-based numeric factorization to accelerate circuit simulation.

The pre-processing step performs row/column reordering to reduce fill-ins. The HSL_MC64 algorithm [33], [34] is also performed in pre-processing to pre-scale the matrix to enhance the numeric stability. During the iterations of circuit simulation, the structure of the circuit matrix is fixed, so only the first factorization is performed with pivoting, subsequent factorizations will use the fixed pivoting order and fixed structure of the LU factors obtained by the first factorization [16]. In circuit simulation, although values of matrix elements change during iterations, they cannot change much according to the physics of circuits, so this approach can ensure the numeric stability of subsequent factorizations. This flow was claimed to be effective in digital circuit simulation [35]. A subsequent factorization without pivoting is also called a re-factorization. Pre-processing is executed just once and its time cost is generally much less

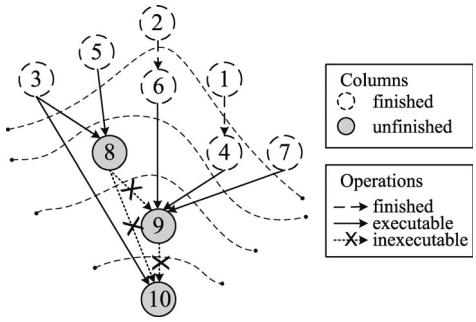


Fig. 5. Parallelism in left-looking algorithm.

than factorization, so its time cost is not of interest. The right-hand-solving step costs much less time than factorization (a test on NICSLU [3], [4], [5] shows that the right-hand-solving step costs on average less than 1 percent of the sequential factorization time), so LU re-factorization by the sparse left-looking algorithm is the most time-consuming step.

The GPU-based numeric re-factorization uses the parallelization strategies mentioned in Section 3.2. In cluster mode, each level launches a GPU kernel; while in pipeline mode, only one kernel is launched. Before each re-factorization executed on the GPU, the values of A are written to GPU's global memory, and when the GPU finishes one re-factorization, the values of the LU factors are copied from GPU's global memory to the host memory. The right-hand-solving step is executed on the host CPU.

4.2 Exploring More Parallelism

As shown in Algorithm 1, the core operation in the sparse left-looking algorithm is vector multiplication-and-add (MAD) (line 6). Cluster mode and pipeline mode are proposed in [4] to describe the parallelism between vector MAD operations. Take Fig. 5 as an example to explain this level of parallelism. Column 1∼7 are finished, and column 8∼10 are being processed. The MAD operations represented by solid arrows are executable. Parallelism exists in these operations, though the numeric updates to a same column must be executed in a strict order. This granularity of parallelism is called inter-vector parallelism, which can also be called task-level parallelism.

The inter-vector/task-level parallelism alone cannot take full advantage of state-of-the-art GPU's high memory bandwidth and compute capability. We expose another intrinsic granularity of parallelism in sparse LU factorization: intra-vector parallelism, which is also called data-level parallelism. This level of parallelism means that one vector is computed by multiple threads simultaneously, which matches the SIMT nature of GPUs. More specifically, each of the vector operations (lines 3, 6, 8 and 9) shown in Algorithm 1 is computed by multiple concurrent threads. Now we consider how to partition the workload to fully utilize the GPU resources. In this point, several factors should be considered.

For convenience, threads that process a same column are called a virtual group. All threads in a virtual group operate on elements in a same column and must synchronize. The size of a virtual group should be carefully decided. First, virtual groups should not be too large. This is because if the size of a virtual group is larger than the number of nonzeros in a column, some threads will idle. So smaller virtual groups can reduce idle threads. However, virtual groups should not be too small either. There are two reasons for this point. First, too small virtual groups result in too few threads in total (the number of virtual groups is limited by the storage space and cannot be very large), leading to too few concurrent columns, which cannot fully utilize the GPU's compute capability. Second, GPUs schedule threads in a SIMT manner. If threads within a warp diverge, different branches are executed serially. Different virtual groups process different columns and hence often diverge. Thus, too small virtual groups increase divergence within SIMT threads.

In cluster mode, columns are very sparse, so while ensuring enough threads in total, virtual groups should be as small as possible to minimize idle threads. In pipeline mode, columns usually contain enough nonzeros for a warp or several warps, so the size of virtual groups matters little in the sense of reducing idle threads. Taking all these factors into consideration, we use one warp as one virtual group in both modes. This strategy not only reduces divergence within SIMT threads, but also saves synchronization costs, since synchronization of threads within a warp is implicitly guaranteed by the SIMT nature of GPUs.

The implementation details and optimization strategies including synchronization for timing order and memory access pattern optimization can be found in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.2312199.

TABLE 2
Specifications of Computing Platforms

	CPU	GPU	
Device	Intel Xeon E5-2690 ×2	NVIDIA Tesla K20x[a]	NVIDIA GTX580
#cores	2×8 =16	14 SMs =2688 SPs	16 SMs =512 SPs
L1 cache	64 KB	48 KB/SM	48 KB/SM
L2 cache	256 KB/core	1.5 MB	768 KB
L3 cache	20 MB	N/A	N/A
Clock frequency	2.9∼3.8 GHz	732 MHz	1544 MHz
Peak power	135 W×2	235 W	244 W
Peak bandwidth[b]	51.2 GB/s	250 GB/s[c]	192.4 GB/s
DRAM	128 GB	6 GB	1.5 GB

[a] K20x is with error correction code (ECC) on.
[b] It is the off-chip memory bandwidth.
[c] Since ECC is on, the effective peak bandwidth is 218.75 GB/s.

5 Experimental Results

5.1 Experiment Setup

All the experiments are tested on three platforms which are shown in Table 2. GPU codes are programmed using CUDA 5.0 [32] and compiled with `-O3` optimization. Twenty-three circuit matrices from University of Florida Sparse Matrix Collection [36] are used to evaluate the GPU solver. The performance of LU factorization on GPUs is compared with NICSLU [3], [4], [5] (optimized by hand-written SSE2 instructions and compiled with `-O2`

TABLE 3
Performance of LU Factorization on GPUs, and the Speedups over KLU (re-factorization) and PARDISO

benchmark[a]	N ($\times 10^3$)	NNZ ($\times 10^3$)	average flop per nonzero	NVIDIA GTX580 time[b] (s)	NVIDIA GTX580 bandwidth[c] (GB/s)	NVIDIA GTX580 speedup vs. KLU (T=1)	NVIDIA GTX580 speedup vs. PARDISO (T=1)	NVIDIA GTX580 speedup vs. PARDISO (T=16)	NVIDIA K20x time[b] (s)	NVIDIA K20x bandwidth[c] (GB/s)	NVIDIA K20x speedup vs. KLU (T=1)	NVIDIA K20x speedup vs. PARDISO (T=1)	NVIDIA K20x speedup vs. PARDISO (T=16)
add32	5.0	23.9	2.0	0.0004	3.9	1.10	68.27	27.14	0.0007	2.2	0.63	39.14	15.56
hcircuit	105.7	513.1	3.8	0.0065	8.4	1.48	23.72	5.17	0.0099	5.5	0.98	15.63	3.41
add20	2.4	17.3	7.5	0.0004	6.8	1.05	14.12	28.17	0.0006	4.1	0.63	8.49	16.95
bcircuit	68.9	375.6	11.7	0.0057	32.2	3.15	23.16	4.60	0.0074	25.0	2.44	17.94	3.57
circuit_4	80.2	307.6	12.4	0.0185	4.8	0.54	5.96	1.70	0.0316	2.8	0.32	3.49	0.99
scircuit	171.0	958.9	21.2	0.0208	36.1	3.61	17.80	2.77	0.0274	27.4	2.74	13.48	2.10
rajat03	7.6	32.7	43.6	0.0030	34.3	1.90	4.94	3.78	0.0043	23.4	1.29	3.36	2.58
coupled	11.3	98.5	66.2	0.0140	25.4	1.40	2.43	1.15	0.0190	18.7	1.03	1.80	0.85
rajat15	37.3	443.6	81.4	0.0417	47.9	3.53	3.30	0.61	0.0449	44.4	3.27	3.06	0.57
rajat18	94.3	485.1	86.1	0.0506	22.4	0.25	5.53	1.24	0.0764	14.8	0.17	3.66	0.82
raj1	263.7	1302.5	98.5	0.2274	45.0	289.48	3.35	0.47	0.3007	34.0	218.99	2.54	0.35
transient	178.9	961.8	113.1	0.1144	28.3	1.81	3.31	0.62	0.1772	18.2	1.17	2.14	0.40
rajat24	358.2	1948.2	143.6	0.2231	36.6	84.16	3.63	0.60	0.3541	23.1	53.01	2.28	0.37
asic_680k	682.9	3871.8	151.6	0.5936	23.9	1.83	53.07	4.53	1.0134	14.0	1.07	31.09	2.65
onetone2	36.1	227.6	159.2	0.0447	62.7	7.66	3.29	0.75	0.0547	51.2	6.26	2.68	0.62
asic_680ks	682.7	2329.2	185.1	0.1506	86.2	5.27	164.77	14.59	0.1370	94.7	5.79	181.12	16.04
asic_320k	321.8	2635.4	218.1	0.4048	42.9	7.54	6.34	1.30	0.6145	28.2	4.97	4.18	0.86
asic_100k	99.3	954.2	277.6	0.2735	57.3	4.52	1.49	0.29	0.3605	43.5	3.43	1.13	0.22
asic_320ks	321.7	1827.8	282.9	0.1681	114.8	18.77	14.07	2.53	0.1686	114.4	18.71	14.02	2.53
asic_100ks	99.2	578.9	383.0	0.1667	117.6	9.37	2.12	0.39	0.1570	124.8	9.95	2.25	0.41
onetone1	36.1	341.1	441.4	0.1610	116.9	54.63	2.34	0.50	0.1599	117.7	55.00	2.36	0.50
g2_circuit	150.1	726.7	502.5	1.0963	128.5	7.12	0.80	0.09	0.8901	158.3	8.77	0.98	0.12
twotone	120.8	1224.2	942.3	0.9838	154.8	47.23	3.95	0.53	0.8286	183.7	56.07	4.69	0.63
arithmetic-mean					**53.81**	**24.24**	**18.77**	**4.50**		**51.05**	**19.86**	**15.72**	**3.18**
geometric-mean						**4.95**	**7.02**	**1.55**			**3.86**	**5.46**	**1.21**

[a] onetone1, onetone2, twotone are from frequency-domain simulation, other matrices are from time-domain simulation.
[b] GPU time includes the numeric factorization time and the time cost of data transfers between host and device.
[c] The bandwidth values are calculated by $\frac{\text{total size of all memory accesses}}{\text{factorization time}}$.

optimization), KLU [16] (compiled with -O3 optimization), and PARDISO [21] from the Intel Math Kernel Library (MKL). We show the main results here, some additional results are shown in Appendix B, available online.

5.2 Results of LU Factorization on GPUs

5.2.1 Performance and Speedups

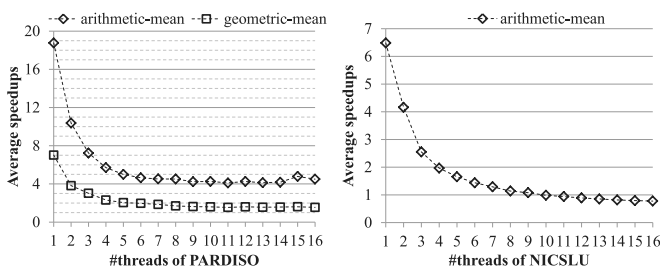
Table 3 shows the results of the proposed LU factorization approach on NVIDIA GTX580 and NVIDIA K20x, and the speedups compared with KLU and PARDISO which are implemented on the host CPU. All the benchmarks are sorted by the average number of floating-point operations (flop) per nonzero ($\frac{flop}{NNZ(L+U-I)}$). The listed time in Table 3 is only for numeric re-factorization, excluding pre-processing and right-hand solving. Some data have to be transferred between the host memory and the device memory in every re-factorization (see Fig. 4), time for these transfers are included in the GPU re-factorization time.

Our GPU-based LU re-factorization outperforms KLU re-factorization for most of the benchmarks. The speedups tend to be higher for matrices with bigger $\frac{flop}{NNZ(L+U-I)}$. This indicates that GPUs are more suitable for problems that are more computationally demanding. The average speedup is 24.24× achieved by GTX580 and 19.86× achieved by K20x. Since the speedups differ greatly for different matrices, the geometric means of speedups are also listed in Table 3.

Compared with sequential PARDISO, the GPU-based LU re-factorization is also faster than PARDISO for most of the benchmarks. The speedups tend to be higher for matrices with smaller $\frac{flop}{NNZ(L+U-I)}$. PARDISO utilizes BLAS to compute dense subblocks so it performs better for matrices with bigger $\frac{flop}{NNZ(L+U-I)}$. The average speedup of the GPU-based LU re-factorization is 18.77× achieved by GTX580 and 15.72× achieved by K20x. The geometric mean of the speedups is 7.02× achieved by GTX580.

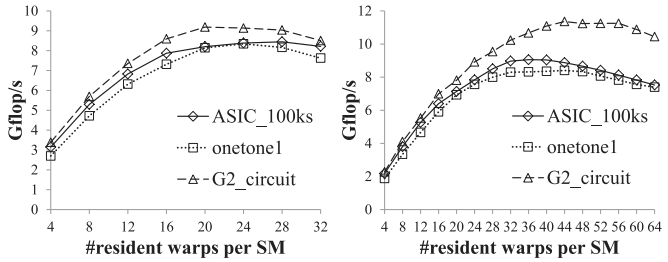
Compared with 16-threaded PARDISO, the GPU-based LU re-factorization is faster than PARDISO for about half of the benchmarks and slower for the other half. The average speedup is 1.55× (geometric mean) achieved by GTX580 and 1.21× achieved by K20x. Fig. 6a shows the average speedup of the GPU-based LU re-factorization (implemented on GTX580), compared with PARDISO using different number of threads. On average, the GPU-based LU re-factorization approach outperforms multi-threaded PARDISO.

Fig. 6b shows the average speedup of the GPU-based LU re-factorization (implemented on GTX580) compared with NICSLU re-factorization. Since the speedups differ little for different matrices in these results, only arithmetic means are presented. Fig. 6b indicates that the performance



(a) GPU vs. PARDISO. (b) GPU vs. NICSLU.

Fig. 6. Average speedups. The GPU results are obtained on GTX580.



(a) NVIDIA GTX580. (b) NVIDIA K20x.

Fig. 7. Achieved GPU performance, using different number of resident warps per SM.

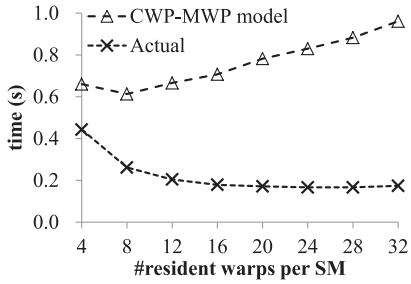


Fig. 8. Predicted LU factorization time by the CWP-MWP model [37], for ASIC_100 ks, obtained on GTX580.

of the GPU-based LU re-factorization is similar to that of 10-threaded NICSLU. When the number of threads of NICSLU is larger than 10, NICSLU will outperform the GPU implementation. The average speedup of the GPU-based LU re-factorization compared with 16-threaded NICSLU is 0.78×.

Although the theoretic peak performance of K20x is about 6× larger than that of GTX580 (for double-precision), the results in Table 3 indicate that the achieved performance on K20x and GTX580 is similar. The main reason is that performance of sparse LU factorization on GPUs is bandwidth-bound. We will explain this point in Section 6. The effective global memory bandwidth of K20x (with ECC on) is very close to that of GTX580, so the higher compute capability of K20x cannot be explored.

5.2.2 Bottleneck Analysis

The achieved average bandwidth of all benchmarks is 53.81, 51.05, and 92.75 GB/s, obtained on NVIDIA GTX580, NVIDIA K20x, and Intel E5-2690×2 (implemented by 16-threaded NICSLU), respectively. For CPU execution, some memory accesses are served by caches, so the bandwidth may greatly exceed the theoretic peak of the main memory. For different platforms, factors that restrict the performance are different. For CPUs, large caches and high clock frequency can greatly improve the computational efficiency. However, for GTX580 and K20x, disabling the L1 cache just results in less than 5 percent performance loss. It is difficult to evaluate the effect of GPU's L2 cache, since there is no way to disable the L2 cache of current NVIDIA GPUs programmed using CUDA. Since GPU's L2 cache is much smaller than modern CPU's last level cache (LLC) and there are much more cores sharing the LLC on a GPU, it is expected that the LLC has less effect on GPUs than on CPUs. The main bottleneck of the proposed LU re-factorization approach on GPUs is the peak global memory bandwidth.

We have found that processing too many columns concurrently may decrease the performance on GPUs. Fig. 7 shows the giga floating-point operations per second (Gflop/s) of three matrices obtained on GTX580 and K20x, using different number of resident warps per SM. The best performance is attained with about 24 (for GTX580)/40 (for K20x) resident warps per SM, rather than with maximum resident warps per SM. Although the reported bandwidth in Table 3 does not reach the peak bandwidth of the GPU, Fig. 7 indicates that we have already fully utilized the bandwidth with 24 (for GTX580)/40 (for K20x) resident warps per SM. It also means that the actual bandwidth is saturated with 24 (for GTX580)/40 (for K20x) resident warps per SM. When warps become more, the performance is constrained and becomes lower. To explain this phenomenon in detail, we build a parametric performance model in Section 6.

6 Performance Evaluation of LU Factorization on GPUs

In this section, a parametric performance model is built to explain the phenomenon shown in Fig. 7 in detail. With this model, we can investigate the relation between the achieved GPU performance and the GPU parameters such as the global memory bandwidth and the number of warps, to help us better understand the bottlenecks of the solver and choose the optimal number of work threads on the given hardware platform. Actually the proposed model is universal and can also be used for other GPU applications.

A CWP-MWP (CWP and MWP are short for computation warp parallelism and memory warp parallelism) model is proposed in [37]. In that model, a warp that is waiting for memory data is called a memory warp, and MWP represents the number of memory warps that can be handled concurrently on one SM. We have attempted to use the CWP-MWP model to evaluate the performance on NVIDIA GTX580, but it gets incorrect results, as shown in Fig. 8. The reason is that MWP in the CWP-MWP model is very small (4∼7 for different matrices), so when the number of resident warps per SM exceeds MWP, the execution time is greatly affected by the global memory latencies, leading to unreasonable predicted results.

In this paper, a better performance model is proposed, which improves the CWP-MWP model at the following points.

- Computation of MWP is changed. MWP is hardware-specific and directly constrained by the peak bandwidth of the global memory.
- Concepts of MWP_req and $warp\_exp$ which have physical meanings are introduced to illustrate the different cases of warp scheduling.
- The effect of the L2 cache is considered.
- The CWP-MWP model performs static analysis on the compiled code to predict performance. However, in the pipeline mode (see Appendix A.1, available

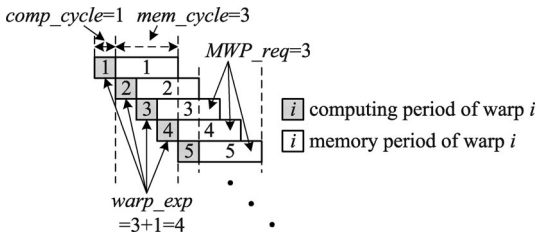


Fig. 9. Illustration of warp scheduling and some parameters.

online) of sparse LU factorization, each warp waits for some other warps to finish, which is a run-time behavior and the time cost of waiting cannot be determined at compilation time. Our solution is based on a DAG analysis to avoid such dynamic behaviors.

Based on the performance model, the execution cycles of each task (i.e., node) in a DAG can be calculated, and then we calculate the execution cycles of the whole DAG by a well-known method which finds the critical path of the DAG (see Appendix C.2, available online, for details).

6.1 Parametric Performance Model

GPUs hide global memory latencies by warp switches, as shown in Fig. 9. When a warp invokes a global memory request, the SIMT engine switches to another warp that is not waiting for data to execute. Our model is based on the basic warp scheduling in GPUs.

6.1.1 Parameters

The parameters used in the model are shown in Table 4. In the following contents, some important parameters are introduced.

mem_cycle represents the average cycles of each memory period as marked in Fig. 9. Considering the cache miss rate $miss\_rate$, mem_cycle is expressed as a weighted combination of $global\_cycle$ and $cache\_cycle$:

$$mem\_cycle = global\_cycle \times miss\_rate + cache\_cycle \times (1 - miss\_rate), \quad (1)$$

$miss\_rate$ is measured on the target GPU using a micro-benchmark, see Appendix C.1, available online, for details.

MWP represents the maximum number of memory warps that can be served concurrently on one SM. It is hardware-specific and directly constrained by the peak bandwidth of the global memory:

$$MWP = \frac{peak\_bandwidth \times mem\_cycle}{\#SM \times freq \times load\_bytes\_per\_wap}, \quad (2)$$

where $peak\_bandwidth$, $\#SM$ and $freq$ are physical parameters of a GPU. $load\_bytes\_per\_wap$ means the average data size accessed from the global memory in each memory period. Since in a MAD operation, there are 1 `int` reading operation, 2 `double` reading operations, and 1 `double` writing operation, the equivalent data size accessed from the global memory considering the cache effect is expressed as:

$$load\_bytes\_per\_wap = \frac{size\,of\,(int) + 3 \times size\,of\,(double)}{4} \times 32 \times miss\_rate. \quad (3)$$

MWP_req is the number of concurrent memory warps requested by the invoked warps on one SM. As can be seen from Fig. 9, if the GPU can support $\frac{mem\_cycle}{comp\_cycle}$ concurrent memory warps on one SM, all the memory requests can be served without delaying. So MWP_req is given by

$$MWP\_req = MIN\left(\frac{mem\_cycle}{comp\_cycle}, \#warp\right), \quad (4)$$

$warp\_exp$ means the expected number of concurrent warps on one SM such that the memory latencies can be completely hidden. Still take Fig. 9 as an example, to hide the latency of one memory period, there should be at least $\frac{mem\_cycle}{comp\_cycle} + 1$ warps running concurrently, so

$$warp\_exp = \frac{mem\_cycle}{comp\_cycle} + 1 \quad (5)$$

TABLE 4
Parameters Used in the Model, for GTX580

parameter	meaning	value	obtained
MWP	memory warp parallelism	N/A	Eq. (2)
MWP_req	number of requested memory warps on one SM	N/A	Eq. (4)
$\#repeat$	number of iterations of a warp	N/A	code analysis
$global\_cycle$	global memory latency	500∼550	formulas in [37]
$cache\_cycle$	L2 cache latency	≈160	measurement
$miss\_rate$	miss rate of L2 cache	N/A	measurement
mem_cycle	equivalent cycles of each memory period	N/A	Eq. (1)
$comp\_cycle$	average cycles of each computing period	<10	PTX[a] code analysis
$\#warp$	number of resident warps per SM	4∼32	assigned
$warp\_exp$	expected number of warps on one SM to hide the memory latency	N/A	Eq. (5)
mem_trans	average memory transactions of each memory period	N/A	method in [37]

[a] Parallel thread execution (PTX) is a assembly-like language in CUDA.

6.1.2 Three Cases in Warp Scheduling

Based on the above parameters, there are three cases in the warp scheduling of GPUs, according to the number of resident warps per SM, as illustrated in Fig. 10.

Case 1 (Fig. 10a). if $\#warp \geq warp\_exp$ and $MWP \geq MWP\_req$, there are enough warps running to hide global memory latencies and the memory bandwidth is also sufficient to support all the concurrent memory warps. In the time line, each memory period follows the corresponding computing period closely and no delay occurs. The total runtime is mainly determined by the computing periods. For a given warp that is repeated for $\#repeat$ iterations ($\#repeat = 3$ in Fig. 10a), the number of total execution cycles is:

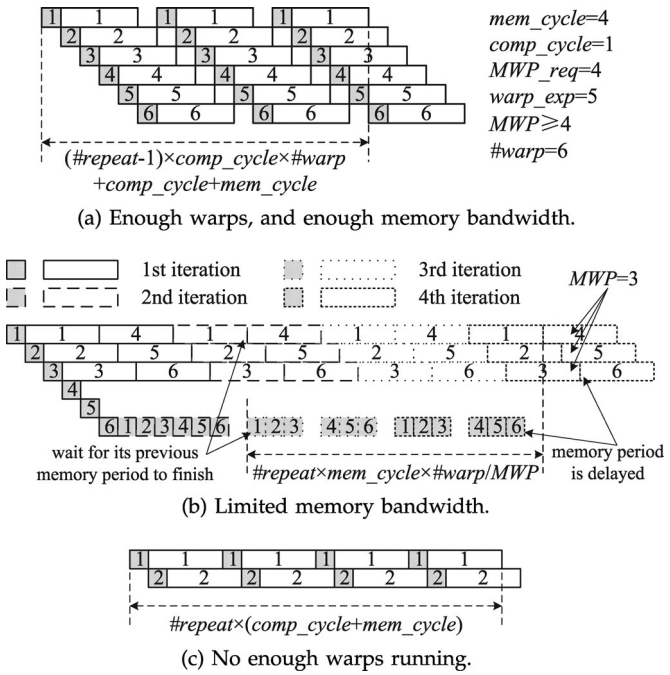


Fig. 10. Illustration of GPU execution.

$$\begin{aligned} &total\_cycle\_per\_warp \\ &\quad = (\#repeat - 1) \times comp\_cycle \times \#warp \\ &\qquad + comp\_cycle + mem\_cycle. \end{aligned} \tag{6}$$

Case 2 (Fig. 10b). If $MWP < MWP\_req$, the limited memory bandwidth cannot support all the memory warps concurrently. For example, in Fig. 10b, $MWP = 3$ and $MWP\_req = 4$, so the requested memory warps cannot be handled concurrently and they are served by queueing. Each memory period cannot closely follow its corresponding computing period, leading to some delay in the subsequent computing periods and memory periods. Note that Fig. 10b is plotted from the beginning of execution, the first two iterations are a little different from the subsequent iterations. The number of execution cycles of a warp is mainly determined by memory periods:

$$\begin{aligned} &total\_cycle\_per\_warp \\ &\quad = \#repeat \times mem\_cycle \times \frac{\#warp}{MWP}. \end{aligned} \tag{7}$$

Case 3 (Fig. 10c). if $\#warp < warp\_exp$ and $MWP \geq MWP\_req$, warps are too few to hide the memory latency (the memory bandwidth is sufficient to support all the concurrent memory warps). This case is simple and we have

$$\begin{aligned} &total\_cycle\_per\_warp \\ &\quad = \#repeat \times (comp\_cycle + mem\_cycle). \end{aligned} \tag{8}$$

6.2 Discussion

We have validated the accuracy of the proposed model, please see Appendix C.3, available online, for details. Here we explain why using maximum resident warps per SM is not the best choice on GPUs, taking GTX580 as an example.

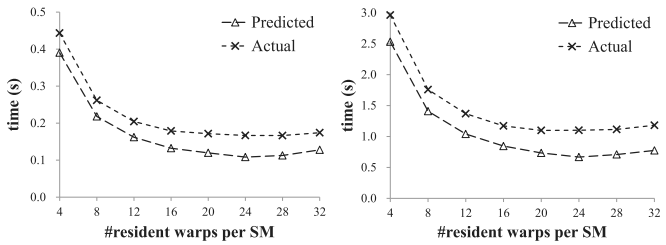


Fig. 11. Predicted time and actual time (on GTX580), with different $\#warp$.

We show the results of two matrices in Fig. 11. The proposed performance model indicates that $MWP \approx 23 \sim 26$ and $warp\_exp \approx 40 \sim 50$ on GTX580, for different benchmarks. When $\#warp \leq 24$ (case 3), warps are too few to hide memory latencies, and the global memory bandwidth is sufficient to support all these warps, so the performance increases with more warps. When $\#warp \geq 28$ (case 2), warps are still too few to hide memory latencies, but on the other hand, warps are so many that the global memory bandwidth cannot support all the memory warps concurrently, so the performance is limited by the memory bandwidth and decreases with more warps. Consequently, $\#warp = 24$ is the best choice on NVIDIA GTX580.

In sparse LU factorization, the computation-to-memory ratio is very low (there are only $2 \sim 5$ computational instructions between two global memory instructions), leading to a low $comp\_cycle$ (< 10) and a high $warp\_exp$ ($40 \sim 50$, for GTX580). Case 1 is the best case in GPU computing, but it does not exist in the proposed LU factorization approach performed on our GPUs, since $warp\_exp$ is larger than the maximum number of resident warps per SM.

It is necessary to explain why the bandwidth reported in Table 3 does not reach the peak bandwidth of the GPU, but the performance is still constrained by the global memory bandwidth. The bandwidth reported in Table 3 only considers the necessary data sizes. We can regard this bandwidth as "effective bandwidth". However, GPUs work in a different way. NVIDIA GPUs access to the global memory via aligned 32-, 64- or 128-byte memory transactions [32]. If a warp only needs an 1-byte data, it still requests a 32-byte transaction. In this case, the unused 31 bytes are unnecessary but they still occupy the memory bandwidth. Consequently, the actual runtime memory bandwidth which constrains the performance, is larger than the reported effective bandwidth. If the peak memory bandwidth can be larger, we still have a great potential to further improve the performance, since currently we do not fully utilize the compute capability of GPUs.

7 Conclusions

This paper presents a GPU-based sparse direct solver intended for circuit simulation problems, which is based on parallel LU numeric factorization without pivoting. We have presented the GPU-based LU factorization approach in detail and analyzed the performance. A parametric performance model is built to investigate the bottleneck of the

proposed approach and choose the optimal number of work threads on the given hardware platform.

The proposed LU factorization approach on NVIDIA GTX580 achieves on average 7.02× (geometric mean) speedup compared with sequential PARDISO, and 1.55× speedup compared with 16-threaded PARDISO. The performance of the proposed GPU solver is comparable to that of the existing GPU solvers listed in Table 1. The model-based analysis indicates that the performance of the GPU-based LU factorization is constrained by the global memory bandwidth. Consequently, if the global memory bandwidth becomes larger, the performance can be further improved.

One limitation of the current approach is that it can be only implemented on single-GPU platforms. For multiple-GPU platforms, blocked algorithms are required, and the data transfers between GPUs need to be carefully controlled. In addition, multiple GPUs can provide larger global memory bandwidth so the performance is expected to be higher.

Acknowledgments

This work was supported by 973 project (2013CB329000), National Science and Technology Major Project (2013ZX03003013-003) and National Natural Science Foundation of China (No. 61373026, No. 61261160501), and Tsinghua University Initiative Scientific Research Program. This work was done when Ling Ren was at Tsinghua University. We would like to thank Prof. Haohuan Fu and Yingqiao Wang from Center for Earth System Science, Tsinghua University for lending us the NVIDIA Tesla K20x platform. We also thank the NVIDIA Corporation for donating us GPUs.

References

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

[2] S. Rajamanickam, E. Boman, and M. Heroux, "ShyLU: A hybrid-hybrid solver for multicore platforms," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, 2012, pp. 631–643.

[3] X. Chen, Y. Wang, and H. Yang, "An adaptive LU factorization algorithm for parallel circuit simulation," in *Proc. 17th Asia and South Pacific Design Autom. Conf.*, Jan. 30, 2012–Feb. 2, 2012, pp. 359–364.

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

[5] X. Chen, Y. Wang, and H. Yang, "NICSLU: An adaptive sparse matrix solver for parallel circuit simulation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 2, pp. 261–274, Feb. 2013.

[6] N. Kapre and A. DeHon, "Performance comparison of single-precision SPICE model-evaluation on FPGA, GPU, Cell, and multi-core processors," in *Proc. Int. Conf. Field Programmable Logic and Appl.*, Aug. 31, 2009–Sep. 2, 2009, pp. 65–72.

[7] K. Gulati, J. Croix, S. Khatri, and R. Shastry, "Fast circuit simulation on graphics processing units," in *Proc. Asia and South Pacific Design Autom. Conf.*, Jan. 2009, pp. 403–408.

[8] R. Poore, "GPU-accelerated time-domain circuit simulation," in *Proc. IEEE Custom Integr. Circuits Conf.*, Sep. 2009, pp. 629–632.

[9] J. W. H. Liu, "The multifrontal method for sparse matrix solution: Theory and practice," *SIAM Rev.*, vol. 34, no. 1, pp. 82–109, 1992.

[10] M. Christen, O. Schenk, and H. Burkhart, "General-purpose sparse matrix building blocks using the NVIDIA CUDA technology platform," in *Proc. First Workshop General Purpose Process. Graph. Process. Units*, 2007, pp. 1–8 .

[11] G. P. Krawezik and G. Poole, "Accelerating the ANSYS direct sparse solver with GPUs," in *Proc. Symp. Appl. Accelerators High Perform. Comput.*, July 2009, pp. 1–3.

[12] C. D. Yu, W. Wang, and D. Pierce, "A CPU-GPU hybrid approach for the unsymmetric multifrontal method," *Parallel Comput.*, vol. 37, no. 12, pp. 759–770, Dec. 2011.

[13] T. George, V. Saxena, A. Gupta, A. Singh, and A. Choudhury, "Multifrontal Factorization of Sparse SPD Matrices on GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 372–383.

[14] R. F. Lucas, G. Wagenbreth, J. J. Tran, and D. M. Davis, "Multifrontal sparse matrix factorization on graphics processing units," Los Angeles, CA, USA, Tech. Rep. ISI-TR-677, 2012.

[15] R.F. Lucas, G. Wagenbreth, D. M. Davis, and R. Grimes, "Multifrontal computations on GPUs and their multi-core hosts," in *Proc. 9th Int. Conf. High Perform. Comput. Comput. Sci.*, 2011, pp. 71–82.

[16] 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, Sep. 2010.

[17] L. Ren, X. Chen, Y. Wang, C. Zhang, and H. Yang, "Sparse LU factorization for parallel circuit simulation on GPU," in *Proc. 49th ACM/EDAC/IEEE Design Autom. Conf.*, Jun. 2012, pp. 1125–1130.

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

[19] J. R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," *SIAM J. Sci. Statist. Comput.*, vol. 9, no. 5, pp. 862–874, 1988.

[20] J. J. Dongarra, J. D. Cruz, S. Hammerling, and I. S. Duff, "Algorithm 679: A set of level 3 basic linear algebra subprograms: Model implementation and test programs," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 18–28, Mar. 1990.

[21] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with PARDISO," *Future Generat. Comput. Syst.*, vol. 20, no. 3, pp. 475–487, Apr. 2004.

[22] G. Poole, Y. Liu, Y. cheng Liu, and J. Mandel, "Advancing analysis capabilities in ansys through solver technology," in *Proc. Electron. Trans. Numer. Anal.*, Jan. 2003, pp. 106–121.

[23] T. A. Davis, "Algorithm 832: Umfpack v4.3—An unsymmetric-pattern multifrontal method," *ACM Trans. Math. Softw.*, vol. 30, no. 2, pp. 196–199, Jun. 2004.

[24] A. Gupta, M. Joshi, and V. Kumar, "WSMP: A high-performance shared- and distributed-memory parallel sparse linear solver," IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, Tech. Rep. RC 22038 (98932), Apr. 2001.

[25] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2008, pp. 1–11.

[26] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *Proc. IEEE Int. Symp. Parallel Distrib. Process., Workshops Phd Forum*, Apr. 2010, pp. 1–8.

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

[28] J. Kurzak, P. Luszczek, M. Faverge, and J. Dongarra, "LU factorization with partial pivoting for a multicore system with accelerators," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 8, pp. 1613–1621, Aug. 2013.

[29] The Univ. Tennessee, Knoxville, TN, USA. *The MAGMA Project*. (2013). [Online]. Available: http://icl.cs.utk.edu/magma/index.html

[30] NVIDIA Corporation. (2012). *CUDA BLAS* [Online]. Available: http://docs.nvidia.com/cuda/cublas/

[31] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate," *ACM Trans. Math. Softw.*, vol. 35, no. 3, pp. 22:1–22:14, Oct. 2008.

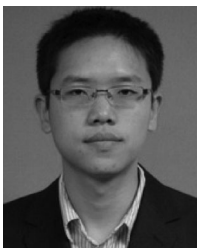
[32] NVIDIA Corporation. *NVIDIA CUDA C Programming guide*. (2012). [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[33] I. S. Duff and J. Koster, "The design and use of algorithms for permuting large entries to the diagonal of sparse matrices," *SIAM J. Matrix Anal. Appl.*, vol. 20, no. 4, pp. 889–901, Jul. 1999.

[34] I. S. Duff and J. Koster, "On algorithms for permuting large entries to the diagonal of a sparse matrix," *SIAM J. Matrix Anal. Appl.*, vol. 22, no. 4, pp. 973–996, Jul. 2000.

[35] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebric decision diagrams and their applications," *Formal Methods Syst. Design*, vol. 10, no. 2/3, pp. 171–206, 1997.
[36] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.
[37] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 152–163.

Xiaoming Chen (S'12) received the BS degree from the Department of Electronic Engineering, Tsinghua University, Beijing, China, in 2009, where he is currently working toward the PhD degree. His current research interests include power and reliability-aware circuit design methodologies, parallel circuit simulation, high-performance computing on GPUs, and GPU architecture. He was nominated for the Best Paper Award in ISLPED 2009 and ASPDAC 2012. He is a student member of the IEEE.



Ling Ren received the BS degree from the Department of Electronic Engineering, Tsinghua University, Beijing, China, in 2012. He is currently working toward the PhD degree in the Department of Electrical Engineering and Computer Science at Massachusetts Institute of Technology. His research interests include computer architecture, computer security, and parallel computing.

Yu Wang (S'05-M'07) received the BS degree and the PhD degree (with honor) from Tsinghua University, Beijing, China in 2002 and 2007, respectively. He is currently an associate professor with the Department of Electronic Engineering, Tsinghua University. His current research interests include parallel circuit analysis, low-power and reliability-aware system design methodology, and application-specific hardware computing, especially for brain-related topics. He has authored and co-authored more than 90 papers in refereed journals and conference proceedings. He received the Best Paper Award in ISVLSI 2012. He was nominated five times for the Best Paper Award (ISLPED 2009, CODES 2009, twice in ASPDAC 2010, and ASPDAC 2012). He is the Technical Program Committee co-chair of ICFPT 2011, the Publicity co-chair of ISLPED 2011, and the finance chair of ISLPED 2013. He is a member of the IEEE.

Huazhong Yang (M'97-SM'00) received the BS degree in microelectronics, and the MS and PhD degrees in electronic engineering from Tsinghua University, Beijing, China, in 1989, 1993, and 1998, respectively. In 1993, he joined the Department of Electronic Engineering, Tsinghua University, where he is currently a specially appointed professor of the Cheung Kong Scholars Program. He has authored and co-authored more than 200 technical papers and holds 70 granted patents. His current research interests include wireless sensor networks, data converters, parallel circuit simulation algorithms, nonvolatile processors, and energy-harvesting circuits. He is a senior member of the IEEE.

GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling (Supplementary Material)

Xiaoming Chen, *Student Member, IEEE*, Ling Ren, Yu Wang, *Member, IEEE*, and Huazhong Yang, *Senior Member, IEEE*

APPENDIX A
IMPLEMENTATION DETAILS AND OPTIMIZATION STRATEGIES

A.1 Ensuring Timing Order on GPUs

Algorithm 1 shows the pipeline mode parallel left-looking algorithm. In this mode, appropriate timing order between columns must be guaranteed. Line 6 shows that a column must wait for its dependent columns to finish. If column k depends on column t, only after column t is finished, can column k be updated by column t. We use Fig. 1 as an example to explain it. Columns 8∼10 are being processed simultaneously, and other columns are finished. Column 9 can be first updated by columns 4, 6 and 7. But currently column 9 cannot be updated by column 8 since column 8 is not finished. Column 9 must wait for column 8 to finish. Similar situation is for column 10.

We set a flag which ia marked as volatile for each column to indicate whether the column is finished (`false` for unfinished and `true` for finished). Line 6 is executed by a `while` loop, and in the loop, the flag corresponding to the dependent column is checked. If the flag changes to `true`, the loop exits. Another work [1] uses a similar method to deal with the timing order in the case of one-way dependence.

Ensuring timing order on GPUs deserves special attention. The number of warps in a GPU kernel must be carefully controlled. This issue is with respect to the concept of resident warp (or active warp) [2] on GPUs. Unlike the context switch mechanism of CPUs, GPUs have large register files to hold lots of warps, and warps which can be switched for execution are resident in registers to ensure a zero-overhead context switch. Consequently, there is a maximum number of resident warps that the register file can handle. In addition to the register file, the shared memory usage can also restrict the maximum number of resident warps (our algorithm does not use any shared memory). If the number of assigned warps exceeds the maximum number of resident warps, some warps will be not resident at the beginning, and they have to wait for other resident warps to finish execution and then become resident.

In pipeline mode, we have to ensure all the warps to be resident from the beginning. If a column is computed by a non-resident warp, columns depending on it have to wait for this column to finish. But in turn, the non-resident warp has no chance to become resident because

Algorithm 1 Pipeline mode algorithm.

```
1: for each warp in parallel do
2:   while there are still unfinished columns do
3:     Get a new unfinished column, say column i;
4:     Put A(:,i) into an intermediate vector x;
5:     for j = 1 : i − 1 where U(j,i) ≠ 0 do
6:       Wait until column j is finished;
7:       x(j+1:n) −= x(j) × L(j+1:n, j);
8:     end for
9:     U(1:i, i) = x(1:i);
10:    L(i:n, i) = x(i:n)/x(i);
11:    Mark column i as finished;
12:  end while
13: end for
```

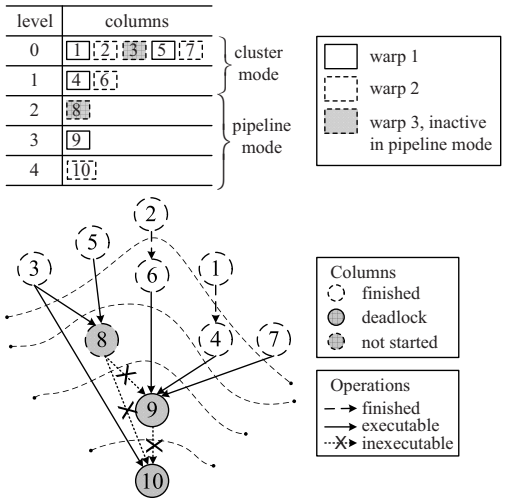


Fig. 1: Illustration of deadlock caused by non-resident warps.

no resident warp can finish. This results in a deadlock. Fig. 1 is an illustration of a deadlock. Suppose we have invoked 3 warps on a GPU that supports only 2 resident warps. There is no problem in cluster mode, since warps 1 and 2 will eventually finish execution so warp 3 can start. But in pipeline mode, columns 9 and 10 depend on column 8, which is allocated to the non-resident warp 3, so the resident warps (warps 1 and 2) fall in deadlocks, waiting for column 8 forever. This in turn leaves no chance for warp 3 to become resident.

Therefore the maximum number of columns that can be factorized concurrently in pipeline mode is exactly the maximum number of resident warps of the kernel. In sparse LU factorization, this number depends on the register usage, which can be obtained from the compiler or some profiling tools. The number of resident warps greatly influences the performance of the solver. We have also found that processing too many columns concurrently is undesirable. Our experiments in Section 5 of the main paper will confirm this point, and a detailed explanation is presented in Section 6 of the main paper.

A.2 Optimization of Memory Access Patterns

Optimization of sparse LU factorization on GPUs is mainly about memory optimization. In this subsection, we discuss the data format for intermediate vectors, and the sorting process for more coalesced access patterns to the global memory.

A.2.1 Format of Intermediate Vectors

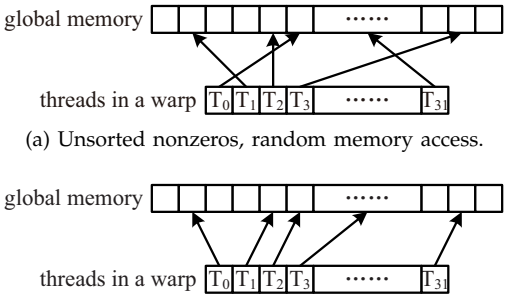
We have two alternative data formats for the intermediate vectors (x in Algorithm 1): compressed sparse column (CSC) vectors or uncompressed vectors. CSC vectors save space and can be placed in the shared memory, while uncompressed vectors have to reside in the global memory. Uncompressed vectors are preferred in this problem for two reasons. First, CSC format is inconvenient for indexed accesses. We have to use binary search, which is very time-consuming even in the shared memory. Moreover, using too much shared memory will reduce the maximum number of resident warps, which results in severe performance degradation:

$$\text{resident warps per SM} \leq \frac{\text{size of shared memory per SM}}{\text{size of a CSC vector}}$$

Our tests have proved that using uncompressed formats is several times faster than using compressed formats.

A.2.2 Improving Data Locality

Higher global memory bandwidth can be achieved on GPUs if memory access patterns are coalesced [2]. The nonzeros in L and U are out of order after pre-processing and the first factorization performed on the host CPU, resulting in highly random memory access patterns. We sort the nonzeros in each column of L and U by their row indices to improve the data locality. As shown in Fig. 2, after sorting, neighboring nonzeros in each column are more likely to be processed by consecutive threads.



(a) Unsorted nonzeros, random memory access.

(b) Sorted nonzeros, more coalesced memory access.

Fig. 2: More coalesced memory access patterns after sorting the nonzeros.

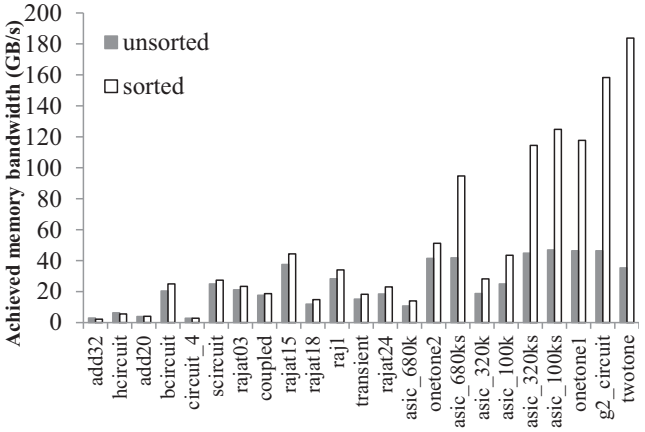


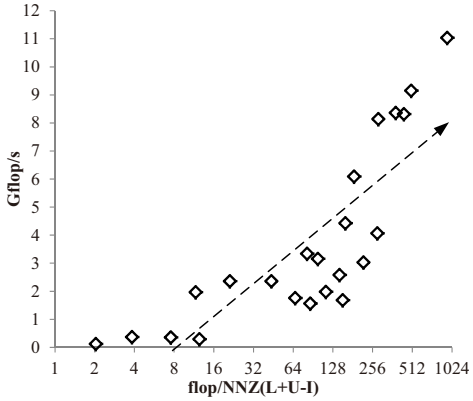
Fig. 3: Performance increases after sorting the nonzeros.

In Fig. 3, we use the test benchmarks to show the effect of sorting. The achieved memory bandwidth is significantly increased by an average of 2.1×. It is worth mentioning that CPU-based sparse LU factorization also benefits from sorting the nonzeros, but the performance gain is tiny (less than 5% on average). Sorting the nonzeros is done by transposing the CSC storages of L and U twice. The time cost of sorting is about 20% on average of the time cost of one LU re-factorization on GPUs. In addition, sorting is performed just once so its time cost is not the bottleneck and negligible in circuit simulation.

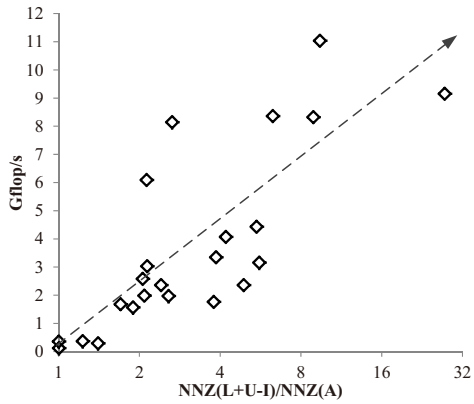
APPENDIX B
ADDITIONAL RESULTS

B.1 Relation Between the Achieved GPU Performance and Matrix Characteristics

We find that the achieved GPU performance strongly depends on the characteristics of the matrices, as shown in Fig. 4. The two subfigures are scatter plots, in which each point represents a matrix. The results are obtained on GTX580. They show that the achieved GPU performance has an approximate logarithmic relation with the relative fill-in ($\frac{NNZ(L+U-I)}{NNZ(A)}$) or the average number of flop per nonzero ($\frac{flop}{NNZ(L+U-I)}$). The fitted lines are also plotted in Fig. 4.



(a) Relation between the achieved GPU performance and the average number of flop per nonzero.



(b) Relation between the achieved GPU performance and the relative fill-in.

Fig. 4: Relation between the achieved GPU performance and matrix characteristics, obtained on NVIDIA GTX580.

B.2 Comparison with GMRES in CUSP

The GPU solver is compared with a preconditioned generalized minimal residual (GMRES) [3] iterative solver in the CUSP library 0.3.1 [4]. The GMRES solver uses an approximate inversion preconditioner named `nonsym_bridson_ainv`, which is based on an incomplete factorization and a dropping strategy proposed in [5]. The results shown in Table 1 are obtained on K20x. The results indicate that for circuit matrices, the GMRES iterative solver is not competitive compared with our direct solver. We have also tried other preconditioners provided by CUSP but they do not help improve the performance of GMRES. Iterative solvers need good preconditioners to ensure convergence, but circuit matrices are asymmetric and irregular so are hard to precondition. In addition, in circuit simulation, the matrix entries change during iterations, so preconditioner is required in each iteration, which significantly increases the computational time of iterative solvers. As shown in Table 1, the preconditioner costs most of the computational time of the iterative solver. Regardless of the preconditioner, time cost of the iterative solving step is also larger than that of the direct solver.

TABLE 1: Comparison with the preconditioned GMRES iterative solver in the CUSP library, obtained on K20x.

	Our approach GPU factoriz.+CPU solve				GMRES solver in CUSP GPU precond.+GPU solve			
	transf.	factoriz.	solve	total	transf.	precond.	solve	total
add32	0.000	0.001	0.000	0.001	0.000	0.220	0.021	0.241
hcircuit	0.003	0.007	0.003	0.012	0.001	2.075	0.193	2.269
add20	0.000	0.000	0.000	0.001	0.000	0.377	0.189	0.566
bcircuit	0.005	0.003	0.003	0.010	0.001	1.042	36.930	37.972
circuit_4	0.002	0.029	0.002	0.033	0.001	0.386	40.024	40.411
scircuit	0.011	0.017	0.008	0.035	0.001	3.561	0.225	3.787
rajat03	0.001	0.003	0.000	0.005	0.000	0.091	0.097	0.188
coupled	0.002	0.017	0.001	0.020	0.000	1.973	0.167	2.140
rajat15	0.008	0.037	0.003	0.048	0.000	9.281	0.159	9.440
rajat18	0.005	0.072	0.003	0.080	0.001	42.358	0.293	42.652
raj1	0.033	0.268	0.019	0.320	0.002	1015.778	0.561	1016.341
transient	0.009	0.168	0.006	0.183	0.002	2340.344	0.487	2340.832
rajat24	0.018	0.336	0.013	0.367	0.003	2740.974	0.788	2741.764
asic_680k	0.029	0.985	0.024	1.038	fail			
onetone2	0.006	0.049	0.002	0.057	0.000	0.657	0.157	0.814
asic_680ks	0.021	0.116	0.019	0.156	0.004	3.716	0.027	3.746
asic_320k	0.025	0.589	0.019	0.634	fail			
asic_100k	0.019	0.342	0.009	0.369	0.001	1057.375	0.382	1057.758
asic_320ks	0.022	0.147	0.017	0.185	0.002	2.304	0.012	2.318
asic_100ks	0.017	0.141	0.008	0.165	0.001	0.766	0.079	0.847
onetone1	0.014	0.146	0.005	0.165	0.000	1.991	0.174	2.165
g2_circuit	0.085	0.805	0.035	0.925	0.001	1.288	16.053	17.343
twotone	0.052	0.777	0.018	0.846	0.001	16.977	0.218	17.197

B.3 Results of Right-Hand-Solving

Though the time cost of right-hand-solving is fairly small, it will affect the scalability of the solver if the LU factorization time is significantly reduced. The parallel strategy of right-hand-solving can be very similar to the parallel re-factorization algorithm proposed in [6]. NVIDIA researches have also proposed a similar parallel triangular solving strategy [7], which is included in the cuSPARSE library [8] (cuSPARSE is an integrated library in CUDA). The comparison between cuSPARSE (version 5.0, implemented on NVIDIA K20x) and CPU-based sequential right-hand-solving is shown in Fig. 5 (the reported GPU time excludes the analysis time). The host CPU is on average 3.16× faster than the GPU for the right-hand-solving step. The right-hand-solving step has much fewer flop than numeric factorization, leading to a high communication-to-memory ratio, especially for circuit matrices. Consequently, GPU-based right-hand-solving for circuit matrices is not expected to obtain high speedups. In addition, a recent research shows that the maximum speedup of parallel right-hand-solving on multicore CPUs is less than 2× regardless of the number of threads [9]. Our hybrid solver uses sequential right-hand-solving executed on CPUs.

APPENDIX C
SUPPLEMENTARY FOR THE PERFORMANCE MODEL

C.1 Measurement of the Cache Miss Rate

We have found that for GTX580 and K20x, disabling the L1 cache just results in less than 5% performance loss, so the effect of the L1 cache is fairly small. Consequently,

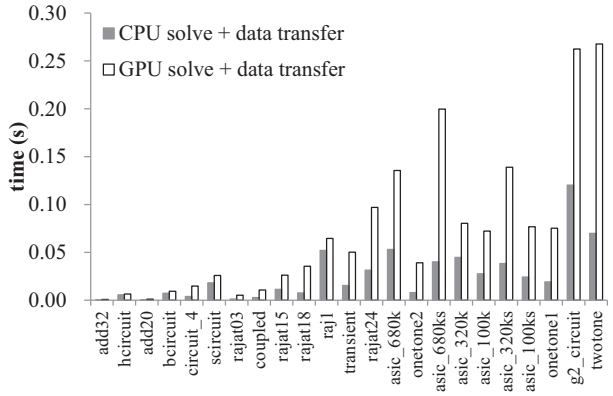


Fig. 5: Comparison of the right-hand-solving time. The CPU code is sequential. The GPU time excludes the analysis time.

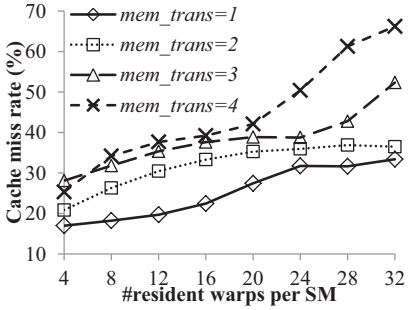


Fig. 6: Measured L2 cache miss rate, obtained on GTX580 using NVIDIA Visual Profiler.

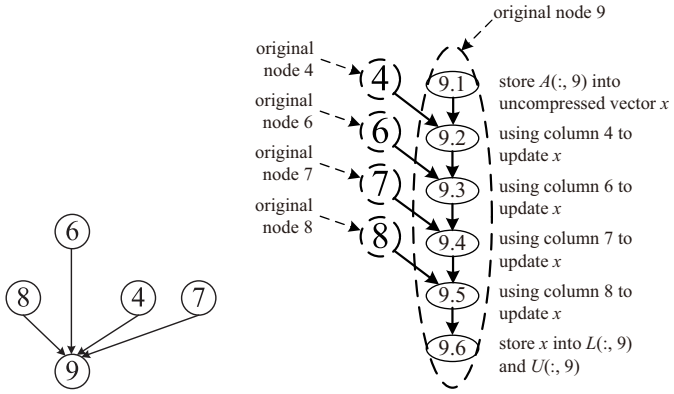
we ignore the L1 cache in the model. In our model, "cache" refers to the L2 cache. The miss rate of the L2 cache $miss\_rate$ is difficult to be analytically modeled, since the cache behavior is very complex and affected by many low-level factors. But we find that $miss\_rate$ has a strong relation with the number of transactions of each memory period (mem_trans). If the memory access patterns are more coalesced, fewer memory transactions are generated, the cache hit ratio is higher. In addition, if there are more resident warps on one SM, the cache hit ratio will decrease since more warps are sharing the same cache. A micro-benchmark designed for this study is used to measure the cache miss rate. The code of the micro-benchmark is shown here:

```
__global__ void CacheMissRateMeasurement
        (double *a, double *b, int stride)
{
    int tid = threadIdx.x
    + blockIdx.x*blockDim.x;
    double x = a[tid];
    for (int i=tid; i<N; i+=32)
    {
        a[(i*stride)%N] -= x * b[i];
    }
}
```

Basically, this micro-benchmark emulates the core operation (multiplication-and-add) in sparse LU factorization. The parameter `stride` is used to control the number of memory transactions. The number of resident warps is controlled by the invoked threads of this micro-benchmark. The measured results on GTX580 are shown in Fig. 6, using NVIDIA Visual Profiler.

C.2 Directed Acyclic Graph (DAG) Analysis

As mentioned in Section 3.2 of the main paper, a dependence graph (DAG) is used to describe the dependence between columns. In this section, the DAG is expanded to describe the detailed dependence in pipeline mode.



(a) Original dependence graph of node 9. (b) Expanded dependence graph of node 9.

Fig. 7: Illustration of DAG analysis.

The operations for factorizing each column is decomposed so each original node is also decomposed into several subnodes. We use node 9 in Fig. 1 to illustrate how the DAG is expanded, as shown in Fig. 7. All the subnodes are labeled, and their corresponding operations are shown in Fig. 7b. All the subnodes belonging to one original node are executed by a same warp. Each solid arrow in Fig. 7b denotes data dependence or computation order. For example, subnodes 9.1 to 9.6 are executed in sequence by a same warp, and subnode 9.2 can start only after node 4 and subnode 9.1 are both finished.

The number of execution cycles of each subnode is calculated by the performance model. Still take Fig. 7b as an example, when the start time of node 9 and finish time of nodes 4, 6, 7 and 8 are known, it is quite simple to calculate the earliest finish time of node 9 (i.e. the earliest finish time of subnode 9.6) by a well-known critical path analysis. The expanded DAG avoids calculating the dynamic waiting time in the pipeline mode (shown in Algorithm 1).

To calculate the earliest finish time of the whole DAG computed by $\#warp \times \#SM$ warps, a task queue is maintained for each warp. All the nodes are visited in a topological order. When a node (say node k) is being visited, the warp that first finishes its last task (say warp t) is selected to compute node k, so node k is put into

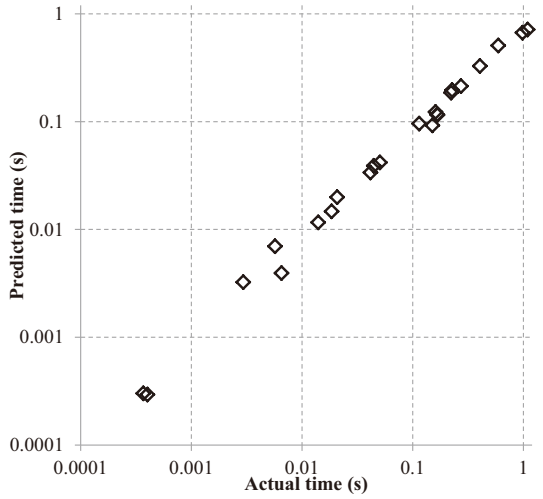


Fig. 8: Predicted time vs. actual time, each point represents a matrix, obtained on GTX580.

warp t's task queue and the finish time of node k can be determined. The earliest finish time of the whole DAG is just the maximum finish time of all the nodes.

C.3 Validation of the Performance Model

The comparison between the predicted LU factorization time on GTX580 and the actual runtime is shown in Fig. 8. The proposed performance model can obtain reasonable results compared with the real results obtained on NVIDIA GTX580. The average relative error between the predicted runtime and the actual runtime is 21.8%.

REFERENCES

[1] S. Yan, G. Long, and Y. Zhang, "StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, 2013, pp. 229–238.

[2] NVIDIA Corporation, "NVIDIA CUDA C programming guide." [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[3] Y. Saad and M. H. Schultz, "GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, vol. 7, no. 3, pp. 856–869, Jul. 1986.

[4] "CUSP." [Online]. Available: https://code.google.com/p/cusp-library/

[5] C.-J. Lin and J. J. More, "Incomplete Cholesky Factorizations With Limited Memory," *SIAM J. SCI. COMPUT*, vol. 21, no. 1, pp. 24–45, 1999.

[6] 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.

[7] M. Naumov, "Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU," NVIDIA Technical Report, NVR-2011-001, Tech. Rep., june 2011.

[8] NVIDIA Corporation, "cuSPARSE." [Online]. Available: https://developer.nvidia.com/cusparse

[9] X. Xiong and J. Wang, "Parallel forward and back substitution for efficient power grid simulation," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, nov. 2012, pp. 660–663.