Sparse LU Factorization for Parallel Circuit Simulation on GPU

Ling Ren, Xiaoming Chen, Yu Wang, Chenxi Zhang, Huazhong Yang
Department of Electronic Engineering
Tsinghua National Laboratory for Information Science and Technology
Tsinghua University, Beijing, China
{rl08@mails, chenxm05@mails, yu-wang@mail}.tsinghua.edu.cn

ABSTRACT

Sparse solver has become the bottleneck of SPICE simulators. There has been few work on GPU-based sparse solver because of the high data-dependency. The strong data-dependency determines that parallel sparse LU factorization runs efficiently on shared-memory computing devices. But the number of CPU cores sharing the same memory is often limited. The state of the art Graphic Processing Units (GPU) naturally have numerous cores sharing the device memory, and provide a possible solution to the problem. In this paper, we propose a GPU-based sparse LU solver for circuit simulation. We optimize the work partitioning, the number of active thread groups, and the memory access pattern, based on GPU architecture. On matrices whose factorization involves many floating-point operations, our GPU-based sparse LU factorization achieves 7.90× speedup over 1-core CPU and 1.49× speedup over 8-core CPU. We also analyze the scalability of parallel sparse LU factorization and investigate the specifications on CPUs and GPUs that most influence the performance.

Categories and Subject Descriptors

J.6 [**Computer-Aided Engineering**]: Computer-aided design (CAD)

General Terms

Performance

Keywords

GPU, Parallel Sparse LU Factorization, Circuit Simulation

1. INTRODUCTION

SPICE (Simulation Program with Integrated Circuit Emphasis) [1] is the most frequently used circuit simulator today. However, the rapid development of VLSI (Very Large Scale Integration) presents great challenges to SPICE simulators' performance. In modern VLSI, the dimension of circuit matrices after post-layout extraction can easily reach several million. It may take SPICE simulators days or even weeks to perform post-layout simulation on modern CPUs. The two most time-consuming steps in SPICE simulation are the sparse matrix solver by LU factorization and the model evaluation. These two steps have to be performed iteratively for many times.

In the last decade, Graphic Processing Units (GPU) prove useful in many fields [2]. There have been works on GPU-based model evaluation [3, 4]. The parallelization of model evaluation in SPICE simulation is straightforward since it consists of a large amount of independent tasks, i.e. model evaluation for each component in the circuit. However, it is difficult to parallelize the sparse solver because of the high data-dependency during the numeric LU factorization and the irregular structure of circuit matrices. To our knowledge, there has been no work on GPU-based sparse solver for circuit simulation.

Previous works on GPU LU factorization mainly focus on dense matrices [5–7]. Though the performance of GPU-based dense LU factorization is very promising, up to 388 Gflop/s (Giga FLoating point OPerations per second) on GTX 280 [6], a simple calculation shows that sparse matrices should not be factorized as dense matrices. Take *onetone2* (36k by 36k) as an example. Even if we calculate the performance of dense factorization as 1000 Gflop/s, it would still cost 15.5 second to factorize *onetone2* as a dense matrix, while a straightforward sequential sparse factorization on a single core usually costs less than 1 second.

Up till now, sparse LU solvers are mainly implemented on CPUs. SuperLU [8–10] incorporates *Supernode* in Gilbert / Peierls (G/P) left-looking algorithm [11], enhancing the computing capability with dense blocks. PARDISO [12] also adopts *Supernode*. Christen et al. mapped PARDISO to GPU [13]. Their work still follows the idea of GPU-based dense LU factorization. They compute dense blocks on GPU and the rest of the work is still done on CPU.

However, it is hard to form supernodes in extremely sparse matrices such as circuit matrices. This feature makes *Supernode*-based algorithms less efficient than column-based algorithms for circuit simulation. So in KLU [14], which is specially optimized for circuit simulation, G/P left-looking algorithm [11] is adopted directly without *Supernode*.

KLU only has the sequential version. Due to the high data-dependency during the numerical factorization, parallel G/P left-looking algorithm is efficient only on shared-memory computing platforms, such as FPGA (Field Programmable Gate Array), multi-core CPU and GPU. Several studies developed G/P left-looking algorithm on FPGA [15–17], but the scalability to large-scale circuits is limited by FPGA on-chip resources. Chen et al. parallelized the algorithm on multi-core CPU [18]. Their implementation scales well with the number of CPU cores, but the number of CPU cores sharing the same memory is often limited. Most commodity CPUs (e.g., Intel Xeon, AMD Phenom) have no more than 6 cores. The state of the art GPUs provide a possible solution to the above problems. Compared with CPUs, GPUs have far more cores sharing the same memory (e.g. GTX580 GPU has 512 CUDA cores), and higher global memory bandwidth. Therefore, in this work, we for the first time present a parallel sparse LU solver (without pivoting) for circuit simulation on GPU. Our contributions are

- **exposing more parallelism for many-core architecture.** We must expose enough parallelism to make the sparse LU solver efficient on GPU (many-core architecture). Two kinds of parallelism are proposed in [18] to describe the

DAC 2012, June 3-7, 2012, San Francisco, California, USA.

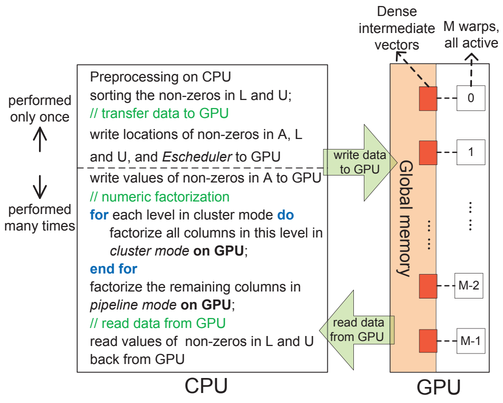

Dense intermediate vectors
M warps, all active
performed only once
performed many times
Preprocessing on CPU
sorting the non-zeros in L and U;
// transfer data to GPU
write locations of non-zeros in A, L and U, and Escheduler to GPU
write values of non-zeros in A to GPU
// numeric factorization
for each 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
write data to GPU
read data from GPU
Global memory
0
1
M-2
M-1
CPU
GPU

Figure 1: The workflow of sparse LU factorization on GPU

parallelism between vector operations. This level of parallelism alone is not enough for the thousands of threads running concurrently on GPU. We also utilize the parallelism within the vector operations. To efficiently deal with the two levels of parallelism, we partition the workload based on the features of the two modes and GPU architecture. Our strategy minimizes idle threads, saves synchronization costs, and ensures enough threads in total.

- **ensuring timing order on GPU.** In parallel left-looking algorithm, appropriate timing order must be guaranteed. Ensuring timing order on GPU involves carefully controlling the number of thread groups.
- **optimizing memory access pattern.** (1) We design the suitable data format of the intermediate vectors on GPU; (2) We propose sorting the nonzeros to improve the data locality for more coalesced accesses to global memory.

Experimental results on 36 matrices show that the GPU-based LU solver is efficient with matrices whose factorization involves many (more than 200M with our platforms) flops. On these matrices, the GPU achieves 7.90× speedup over 1-core CPU and 1.49× speedup over 8-core CPU. For matrices with denormal numbers [19], the speedup is even greater. We further analyze the scalability on different CPUs and GPUs and investigate which specifications of the devices have the greatest influence on the performance of sparse LU solver.

The rest of this paper is organized as follows. In Section 2, we introduce our GPU-based sparse LU factorization in detail. Experimental results and discussion are presented in Section 3. Section 4 concludes the paper.

2. SPARSE LU FACTORIZATION ON GPU

In this section, we present our GPU-based sparse LU factorization in detail. Fig. 1 is the workflow. The preprocessing is performed only once on CPU (Section 2.1). Numeric factorization is done on GPU in two parallel modes, where we optimize the workload partitioning based on the different features of the two modes and the GPU architecture (Section 2.2). Then we discuss several important points in our GPU-based sparse LU solver, including timing order between GPU thread groups (Section 2.3) and the optimizations to the memory access pattern (Section 2.4).

2.1 Preprocessing

The preprocessing consists of three operations: (1) HSL_MC64 algorithm [20] to improve numeric stability; (2) AMD (Approximate Minimum Degree) algorithm [21] to reduce fill-ins; (3) G/P algorithm based pre-factorization (a complete numeric factorization with partial pivoting) [11] to calculate the symbolic structure of the LU factors. We denote the matrix after preprocessing as $\mathbf{A}$. In circuit simulation, the nonzero structure of $\mathbf{A}$, $\mathbf{L}$ and $\mathbf{U}$ remains unchanged through the iterations of numeric factorization.

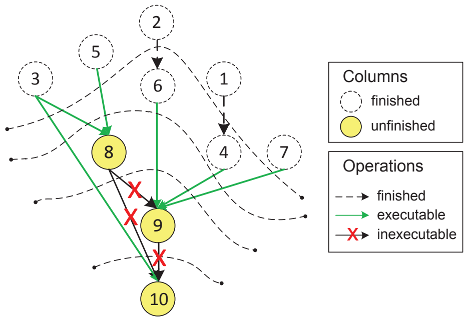


Figure 2: Parallelism and timing order in parallel G/P algorithm

2.2 Exposing More Parallelism

Algorithm 1 Sequential G/P left-looking algorithm

```
$\mathbf{L} = \mathbf{I}$
for $k = 1 : n$ do
    // solving $Lx = b$   $b = A(:, k)$   the kth column of $\mathbf{A}$
    $x = b$;
    for $j = 1 : k - 1$ where $U(j, k) \neq 0$ do
        // Vector MAD
        $x(j + 1 : n) = x(j + 1 : n) - L(j + 1 : n, j) \cdot x(j)$;
    end for
    $U(1 : k, k) = x(1 : k)$;
    $L(k : n, k) = x(k : n)/U(k, k)$;
end for
```

Algorithm 1 is the sequential G/P left-looking algorithm [11]. The core operation in the algorithm is vector multiple-and-add (MAD). Two parallel modes are proposed in [18] to describe the parallelism between vector MAD operations, i.e. the *cluster mode* and the *pipeline mode*. We take Fig. 2 as an example to explain this level of parallelism. Suppose the columns in dashed circles are already finished, and column 8, 9 and 10 are being processed. The MAD operations represented by green solid arrows are executable. Parallelism exists in these operations, though the operations to the same column must be executed in a strict order.

The parallelism between MAD operations alone cannot take full advantage of GPUs' memory bandwidth. We utilize another intrinsic level of parallelism in sparse LU factorization: the parallelism within vector operations. Now we consider how to partition the workload to fully utilize GPU resources. In the process, several factors should be considered.

For convenience, we refer to the threads that process the same column as a virtue group. Threads in a virtue group operate on nonzeros in the same column and must synchronize. Smaller virtue groups can reduce idle threads. But virtue groups should not be too small for two reasons. First, too small virtue groups result in too few threads in total (the number of columns factorized simultaneously, i.e. the number of virtue groups, is limited by the storage space and cannot be very large), which is undesirable in GPU computing. Second, GPUs schedule threads in a SIMD (Single-Instruction-Multiple-Data) manner. A group of SIMD threads are called a *warp* on NVIDIA GPUs or a *wavefront* on AMD GPUs. If threads within a warp diverge, all necessary paths are executed serially. Different virtue groups process different columns and hence often diverge. Thus, too small virtue groups increase divergence between SIMD threads.

Taking all the above factors into consideration, we propose the following work partitioning strategy. In *cluster mode*, columns are very sparse, so while ensuring enough threads in total, we make virtue groups 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 virtue groups matters little in the sense of reducing idle threads. We use one warp as one virtue group. This strategy not only reduces divergence between SIMD threads, but also saves the cost of synchronization, since synchronization between threads within the same warp is automatically guaranteed by GPU's SIMD architecture.

2.3 Ensuring Timing Order on GPU

Algorithm 2 Pipeline mode parallel left-looking algorithm

```
for each work-group in parallel do
    while there are still unfinished columns do
        Get a new column, say column $k$;
        Put the nonzeros of $\mathbf{A}$ into the intermediate vector $x$;
        for all column $j$ that $U(j,k) \neq 0$ do
            Wait until column $j$ is finished;
            $x(j+1:n) = x(j+1:n) - L(j+1:n, j) \cdot x(j)$;
        end for
        $U(1:k,k) = x(1:k)$;
        $L(k:n,k) = x(k:n)/U(k,k)$;
        Mark column $k$ finished;
    end while
end for
```

Algorithm 2 is the *pipeline mode* parallel left-looking algorithm. In this parallel mode, appropriate timing order between columns must be guaranteed. If column k depends on column t, only after column t is finished, can column k be updated by column t. We still use the example in Fig. 2 to explain the required timing order. Suppose column 8, 9 and 10 are being processed, and other columns are finished. Column 9 can be first updated with column 4, 6, 7, corresponding to the solid green arrows. But currently column 9 can not be updated with column 8. It must wait for column 8 to finish. Similar situation for column 10.

Ensuring timing order on GPU deserves special attention. The number of warps in the GPU kernel must be carefully controlled. It has to do with the concept of resident warps on GPU [22]. Resident warps refer to warps that reside on Streaming Multiprocessor (SMs) and are active for execution. A GPU kernel can have many warps. Often, due to the limited resources, some warps are not resident at the beginning. Rather, they have to wait for other resident warps to finish execution and then become resident.

However, in *pipeline mode* of sparse LU factorization, we have to ensure all the warps to be resident from the beginning. If a column is allocated to an non-resident warp, columns depending on it have to wait for this column to finish. But in turn, the non-resident warp would have no chance to become resident because no resident warp can ever finish. This results in a deadlock. Fig. 3 is an illustration of the situation. Suppose we have issued 3 warps on a GPU that supports only 2 resident warps. There is no problem in *cluster mode*, since warp 1 and 2 will eventually finish execution so that warp 3 can start. But in *pipeline mode*, column 9 and column 10 depend on column 8, which is allocated to the non-resident warp 3, so the resident warps (warp 1 and 2) fall in dead loops, 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 simultaneously in *pipeline mode* is exactly the number of resident warps in this kernel. This number depends on factors such as the resource usage and the number of branches or loops [22], and greatly influences the performance of our GPU-based sparse LU solver. But processing too many columns simultaneously is also undesirable. It is likely that for a certain column, its corresponding warp has nothing to do but to wait, because columns it depends on are unfinished. (In this case, this warp still consumes GPU cores and memory bandwidth.) Our experiments in Section 3.3 will confirm this point.

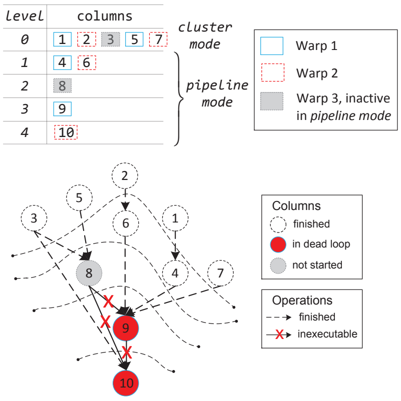
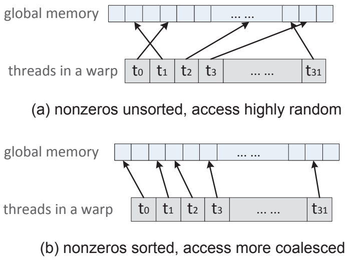


Figure 3: An illustration of deadlocks resulting from non-resident warps



(a) nonzeros unsorted, access highly random

(b) nonzeros sorted, access more coalesced

Figure 4: More coalesced memory accesses after sorting

2.4 Optimization of Memory Access Pattern

Optimization for GPU-based sparse LU factorization is mainly about memory optimization. In this subsection, we discuss the data format for intermediate vectors, and the sorting process for more coalesced accesses to global memory.

Intermediate Vectors' Format. We have two alternative data formats for the intermediate vectors (x in Algorithm 2): CSC (Compressed Sparse Column) sparse vectors and dense arrays. CSC sparse vectors save space and can be placed in shared memory, while dense arrays have to reside in global memory. Dense arrays 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 within shared memory. Moreover, using too much shared memory would reduce the number of resident warps per SM:

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

which results in severe performance degradation.

Improving Data Locality. Higher global memory bandwidth is achieved on GPU if memory accesses are coalesced [22]. But the nonzeros in $\mathbf{L}$ and $\mathbf{U}$ are out of order after preprocessing, which affects the coalesced accesses. We sort the nonzeros in $\mathbf{L}$ and $\mathbf{U}$ by their row indices to improve the data locality. As shown in Fig. 4, after sorting, neighboring nonzeros in each column are more likely to be processed by consecutive threads.

In Fig. 5, we use the 21 matrices in the Group B in Table 2 to show the effectiveness of our sorting process. On average, GPU bandwidth is significantly increased from 37.69 GB/s to 91.17 GB/s (2.4× higher). It's worth mentioning that CPU sparse LU factorization also benefits from sorted nonzeros, but the performance increase is only 1.15×. The sorting overheads are negligible, since sorting is performed only once and the time for sorting is usually less than one factorization. We incorporate the sorting procedure in the preprocessing stage.

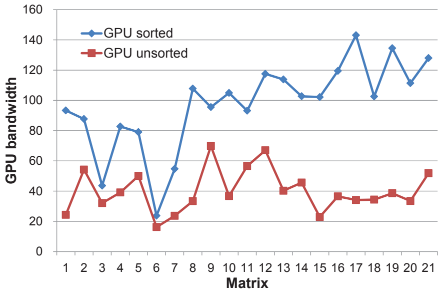


Figure 5: Performance increases by sorting the nonzeros

Table 1: Related specifications of different devices

Devices	Xeon E5405	Xeon X5680	Radeon 5870	GTX580
Peak Bandwidth	−−	−−	153.6 GB/s	192.4 GB/s
Number of Cores	2×4 = 8 cores	2×6 = 12 cores	20 CUs 320 cores[1]	16 SMs[2] 512 cores
Active groups Active threads	−− 8	−− 8	160 10240	512 16384
L1 cache L2 cache L3 cache	32KB/core 12MB/4 cores −−	32KB/core 256KB/core 12MB/6 cores	8KB/CU 512KB/all −−	16KB/SM 768KB/all −−
Clock rate	2.0 GHz	3.2 GHz	850 MHz	772 MHz

[1] CU = Compute Unit. In Radeon 5870, each core contains 5 processing elements (PEs). But PEs are combined for double precision floating point operations [25], so they can be regarded as a single core in our problem.
[2] SM = Stream Multiprocessor

3. EXPERIMENTAL RESULTS AND DISCUSSION

3.1 Experiment Setup

We test the performance of parallel sparse LU factorization on the following four computing platforms: 2 Xeon E5405 CPUs, 2 Xeon X5680 CPUs, AMD Radeon 5870 GPU, and NVIDIA GTX 580 GPU. The experiments on CPU are implemented with C (SSE used) on a 64-bit Linux server. Radeon 5870 is programmed using OpenCL v1.1 [23]. GTX580 is programmed with CUDA 4.0 [22]. The related specifications of all the four devices are listed in Table 1. 36 matrices from University of Florida Sparse Matrix Collection [24] are used to evaluate our GPU sparse LU factorization. Though our intention is for circuit matrices, we also include some matrices from other applications to show that our GPU-based sparse solver is not confined to circuit simulation.

3.2 Performance and Speedup

In Table 2, we present the performance of our GPU-based sparse LU factorization on GTX 580 and compare with KLU and our CPU implementation on Xeon X5680 with different number of cores. We also intended to compare with SuperLU. But the parallel version of SuperLU fails on more than 1/3 of the matrices, and for those successfully factorized matrices, it is also 4.4× slower than our CPU implementation on average. Thus the detailed results of SuperLU are not presented.

The listed time is only for numeric factorization, excluding preprocessing and right-hand solving. Some data have to be transferred between CPU and GPU in every factorization (see Fig. 1). Time for these transfers are included in GPU runtime. We find that GPU bandwidth is strongly related to the Mflops (Mega FLoating-point OPeration) in factorization. So the average bandwidth and speedup in the last row of the table do not convey much useful information.

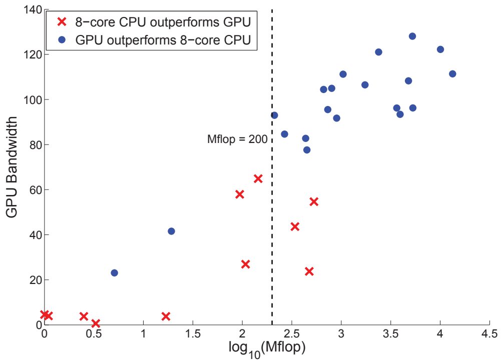


Figure 6: Relation between Mflops and GPU speedups

We categorize our test matrices into three groups. The first two groups are according to the Mflops in their factorization, less than 200M flops in Group A and more than 200M flops in Group B. We show the relation between Mflops and GPU bandwidth for these two groups of matrices in Fig. 6. From the figure, we can see the GPU bandwidth is positively related to Mflops, which indicates that in sparse LU factorization, the high memory bandwidth of GPU can be exploited only when the problem scale is large enough. The low bandwidth for Group A indicates that some overheads (e.g. data transfer, launching kernels) account for most of the runtime. For matrices in Group B, GPU achieves 7.90× speedup over 1-core and 1.49× speedup over 8-core CPUs.

Group C is in some sense special. Many denormal floating point numbers occur when factorizing these matrices. Denormal numbers are used to represent extremely small real numbers. CPUs deal with denormal numbers much slower than with normal represented numbers [19]. This is the major reason why CPU achieves very poor bandwidth on these matrices. In contrast, the state of the art GPUs can handle denormal numbers at the same speed as normal numbers. So GPU speedups for these matrices are very high. Full speed support for denormal numbers is an advantage of GPU in sparse LU factorization and other general purposed computing.

3.3 Scalability Analysis

Table 3: Bandwidth achieved on different devices in GB/s

Devices	Xeon E5405	Xeon X5680	Radeon 5870	GTX 580
Bandwidth Achieved	20.76	61.25	38.18	91.17

The average performance on the four devices are listed in Table 3. The detailed performance on the 21 matrices in Group B are presented in Fig. 7. On different platforms, the factors that restrict the performance are different. The cache size and speed has great influence on the performance of parallel sparse LU factorization on CPU. But cache has little influence on GPU performance. We have tried to declare all the variables on GPU as 'volatile' so that no data are cached. This only results in less than 10% performance loss. The reason is possibly that the cache on GPU is very small so that the cache hit rate is low even in the original kernel. The dominant factors on GPU performance are the peak global memory bandwidth, and whether there are enough active threads to bring out the high bandwidth.

We have mentioned in Section 2.3 that processing too many columns simultaneously may decrease the performance. This phenomenon is not seen on CPUs, but is indeed the case with GTX 580. Presented in Fig. 8 is the achieved bandwidth on 4 matrices on GTX 580 with different number of resident warps. The best performance is attained with about 24 resident warps per SM, rather than with maximum resident warps. This suggests we have fully utilized the *pipeline mode* parallelism on GTX 580 with 24 resident warps per SM. On GTX 580, we achieve 74% peak bandwidth at most (on *twotone*). Considering that the memory

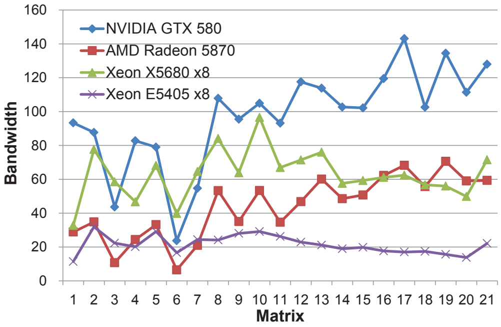


Figure 7: Performance of different devices on the second group of matrices

Table 2: Performance of sparse LU factorization on GTX 580 and Xeon X5680

	Matrix	[1]N (K)	[2]nonzeros (K)	[3]Mflops	1-core CPU time (s)	1-core CPU Bandwidth (GB/s)	GPU time (s)	GPU Bandwidth (GB/s)	GPU speedup over CPU			speedup over KLU
									over 1-core	over 4-core	over 8-core	
A1	hcircuit	103.2	513.1	1.0	0.016	1.76	0.006	4.57	2.60	1.22	0.99	2.45
A2	lung2	109.5	492.6	1.1	0.007	4.44	0.007	4.37	0.99	0.80	0.70	1.80
A3	circuit_4	80.2	307.6	2.5	0.016	4.24	0.018	3.79	0.89	0.38	0.30	0.74
A4	rajat21	402.0	1893.4	3.3	0.051	1.77	0.146	0.63	0.35	0.34	0.28	0.56
A5	bcircuit	67.3	375.6	5.1	0.019	7.27	0.006	23.07	3.17	1.72	1.13	4.24
A6	dc1	116.8	766.4	16.9	0.053	8.74	0.123	3.77	0.43	0.17	0.13	0.51
A7	trans4	116.8	766.4	16.9	0.054	8.56	0.123	3.77	0.44	0.20	0.13	0.54
A8	hvdc2	189.9	1347.3	19.2	0.069	7.63	0.013	41.56	5.45	3.42	2.21	23.35
A9	onetone2	36.1	227.6	94.1	0.217	11.84	0.043	59.71	5.04	1.56	0.86	10.47
A10	transient	178.9	961.8	107.8	0.306	9.64	0.109	26.92	2.79	0.92	0.53	2.63
A11	ckt11752_dc_1	49.7	333.0	144.6	0.305	12.94	0.059	67.30	5.20	1.71	0.98	0.75
A	**Average**					**6.00**		**9.68**	**1.61**	**0.77**	**0.54**	**1.88**
B1	TSOPF_RS_b300_c3	42.1	4413.5	211.1	0.480	12.04	0.062	93.33	7.75	4.99	2.85	1.86
B2	epb3	84.6	463.6	267.2	0.567	12.88	0.083	87.68	6.81	2.12	1.13	6.69
B3	raj1	263.7	1302.5	340.7	0.820	11.36	0.214	43.60	3.84	1.24	0.74	452.71
B4	ASIC_680ks	682.7	2329.2	436.5	1.446	8.26	0.144	82.74	10.02	3.27	1.77	9.70
B5	thermomech_TC	102.2	711.6	449.4	0.950	12.93	0.155	79.03	6.11	2.05	1.16	6.13
B6	ASIC_680k	682.9	3871.8	474.8	1.622	8.00	0.547	23.73	2.96	0.97	0.60	4.23
B7	ASIC_100k	99.3	954.2	529.6	1.253	11.55	0.265	54.68	4.73	1.51	0.85	7.47
B8	ASIC_100ks	99.2	578.9	663.0	1.465	12.37	0.168	107.81	8.71	2.34	1.28	15.22
B9	rma10	48.6	2374.0	730.6	1.513	13.21	0.209	95.52	7.23	2.84	1.50	6.89
B10	onetone1	36.1	341.1	799.8	1.370	15.96	0.208	104.94	6.58	1.94	1.09	59.74
B11	thermomech_dM	204.3	1423.1	898.8	1.914	12.84	0.264	93.14	7.25	2.40	1.39	7.24
B12	venkat50	62.4	1717.8	1043.5	2.220	12.85	0.243	117.52	9.14	3.09	1.65	9.03
B13	Zhao1	33.9	166.5	1737.1	4.088	11.62	0.417	113.80	9.79	2.78	1.50	10.00
B14	thermomech_dK	204.3	2846.2	3637.9	8.348	11.92	0.969	102.69	8.62	3.11	1.78	8.75
B15	crashbasis	160.0	1750.4	3933.0	10.388	10.35	1.052	102.18	9.87	3.00	1.72	9.24
B16	G2_circuit	150.1	726.7	4780.0	12.101	10.80	1.094	119.45	11.06	3.17	1.95	10.74
B17	twotone	120.8	1222.4	5245.0	13.229	10.84	1.002	143.08	13.20	3.70	2.29	66.27
B18	sme3Dc	42.9	3148.7	5291.9	12.821	11.29	1.411	102.56	9.09	2.75	1.81	8.80
B19	xenon1	48.6	1181.1	10066.3	24.802	11.10	2.047	134.47	12.12	3.77	2.40	11.83
B20	helm2d03	392.3	2741.9	13331.8	32.357	11.27	3.273	111.37	9.89	3.47	2.23	9.88
B21	denormal	89.4	1156.2	2387.1	5.676	11.50	0.510	127.94	11.13	3.27	1.79	11.14
B	**Average**					**11.54**		**91.17**	**7.90**	**2.58**	**1.49**	**11.73**
C1	torso2	1033.5	116.0	651.4	4.243	4.20	0.184	97.02	23.11	6.09	3.20	20.13
C2	majorbasis	160.0	1750.4	3933.0	25.490	4.22	1.050	102.37	24.26	7.10	3.81	23.65
C3	ASIC_320k	321.8	2635.4	584.1	25.060	0.64	0.384	41.58	65.25	15.61	8.14	64.12
C4	ASIC_320ks	321.7	1827.8	651.6	28.470	0.63	0.170	104.52	167.02	41.07	21.25	163.28
C	**Average**					**1.63**		**81.06**	**49.72**	**12.90**	**6.77**	**45.78**

[1] the matrix dimension

[2] the number of nonzeros in Matrix **A**

[3] the number of Mega floating point operates

[4] all the average values are geometric average

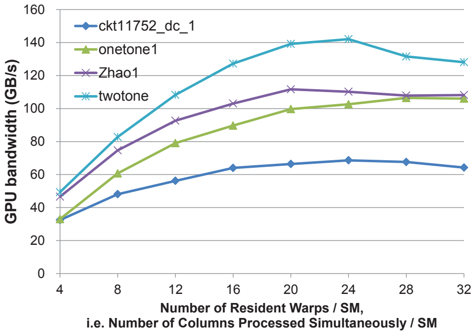


Figure 8: GPU Bandwidth on vs. number of resident warps on GTX 580

accesses are not fully coalesced and that warps sometimes have to wait, this is already close to the peak 192 GB/s. The performance can be improved if GPU peak bandwidth increases in the future.

On Radeon 5870, we achieve 45% peak bandwidth at most (on *xenon1*). A primary reason is that there are too few active wavefronts on Radeon 5870 to fully utilize the global memory bandwidth. On the two CPUs and Radeon 5870 GPU, the bandwidth keeps increasing with the issued threads (wavefronts), as shown in Table 4, which means the performance of our sparse LU solver can be improved if there are more CPU cores sharing the same memory, or Radeon 5870 supports more active wavefronts.

Table 4: Performance on matrices Group B on CPUs and Radeon 5870

Devices	Bandwidth achieved (GB/s)		
	1-core	4-core	8-core
Xeon E5405	4.99	14.41	20.38
Xeon X5680	11.54	35.38	61.22
	2 wavefronts / CU	4 wavefronts / CU	8 wavefronts / CU
Radeon 5870	26.96	39.20	52.59

3.4 Hybrid Sparse LU solver

We have observed that matrices with few flops in factorization are not suitable for GPU acceleration, so we propose a CPU/GPU hybrid sparse solver for circuit simulation. The entire workflow is shown in Fig. 9.

For an input matrix, we first factorize it on CPU with partial pivoting [26] (this is part of the preprocessing). From the preprocessing, we obtain the FLOPs in factorizing the input matrix, and based on this information choose the appropriate platform for numeric factorization. If GPU is chosen, auto-tuning is performed to find the optimal number of warps to be issued. After each iteration, we check the residual between the solutions in two consecutive iterations. If the residual is large, we perform the preprocessing again; otherwise, we enter the next iteration directly. In most cases, the nonzero values do not change rapidly during

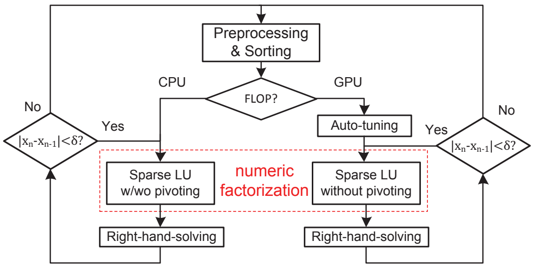


Figure 9: The proposed workflow of CPU/GPU hybrid sparse solver for circuit simulation.

several round of iterations. Thus the pivoting results from the preprocessing can be used in subsequent factorizations to ensure the numerical stability. When nonzeros values vary greatly from previous iterations, we may consider factorizing with partial pivoting on CPU again. In this way the hybrid solver also solves the problem of numerical accuracy.

4. CONCLUSIONS

This is the first work on GPU-based sparse LU factorization intended for circuit simulation. We have presented our GPU sparse solver in detail and analyzed the performance. Our experiments demonstrate that GPU outperforms CPU on matrices with many floating point operations in factorization.

One limitation of our GPU-based sparse LU solver is the inability to handle matrices with too many nonzeros in $\mathbf{L} + \mathbf{U} - \mathbf{I}$, because of the relatively small global memory (1GB) of GTX 580. Yet, current state of the art GPUs such as NVIDIA GTX590 already have 3GB global memory, which is sufficient for factorizing most matrices. With the development of GPU, the scalability to matrices with more nonzeros and fill-ins can be improved.

5. ACKNOWLEDGMENTS

This work was supported by National Science and Technology Major Project (2011ZX01035-001-001-002, 2010ZX01030-001, 2011ZX03003-003-01), National Natural Science Foundation of China (No.60870001, 61028006), Microsoft/AMD and Tsinghua University Initiative Scientific Research Program.

6. REFERENCES

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

[2] *GPGPU'10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. New York, NY, USA: ACM, 2010.

[3] N. Kapre and A. DeHon, "Performance comparison of single-precision spice model-evaluation on FPGA, GPU, Cell, and multi-core processors," in *Field Programmable Logic and Applications. International Conference on*, 2009, pp. 65 –72.

[4] K. Gulati, J. F. Croix, S. P. Khatr, and R. Shastry, "Fast circuit simulation on graphics processing units," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*. IEEE Press, 2009, pp. 403–408.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] X. S. Li and J. W. Demmel, "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Trans. Mathematical Software*, vol. 29, no. 2, pp. 110–140, June 2003.

[11] 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.

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

[14] 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.

[15] J. Johnson, T. Chagnon, P. Vachranukunkiet, P. Nagvajara, and C. Nwankpa, "Sparse LU decomposition using FPGA," *International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, 2008.

[16] N. Kapre, "SPICE2 - a spatial parallel architecture for accelerating the spice circuit simulator," Ph.D. dissertation, California Institute of Technology, 2010.

[17] T. Nechma, M. Zwolinski, and J. Reeve, "Parallel sparse matrix solver for direct circuit simulations on FPGAs," *Circuits and Systems (ISCAS), Proceedings of IEEE International Symposium on*, pp. 2358–2361, 2010.

[18] 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.

[19] L. de Soras, "Denormal numbers in floating point signal processing applications," 2002.

[20] 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. and Applics*, no. 4, pp. 889–901, 1997.

[21] P. R. Amestoy, Enseeiht-Irit, T. A. Davis, and I. S. Duff, "Algorithm 837: AMD, an approximate minimum degree ordering algorithm," *ACM Trans. Math. Softw.*, vol. 30, pp. 381–388, September 2004.

[22] NVIDIA Corporation, "NVIDIA CUDA C programming guide v3.2," 2010.

[23] Khronos OpenCL Working Group, "The opencl specification v1.1," 2010.

[24] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *to appear in ACM Transactions on Mathematical Software.*

[25] Advanced Micro Devices, Inc, "AMD accelerated parallel processing OpenCL programming guide v1.3," 2011.

[26] 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.