INTEGRATION, the VLSI journal 52 (2016) 10–22

Contents lists available at ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

Parallel GMRES solver for fast analysis of large linear dynamic systems on GPU platforms ☆

Kai He [a], Sheldon X.-D. Tan [a,*], Hengyang Zhao [a], Xue-Xin Liu [b], Hai Wang [c], Guoyong Shi [d]

[a] Department of Electrical and Computer Engineering, University of California at Riverside, Riverside, CA 92521, USA
[b] Synopsys Inc., Mountain View, CA 94043, USA
[c] School of Microelectronics & Solid-State Electronics, University of Electronic Science & Technology of China, Chengdu, Sichuan 610054, China
[d] School of Microelectronics, Shanghai Jiao Tong University, Shanghai 200240, China

ARTICLE INFO

Article history:
Received 17 February 2015
Received in revised form
16 June 2015
Accepted 13 July 2015
Available online 31 July 2015

Keywords:
Parallel analysis
GPU
Sparse vector and matrix multiplication
Dynamic linear systems
Circuit simulation

ABSTRACT

In this paper, we propose an efficient parallel dynamic linear solver, called *GPU-GMRES*, for transient analysis of large linear dynamic systems such as large power grid networks. The new method is based on the preconditioned generalized minimum residual (GMRES) iterative method implemented on heterogeneous CPU–GPU platforms. The new solver is very robust and can be applied to power grids with different structures as well as for general analysis problems for large linear dynamic systems with asymmetric matrices. The proposed GPU-GMRES solver adopts the very general and robust incomplete LU based preconditioner. We show that by properly selecting the right amount of fill-ins in the incomplete LU factors, a good trade-off between GPU efficiency and convergence rate can be achieved for the best overall performance. Such tunable feature can make this algorithm very adaptive to different problems. GPU-GMRES solver properly partitions the major computing tasks in GMRES solver to minimize the data traffic between CPU and GPUs to enhance performance of the proposed method. Furthermore, we propose a new fast parallel sparse matrix–vector (SpMV) multiplication algorithm to further accelerate the GPU-GMRES solver. The new algorithm, called *segSpMV*, can enjoy full coalesced memory access compared to existing approaches. To further improve the scalability and efficiency, *segSpMV* method is further extended to multi-GPU platforms, which leads to more scalable and faster *multi-GPU GMRES* solver. Experimental results on the set of the published IBM benchmark circuits and mesh-structured power grid networks show that the GPU-GMRES solver can deliver order of magnitudes speedup over the direct LU solver, UMFPACK. The resulting *multi-GPU-GMRES* can also deliver 3–12 × speedup over the CPU implementation of the same GMRES method on transient analysis.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

The verification of today's large linear global networks such as on-chip large power grid networks is very challenging for chip designers. Fast verification of voltage drops and other noises on power delivery networks is critical for final design closure. As the VLSI technology proceeds into sub-65 nm scale [15], one challenging job of power grid network is to predict and ensure a reliable on-chip power delivery. Since the power grid network usually comes with a huge size, its simulation and verification take a lot of time, and sometimes even make the analysis completely failed. Intensive researches have been carried out to seek for efficient analysis of large power grid networks in the past decade. Various algorithms have been proposed to improve scalability in computing time and to reduce memory footprints [25,36,30,9,17]. But most of those techniques are based on the homogeneous single-core architectures.

The course of computing has been permanently altered by the recent leap from single-core to multi-core or many-core technologies. Among them, the graphics processing unit (GPU), is one of the most powerful many-core computing systems arousing interests and input from both research and industry community [28]. Today, more and more high performance computing servers are equipped with GPUs as co-processors. These GPUs work in tandem with CPUs (on same computing node) connected by high-speed link like PCIe buses. GPU's massively parallel architecture allows high data throughput in terms of floating point operations (flops). For instance, the state-of-the-art NVIDIA Kepler K40c chip has a peak performance of over 4 Tflops performance in comparison with about 80–100 Gflops of Intel i7 series quad-core CPUs [3].

☆ This work is supported in part by NSF grant under No. CCF-1017090, in part by NSF Grant under No. OISE-1130402.

* Corresponding author.
E-mail address: stan@ee.ucr.edu (S.X.-D. Tan).

http://dx.doi.org/10.1016/j.vlsi.2015.07.005
0167-9260/© 2015 Elsevier B.V. All rights reserved.

Currently, GPUs or GPU-clusters can easily deliver tera-scale computing, which was only available on super-computers in the past, for solving many large scientific and engineering problems.

The NVIDIA CUBLAS library [27] provides good dense linear algebra support on GPU, but the sparse linear algebra support is still limited. Although there are some recent efforts in this direction [34,31], the sparse LU solvers on GPU is considered to be difficult due to the complicated data dependency. On the other hand, iterative solvers, which mainly depend on simple operations such as matrix–vector multiplication and inner product of vectors, are more amicable for parallelization, especially on GPU platforms. There are some newly published papers, such as [14,10,37,40,39], which confirm the practicality and effectiveness of iterative solvers in solving large linear dynamic networks like power grid networks.

Recently, there are also some research works for GPU-based iterative solver for sparse systems [13,38,8,41,4,5,12,20]. In [38], GMRES solver has been accelerated on GPU by simply parallelizing the computing of polynomial preconditioners. In [8], Jacobi-preconditioned conjugate gradient algorithm is parallelized based block compressed row storage format. But this solver only works on single GPU and symmetric matrices. Work in [20] proposed a parallel GMRES based on existing GPU-enabled BLAS library [27]. Also a few existing works have been proposed to explore the hybrid accelerators such as GPUs and Xeon Phi.

In this paper, an efficient parallel dynamic linear solver with application on transient analysis of large power grid networks of VLSI systems is proposed. We aim at developing a general GPU-accelerated dynamic linear solver, which not only can be applied to analyze power grid networks with different structures and properties, but also can be used to solve more general problems with asymmetric matrices. Examples include the power grid networks or thermal circuits with compact models, which may consist of controlled sources, constructed from model order reduction, subspace identification and other methods [22,21]. Another example is the co-simulation of the power grids and voltage regulators. As a result, the new method, called *GPU-GMRES*, is based on the preconditioned Generalized Minimum RESidual (GMRES) iterative solver, and is implemented on heterogeneous CPU–GPU platforms with multiple GPUs. The proposed GPU-GMRES solver adopts a very general and robust incomplete LU based preconditioner with tunable fill-ins. We show that by properly selecting the right amount of fill-ins in the incomplete LU factors, a good trade-off between GPU efficiency and convergence rate can be made to achieve the best overall performance of the solver. Such tunable feature can make this algorithm very adaptive and flexible for different problems.

In addition, since SpMV multiplication is a key and the most time-consuming operation in a preconditioned GMRES solver, we also propose a new fast parallel SpMV algorithm on GPU platforms. The new algorithm, called *segSpMV*, reduces the memory access by partitioning the rows, whose nonzero patterns are irregular in general, into a number of fixed-length segments. As a result, the segSpMV method can enjoy the fully coalesced memory access and outperform existing GPU-enabled SpMV methods. To further improve the scalability and efficiency, *segSpMV* method is further extended to multi-GPU platforms, which leads to more scalable and faster *multi-GPU GMRES* solver. Furthermore, since many operations in the preconditioned GMRES solver such as SpMV and sparse triangular solving are bandwidth limited operations, it is important to reduce the data communication traffics. As a result, we properly partition the major computing tasks in the GMRES solver to minimize the data traffic between CPU and GPU, which further boosts performance of the proposed method.

Experimental results on the set of the published IBM benchmark circuits and mesh-structured power grid networks show that the *GPU-GMRES* solver can deliver order of magnitudes speedup over the director solver, UMFPACK [35]. The resulting *multi-GPU-GMRES* can also deliver 3–12 × speedup over the CPU implementation of the same GMRES method on transient analysis. We also show that the matrix structures and property have a huge impact on the efficiency of GMRES solvers. Note that some preliminary results of this paper appeared in [19].

This paper is organized as follows. Section 2 reviews power grid analysis problem and the GPU architecture. Section 3 describes the proposed GPU-GMRES parallel algorithm and discussion of ILU preconditioner. In Section 4, we present a new fast parallel SpMV algorithm and its implementation on multi-GPU platforms, followed by several numerical examples in section 5. Last, Section 6 concludes this paper.

2. Review of power gird simulation and GPU architecture

2.1. The problem of power grid simulation

As shown in Fig. 1, a power grid network can be modeled as RLC (or RC) networks with known time-variant current sources, which can be obtained by gate-level logic simulations of the circuits. A typical power grid model has a tremendous size of over million nodes, and up to hundreds of thousands of input current sources. There are some nodes with known voltages in the grid, and are modeled as nodes connected with DC voltage sources. For C4 power grids, the known voltage nodes can be internal nodes inside the power grid.

The node voltages can be obtained by solving the differential equation which is formulated by modified nodal analysis (MNA),

$$\mathbf{G}\mathbf{x}(t)+\mathbf{C}\frac{d\mathbf{x}(t)}{dt}=\mathbf{B}\mathbf{U}(t), \tag{1}$$

where $\mathbf{U}(t)$ is the given current source vector, $\mathbf{G}\in\mathbb{R}^{n\times n}$ is the conductance matrix, $\mathbf{C}\in\mathbb{R}^{n\times n}$ is the matrix resulting from charge storage elements, $\mathbf{B}\in\mathbb{R}^{n\times m}$ is the input selector matrix, $\mathbf{x}(t)\in\mathbb{R}^{n}$ is the vector of time-varying node voltages and branch currents of inductors and voltage sources, and $\mathbf{U}(t)\in\mathbb{R}^{m}$ is the vector of independent power sources. In general, the matrices $\mathbf{G}$ and $\mathbf{C}$ can be asymmetric. As a result, the conjugate gradient (CG) method may not be applied to the given problem. So we adopt the more general GMRES solver in this work to solve this problem.

With the backward Euler method, the transient behavior of the power grid can be solved step by step from a given initial condition $\mathbf{x}(0)$ using

$$\left(\mathbf{G}+\frac{1}{h}\mathbf{C}\right)\mathbf{x}(t+h)=\frac{1}{h}\mathbf{C}\mathbf{x}(t)+\mathbf{B}\mathbf{U}(t+h), \tag{2}$$

where h is the time step length. If a fixed time step h is chosen, then the left-hand side matrix, $\mathbf{G}+(1/h)\mathbf{C}$, will remain the same

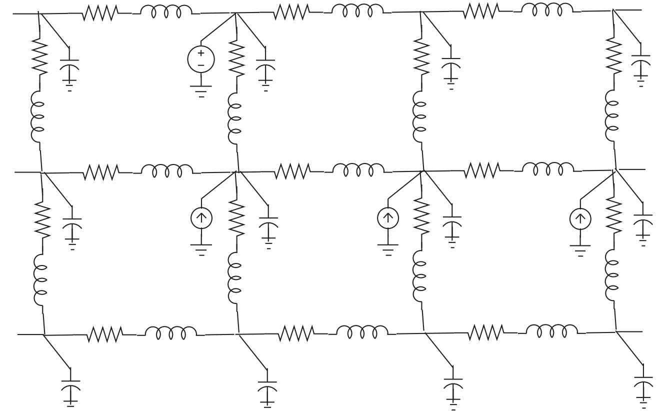


Fig. 1. An RLC model of power grid network.

along all time steps. Hence, when applying LU solver on this case, LU factorization only needs to be done once to obtain the LU factors of $\mathbf{G}+(1/h)\mathbf{C}$, and they can be reused for all the triangular solves in the following time steps.

2.2. Review of GPU architecture and CUDA programming

In this subsection, we review the GPU architecture and CUDA programming. CUDA, short for Compute Unified Device Architecture, is the parallel programming model for NVIDIA's general-purpose GPUs. The architecture of a typical CUDA-capable GPU is consisted of an array of highly threaded streaming multiprocessors (SM) and comes with up to a huge amount of DRAM, referred to as global memory. Take the Tesla C2070 GPU for example. It contains 14 SMs, each of which has 32 streaming processors (SPs, or CUDA cores called by NVIDIA), 4 special function units (SFU), and its own shared memory/L1 cache. The structure of a streaming multiprocessor is shown in Fig. 2.

As the programming model of GPU, CUDA extends C into CUDA C and supports such tasks as threads calling and memory allocation, which makes programmers able to explore most of the capabilities of GPU parallelism. In CUDA programming model, illustrated in Fig. 3, threads are organized into blocks; blocks of threads are organized as grids. CUDA also assumes that both the host (CPU) and the device (GPU) maintain their own separate memory spaces, which are referred to as host memory and device memory, respectively. For every block of threads, a shared memory is accessible to all threads in that same block. The global memory is accessible to all threads in all blocks. Developers can write programs running millions of threads with thousands of blocks in parallel. This massive parallelism forms the reason that programs with GPU acceleration can be much faster than their CPU counterparts. CUDA C provides its extended keywords and built-in variables, such as *blockIdx.*{*x,y,z*} and *threadIdx.*{*x,y.z*}, to assign unique ID to all blocks and threads in the whole grid partition.

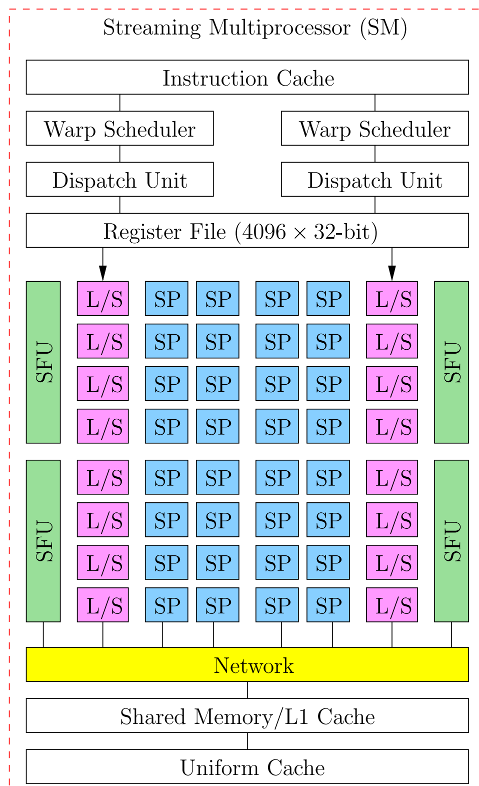


Fig. 2. Diagram of a streaming multiprocessor in NVIDIA Tesla C2070. (SP is short for streaming processor, L/S for load/store unit, and SFU for Special Function Unit.)

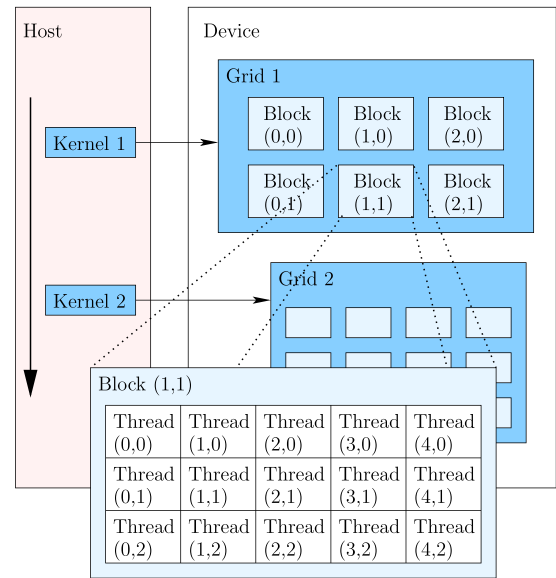


Fig. 3. The programming model of CUDA.

Therefore, programmers can easily map the data partition to the parallel threads, and instruct the specific thread to compute its own responsible data elements. Fig. 3 shows an example of 2-dim blocks and 2-dim threads in a grid, the block ID and thread ID are indicated by their row and column positions.

Although GPU provides massive parallelism and raw computing power, programming on GPUs, however, still remains a challenging problem. The reason is that many modern GPUs exhibit complex memory organization with multiple low latency on-chip memories in addition to the off-chip memory. The access latencies and the optimal access patterns of each of the memories vary significantly, posing a significant challenge to develop techniques that optimally utilize the various memories to tolerate the latency and improve the memory thoughtful. VLSI simulations task typically are memory-intensive operations as they need to analyze and transform huge amount of design data. Many operations such as SpMV multiplication [6], which is the critical kernel for most analysis and simulation tasks for VLSI chip designs, have low computing over communication ratios. This will pose difficulty to map the operations into GPUs because GPUs' performance is mainly limited by the memory bandwidth. For multi-GPUs and GPU-cluster, this becomes a even more challenging problem.

3. Parallel GMRES solver on the GPU–CPU platform

3.1. ILU-based GMRES solver

In general, our problem is how to solve a linear system

$$\mathbf{Ax}=\mathbf{b}, \tag{3}$$

In our application, the coefficient matrix is $\mathbf{A}=\mathbf{G}+(1/h)\mathbf{C}$, and the right-hand side vector is $\mathbf{b}=(1/h)\mathbf{C}\cdot\mathbf{x}_{i-1}+\mathbf{U}_i$. The index of transient point is denoted by the subscript i, i.e., $\mathbf{x}_i=\mathbf{x}(t_i)=\mathbf{x}(i\cdot h)$. The linear equation like Eq. (3) can be solved by applying the LU factorization (direct) method or iterative methods. However, the implementations of LU-factorization solver are considered to be difficult on GPU due to many inherent data dependency and irregular memory access. On the other hand, the iterative solvers, are more amenable for GPU computing as only SpMV multiplication and triangular matrix solving (in our implementation) operations are required, which are both GPU-friendly.

We investigate the GPU-accelerated GMRES iterative solver to solve the proposed power grid analysis problem. Considering the following system equivalent to $\mathbf{Ax}=\mathbf{b}$:

$$\mathbf{C}_L\mathbf{AC}_R\mathbf{y}=\mathbf{C}_L\mathbf{b}, \quad \mathbf{y}=\mathbf{C}_R^{-1}\mathbf{x}, \tag{4}$$

where $\mathbf{C}_L$ and $\mathbf{C}_R \in \mathbb{R}^{n\times n}$ are non-singular. The $\mathbf{C}_L$ and $\mathbf{C}_R$ are referred as left preconditioner and right preconditioner, respectively. The intuitive idea of preconditioning is to choose the matrices $\mathbf{C}_L$ and $\mathbf{C}_R$ so that $\mathbf{C}_L\mathbf{AC}_R$ can approximate the identity matrix. This can be done by squeezing eigenvalues of $\mathbf{C}_L\mathbf{AC}_R$ close to unity. In Eq. (4), we express this preconditioning process in the form of matrix multiplication. However, there can be other operations involved in practice. For instance, in our proposed solver, the two matrices $\mathbf{C}_L$ and $\mathbf{C}_R$ are actually the applications of lower and upper triangular solvers using the factors derived from incomplete LU factorization.

The combined efforts of the left factor and right factor in this splitting style preconditioning contribute to a more efficient GMRES, which is much better than using a single side precondition factor. Existing works have shown that such kind of simple preconditioners, e.g., diagonal (or Jacobi) preconditioner and approximate inverse preconditioner (AINV), do not have ideal preconditioning quality and they may even fail on some cases [33]. Moreover, very attractive preconditioners are defined in terms of an incomplete LU (ILU) factorization of $\mathbf{A}$. That is, we use the simplified version (or say, the cheaper variant) of LU method to compute $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{U}}$, where $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{U}}$ are sparse triangular matrices achieving the approximation $\mathbf{A}\approx\tilde{\mathbf{L}}\cdot\tilde{\mathbf{U}}$. Incomplete LU factorization is generally based on a modified Gaussian elimination, where the number of fill-in elements during factorization is strictly controlled below a preset limit. With row and column permutations, the generalized ILU can be used in most cases for preconditioning. With the application of permutation matrices, the preconditioned matrix system in Eq. (4) is

$$\mathbf{C}_L\mathbf{AC}_R=(\tilde{\mathbf{L}}^{-1}\mathbf{P})\mathbf{A}(\mathbf{Q}\tilde{\mathbf{U}}^{-1}) \tag{5}$$

where $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{U}}$ are ILU factors, and $\mathbf{P}$ and $\mathbf{Q}$ are permutation matrices. The construction of the two ILU factors shall satisfy the approximation

$$\mathbf{PAQ}\approx\tilde{\mathbf{L}}\tilde{\mathbf{U}},$$

which is equivalently to say that $\tilde{\mathbf{L}}^{-1}\mathbf{PAQ}\tilde{\mathbf{U}}^{-1}$ is an approximation to identity matrix $\mathbf{I}$.

There is a critical trade-off between this approximation and the fill-in ratio of the ILU factors. A closer approximation needs more efforts in factorization and results in high fill-in ratio of LU factors. Therefore, it will incur a high computation cost during the preconditioning process, i.e., calculating Eq. (5). On the contrary, ILU factors with low fill-in ratio are cheap to be factorized, and they also require less effort in the triangular solves, but it could take more iterations in GMRES since the spectral property of the preconditioned system deteriorates. We will study this trade-off relationship in our experiment section.

The GMRES method is an iterative method for solving large-scale systems of linear equations ($\mathbf{Ax}=\mathbf{b}$), where $\mathbf{A}$ is sparse in our case. Algorithm 1 shows the standard Krylov-subspace based GMRES method with preconditioner [33], which uses projection method to form the mth order Krylov-subspace [32,33], e.g.,

$$\mathcal{K}_m=\text{span}(\mathbf{r}_0,\mathbf{MAr}_0,(\mathbf{MA})^2\mathbf{r}_0,\ldots,(\mathbf{MA})^{m-1}\mathbf{r}_0), \tag{6}$$

where $\mathbf{r}_0=\mathbf{b}-\mathbf{Ax}_0$ and $\mathbf{M}$ is the preconditioner. Note that for the sake of simplicity, we represent the ILU preconditioning process as an operation $\mathbf{M}$ here, and from now on, all the occurrence of $\mathbf{MA}$ should denote the operation in Eq. (5), which contains two sparse triangular solves and one SpMV multiplication. After orthogonalization and normalization, the orthonormal basis of this subspace is $\mathbf{V}_m$. To generate the Krylov subspace in GMRES, Arnoldi iteration is employed to form $\mathbf{V}_m$. Each Arnoldi iteration generates a new basis vector and is appended to the previous Krylov subspace basis $\mathcal{K}_j$ to obtain the augmented subspace $\mathcal{K}_{j+1}$. Arnoldi iteration also creates an upper Hessenberg matrix $\tilde{\mathbf{H}}_m$ used to check the solution at the current iteration. As a result,the approximated solution $\mathbf{x}$ becomes the linear combination of $\mathbf{x}_m=\mathbf{x}_0+\mathbf{V}_m\mathbf{y}_m$,where $\mathbf{y}_m$ is calculated in Line 12 of Algorithm 1.

Algorithm 1. GMRES with left and right preconditioning.

Require: $\mathbf{A}\in\mathbb{R}^{n\times n}$, $\mathbf{b}\in\mathbb{R}^n$, $\mathbf{x}_0\in\mathbb{R}^n$ (initial guess),m (restart)
Ensure: $\mathbf{x}\in\mathbb{R}^n$: $\mathbf{Ax}\simeq\mathbf{b}$
1: $\mathbf{r}_0=\mathbf{b}-\mathbf{Ax}_0$
2: $\tilde{\mathbf{r}}_0=\mathbf{C}_L\mathbf{r}_0$, $\beta=\|\tilde{\mathbf{r}}_0\|_2$, $\mathbf{v}_1=\tilde{\mathbf{r}}_0/\beta$
3: **for** $j=1,2,\ldots,m$ **do** [Arnoldi iteration on GPU]
4: $\mathbf{w}=\mathbf{C}_L\mathbf{AC}_R\mathbf{v}_j$ {Eq. (5) using segSpMV and CUSPARSE csrsv}
5: **for** $i=1,2,\ldots,j$ **do** {using CUBLAS functions}
6: $h_{i,j}=\mathbf{w}_i^T\mathbf{v}_j$
7: $\mathbf{w}=\mathbf{w}-h_{i,j}\mathbf{v}_i$
8: **end for**
9: $h_{j+1,j}=\|\mathbf{w}\|_2$, $\mathbf{v}_{j+1}=\mathbf{w}/h_{j+1,j}$
10: **end for**
11: $\mathbf{V}_m=[\mathbf{v}_1,\ldots,\mathbf{v}_m]$, $\tilde{\mathbf{H}}_m=\{h_{i,j}\}_{1\le i\le j+1,\ 1\le j\le m}$
12: $\mathbf{y}_m=\text{argmin}_{\mathbf{y}}\|\beta\mathbf{e}_1-\tilde{\mathbf{H}}_m\mathbf{y}\|_2$
13: $\mathbf{x}_m=\mathbf{x}_0+\mathbf{C}_R\mathbf{V}_m\mathbf{y}_m$
14: **if** not converge **then**
15: $\mathbf{x}_0=\mathbf{x}_m$, go to Line 1
16: **end if**

The least squares problem is usually solved by computing the QR factorization of the Hessenberg matrix. In fact, the Hessenberg matrix can be maintained in factorized form by successively updating the factors. This procedure, which can be efficiently implemented by Givens rotations, is numerically reliable. However, the Gram–Schmidt orthogonalization inherent in Arnoldi method may be a source of numerical errors. Instead, we may use the modified Gram–Schmidt processes, or better, apply Householder transformations. The latter alternative is also well suited for parallel implementation.

3.2. Parallelization on GPU–CPU platforms

To parallelize the GMRES solver, we need to identify several computation intensive steps in Algorithm 1. There exist many GPU-friendly operations in GMRES, such as vector addition (`axpy`), 2-norm of vectors (`nrm2`), and SpMV multiplication (`segSpMV`). With preconditioning process, the triangular solves (`csrsv`) using ILU factors are also the beneficiaries of parallel computing, since many rows in ILU factors are independent and the solving of these rows can be done in parallel [26]. Based on the examples we focus on, we have noticed that SpMV multiplication and triangular solving take up to 70% of the overall runtime to build the Krylov subspace shown in Eq. (6). Those routines are GPU-friendly (but they are bandwidth limited operations) and efforts have been made already to parallelize these routines in generic parallel algorithms for sparse matrix computations library CUSPARSE [29].

GPU programming for many engineering problems are typically limited by the data transfer bandwidth as GPU favors computationally intensive algorithms [16]. This is especially true for operations such as SpMV multiplication and sparse triangular solving, which are bandwidth limited. For instance, SpMV has $O(n)$ communication and $O(n)$ computing, so it has 1 to 1 computing and communication ratio (n is number of non-zero elements in the sparse matrices). Hence, it is important to reduce the data communication traffic for the proposed GPU-GMRES solver.

As a result, how to wisely partition the data between CPU memory (host side) and GPU memory (device side) to minimize data traffic is crucial for GPU computing. In the sequel, we make some detailed analysis first for GMRES in Algorithm 1. Although GMRES tends to converge quickly for most circuit examples, i.e., the iteration number $m \ll n$, the space needed to store the subspace $\mathbf{V}_m$ with a size of n-by-m, i.e., m column vectors with n-length, is still big. Therefore, transferring the memory of the subspace vectors between CPU memory and GPU memory is not an efficient choice. In addition, every newly generated matrix–vector product needs to be orthogonalized with respect to all its previous basis vectors in the Arnoldi processes. To utilize the data intensive capability of GPU, we keep all the vectors of $\mathbf{v}_m$ in GPU global memory. In this case, GPU is allowed to handle those operations, such as inner-product of basis vectors (`dot`) and vector subtraction (`axpy`), in parallel.

On the other hand, it is better to keep the Hessenberg matrix $\tilde{\mathbf{H}}$, where intermediate results of the orthogonalization are stored, at the CPU host side, because of the following reasons. First, its size is $(m+1)$-by-m at most, rather small if compared with circuit matrices and Krylov basis vectors. Besides, it is also necessary to triangularize $\tilde{\mathbf{H}}$ and check the residual in each iteration so the GMRES can return the approximate solution as soon as the residual is below a preset tolerance. Hence, in light of the sequential nature of the triangularization, the small size of Hessenberg matrix, and the frequent inspection of values by the host, it is preferable to allocate $\tilde{\mathbf{H}}$ in host memory. As shown in Algorithm 1, the memory copy from device to host is called each time when Arnoldi iteration generates a new vector and the orthogonalization produces a new vector $\mathbf{h}$, which is the $(j+1)$th column of $\tilde{\mathbf{H}}$, and is transferred to the CPU, where a least square minimization (a series of Givens rotations, in fact) is performed to see if the desired tolerance of residual has been met. Our observation shows that the data transfer and subsequent CPU based computation takes up less than 0.1% of the total run time.

Fig. 4 illustrates the computation flow, the partitions of the major computing steps and the memory accesses between CPU and GPU during the operations we mentioned above.

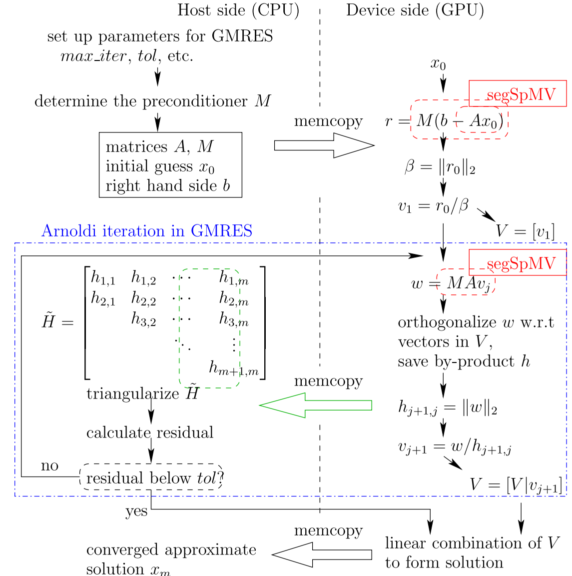


Fig. 4. The proposed GPU-accelerated parallel preconditioned GMRES solver. We also show the partitioning of the major computing tasks between CPU and GPU here.

3.3. GPU-friendly implementation of preconditioners

One important aspect of the iterative solver is the preconditioner. Preconditioners increase the rate of convergence and thus reduce the number of iterations. A well chosen preconditioner will potentially make GMRES much faster than the one without precondi- tioner. In this section, we discuss the preconditioner for GPU GMRES.

We know from the preceding discussion that in ILU preconditioning process of Eq. (5), the two major participants are $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{U}}$, who are sparse triangular matrices and approximate the $\mathbf{L}$ and $\mathbf{U}$ factors of $\mathbf{A}$, respectively. At the beginning of each Arnoldi iteration in Line 4 in Algorithm 1, this preconditioning procedure is needed to modify the property of a newly spanned Krylov subspace vector. For GMRES without preconditioner, Line 4 only consists a matrix–vector multiplication $\mathbf{A}\mathbf{v}_j$. In the new preconditioned GMRES solver, applying the ILU preconditioner requires two more operations: the solving of two sparse triangular systems (forward and backward substitutions).

For the two triangular ILU factors, we have two conflicting requirements. On one hand, the two triangular factors in ILU are supposed to approximate the complete LU factors as much as possible to increase the convergence rate. The more fill-in elements there are in $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{U}}$, the more similarities there are between the preconditioned system in Eq. (5) and the identity matrix $\mathbf{I}$. Consider an extreme example in the other end. An ILU is called ILU0 if no fill-in elements are tolerated, and existing researches have shown that ILU0's applicability on many cases is very limited due to its poor performance in accelerating convergence of iterative solvers. On the other hand, when parallelizing the triangular solving of $\mathbf{L}$ and $\mathbf{U}$ matrices in GPUs, the efficiency of the GPU solver requires less data dependency (less dependency among rows) [26]. As a result, less fill-ins benefit GPU triangular solvers [18].

As a result, in this work, we adopt the strategy of ILU with fill-in ratio control. The ILU++ package we employ in our solver allows users to provide a threshold parameter and fill-in elements smaller than this threshold will be dropped off. This parameter gives us the freedom to adjust and tune our ILU preconditioner, and delivers the optimal performance of the resulting GPU GMRES solver. But selection of the best threshold is still done by experiments and the best value is problem-specific in our work.

Once the circuit MNA matrix $\mathbf{A}$ is available, ILU is run to construct and set up the preconditioner. Then we transfer the matrices to GPU global memory. Before calling NVIDIA CUSPARSE's triangular solve function in calculating Eq. (5), there is one prerequisite step to analyze the structure of ILU factors $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{U}}$. According to CUSPARSE document, this step, which is called `csrsv_analysis`, makes an exploration of the matrix sparsity and the dependency between different rows (independent rows of triangular solve can be computed in parallel), so that information is collected and saved for future use in `csrsv_solve`. In a word, the analysis step is run only once for the whole simulation. The triangular solves in all GMRES iterations and all transient steps of circuit simulation can reuse this analysis information, and each time only `csrsv_solve` is called. More details will also be described in the experimental section.

4. Parallel SpMV algorithm on the GPU–CPU platform

As we can see, in the preconditioned GRMES solver, one key computing step is the SpMV multiplication. In this section, we present the new GPU-accelerated parallel SpMV multiplication method, *segSpmV*.

4.1. Review of existing GPU-enabled SpMV algorithms

There are many sparse matrices formats such as DIA, ELL, CSR, HYB, PKT, COO with applications ranging from highly structured matrices (DIA, ELL) to unstructured matrices (HYB, COO) [7]. Among them, the compressed sparse row (CSR) can be used for both structured and unstructured sparse matrices and has wide application for sparse matrix computations. The CSR format is a popular, general-purpose sparse matrix representation. CSR explicitly stores column indices and nonzero values in arrays *col_idx* and *data*. A third array of row pointers, *row_ptr*, takes the CSR representation as shown in Fig. 5 for a 5×5 sparse matrix. For an $M \times M$ matrix, the *row_ptr* with length $M+1$, stores the offset into *data* for the start point of each row, with the convention that $row_ptr[M] = N_{nz}$, where N_{nz} is the number of nonzeros in the matrix.

The SpMV computation consists of two phases: the first *product* phase, which performs the element–element production between the matrix and the vector, the second *summation* phase adds the results for each row to get the final result. Several relevant SpMV algorithms on GPU platforms can be summarized as follows:

4.1.1. The row-based B&G method

Bell and Garland [7] first propose a straightforward implementation, in which each row will take care of all the computing (multiplication and summation) by a single thread as shown in Fig. 6. The algorithm only requires one kernel launch (one kernel launch means one CPU-to-GPU invocation). The main drawback of this approach is that each thread will read many sequential data from a *data* vector in the CSR format from the global memory of GPUs, which leads to slow non-coalesced memory access.

4.1.2. The warp-based B&G method

The row-based *B&G* method is further improved by the warp-based *B&G* method [7] in which one warp is assigned to each row of a matrix. After the multiplication phase, the warp reduction is performed to compute the summation result. The algorithm is illustrated in Fig. 7. Compared to the row-based *B&G* method, its memory access can be coalesced because 32 continuous threads in the same warp could work together to load the non-zero elements in one row. This method, however, may suffer from low performance when the number of nonzeros in each row is smaller than 32, which can be the case for many finite difference and finite element based methods.

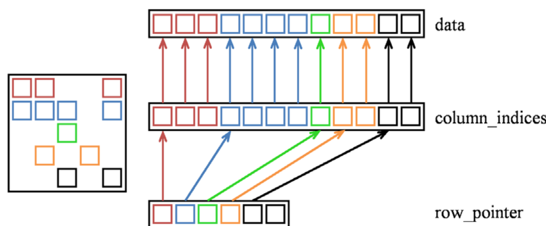


Fig. 5. The CSR format of a sparse matrix.

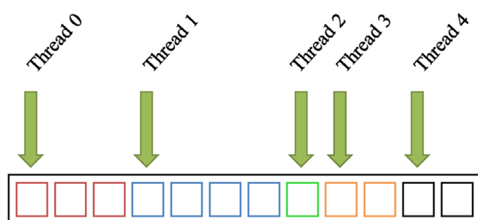


Fig. 6. The illustration of the row-based *B&G* algorithm.

4.1.3. The P&S method

Deng et al. later proposed an improved SpMV method, called *P&S* method, for many electronic design automation (EDA) related problems [12]. The approach will not directly operate on the CSR data structure. Instead, it creates a new vector, called *expanded vector*, of the same size of the *data* first as shown in Fig. 8. The *expanded vector* consists of the elements from the multiplication vector $[b1, \ldots, b3]$. And each element in the vector, *expended_vector* $[i]$, corresponds to one element in *data*, which is *data*$[i]$, and both of them will be multiplied in the production phase.

After the generation of *expanded vector,* the remaining operations are two vector multiplication and partial summation over rows. However, the method requires two sequential kernel launches, one for element-wise multiplication (or production) for the two vectors, which can enjoy fast coalesced memory access. Another one is for carrying out partial summation for each row after the vector multiplication as shown in Fig. 9. The second phase, however, cannot avoid irregular memory access because of varying length of rows. Also only one thread per row is assigned to perform the addition. To mitigate the problem, the authors proposed to load the immediate production results into the shared memory via the coalesced memory access. The threads only read from shared memory for the addition operations. But due to the limited resources of shared memory in each streaming multiprocessor (SM) in GPUs, the much slow global memory access will still be needed in case of missing data in the shared memory.

4.2. New parallel SpMV algorithm

Now we present a new parallel SpMV algorithm on the GPU–CPU platform, called *segSpMV* method. As we can see, the *P&S* method mitigates the irregular memory access for the multiplication phase by

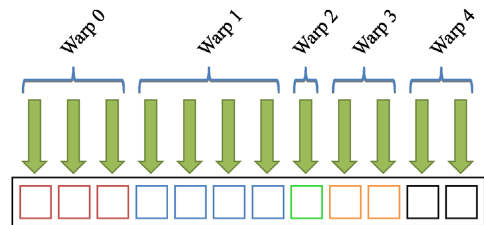


Fig. 7. The illustration of the warp-based *B&G* algorithm.

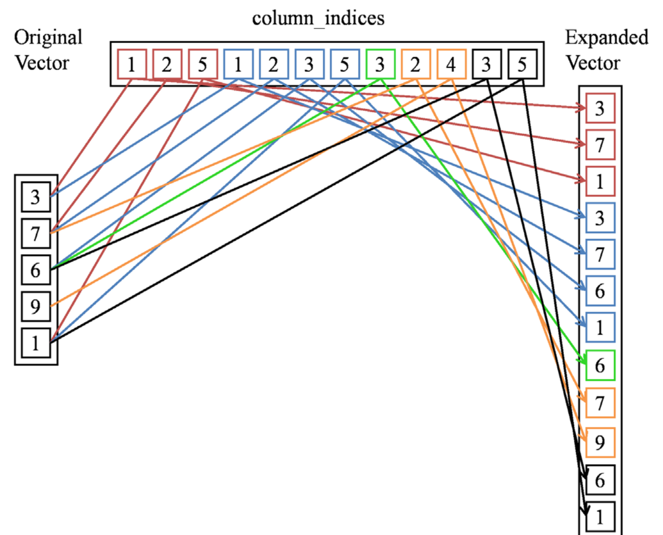


Fig. 8. The vector expansion concept in the *P&S* method.

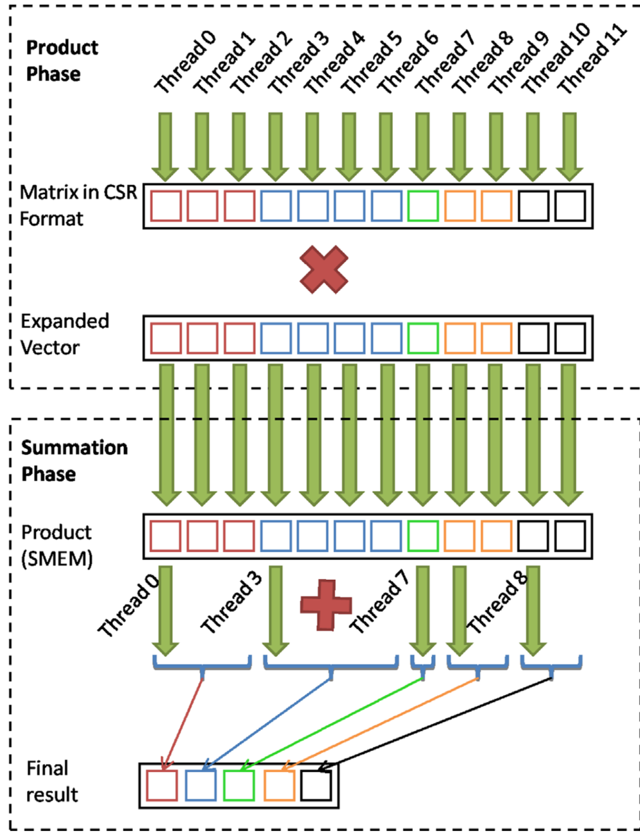


Fig. 9. The illustration of the *P&S* algorithm.

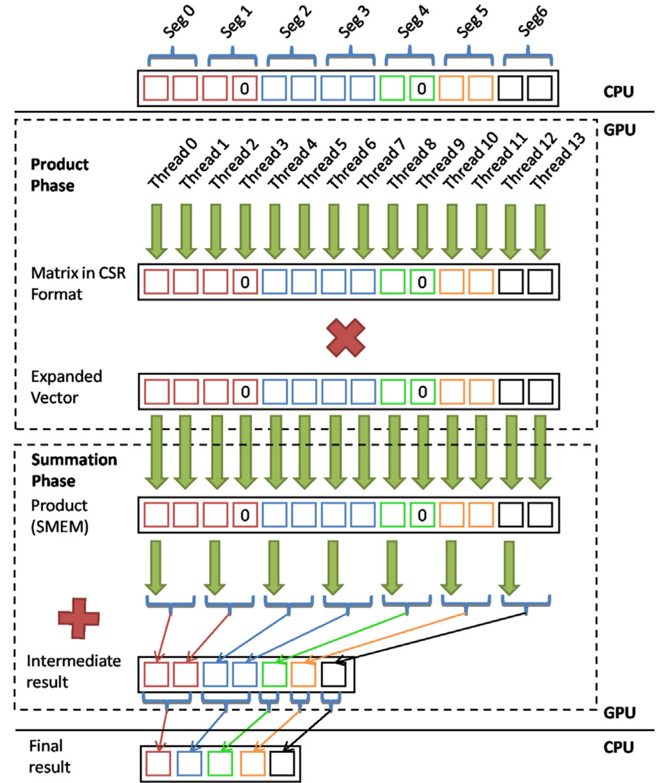


Fig. 10. The proposed segment-SpMV method or segSpMV method.

using *expanded vector*. However, in the summation phase, it still suffers the irregular memory access issue as the length of rows is irregular. Using shared memory can partially mitigate this problem. However, given the fact that the shared memory is limited, the number of nonzeros per row cannot be too large. To see this, let us assume that we need 48 KB data for each block. If we can only have 16 KB shared memory (for instance in Tesla T10 GPU). The hit rate (probability of required data in the shared memory) would be only about 33%.

In addition, the *P&S* method uses only one thread per row for the summation after loading the data into the shared memory. As a result, the row with less non-zero elements would get the summation result faster compared to the row with more non-zero elements. Therefore, the memory access cannot be full coalesced and the performance of *P&S* method is limited by imbalanced workload. Furthermore, it requires two sequential kernel launches. In the second launch for the summation, one has to load intermediate production result vector and the *row_ptr* vector from global memory.

The segSpMV method can overcome the aforementioned problems in the existing *P&S* method. The new algorithm is also based on the expanded vector concept for the multiplication phase. But different from the *P&S* method, the new algorithm can mitigate the irregular memory access problem in the summation phase, and thus lead to more simple implementation and yet better performance. The main idea is to partition the rows into a number of fixed-length regular segments before the operation. The length of the segment typically is selected to be just bigger than the average number of nonzero elements per row in the given matrix and they also should be the power of 2 for easy reduction operation. For instance, if the average number of nonzero elements is 14, then segment length $2^4 = 16$ is selected. For rows with more nonzeros than the average number, multiple segments will be needed.

After the segment length is determined, each row is partitioned into a number of regular segments. If a segment is not fully filled by the elements from the given row, 0 is padded to the rest of the empty positions in the segment, as shown in Fig. 10. In this figure, one 0 is padded at the end of segment2. We perform this segment-based expansion for both original vector and the *expanded vector* of the matrix. After this step, the two segment-expanded vectors are sent to GPU global memory for multiplication and addition phases with just one kernel launch as shown in Fig. 10. Note that it takes $O(N_{nz})$ to do the zero padding. In the product phase, each thread first will read two elements from the two segment-expanded vectors, respectively via the coalesced memory access from the GPU global memory. Then each thread multiplies this pair of elements. But it stores the product result immediately into the shared memory instead. In this case, all the intermediate product results from all threads are stored in shared memory, which is ready for the second phase of addition operation right away.

In the summation phase, the new algorithm does not need to check the boundaries of each row any more, which causes the irregular memory access, as it can simply add all the results for each regular segment instead. Because the segment size is fixed, the summation can be very easily done by one thread or by multiple threads via reduction. Also the addition operation will take almost same time for all the threads. We add the *synchronize*() to ensure all the partial results from each segment finish first before they are written back into shared memory using the coalesced memory access. Finally, *segSpMV* adds up the immediate results of segments corresponding to the same row to get the final results in the CPU side, which can be done very efficiently.

We note that the new method will never run out of shared memory, which is the major advantage of the proposed method over the existing approach. The reason is that the amount of memory needed is 4 times of number of threads in each block, as the size of each intermediate element is 4 Byte. So given 1 K maximum thread allowed in each block in K20c and K40c GPUs,

the maximum memory is just 4 KB, which is far less than the 48 KB shared memory in each SM. This is also the case for other GPUs as well. As a result, we do not need to write the product results back to global memory and then read them back again, which leads to one more kernel launch. In the addition phase, each thread sums products in one segment and each block is responsible for the same number of segments. The number of non-zero elements in each row may be different, but all segments are with the same length. Compared to the *P&S* method, we do not need to check if a data is cached in shared memory and do not need to worry about the low hit rate when the number of non-zero elements per row is large as we make full use of the shared memory and has equivalent 100% hit rate in this sense.

5. Numerical results and discussion

All the aforementioned methods are implemented in C programming language. The GPU part of the proposed new method is incorporated into the main program with CUDA C programming interface.

To put our new simulator's performance into a right perspective, we compare multi-GPU GMRES with CPU GMRES and a standard LU-based method based on UMFPACK [35]. We remark that we do not compare our multi-GPU GMRES solver with other iterative solvers as most of existing iterative solvers are highly tuned to specific problems, and are not general enough for general linear systems. On the other hand, the proposed multi-GPU GMRES solver is a general solver for any linear dynamic systems, which include but do not limit to the examples of power grid circuits and thermal circuits for both symmetric and non-symmetric matrices. In addition, it does not assume or exploit any structures of the given systems. As a result, it will be more fair to compare our tool with the general LU-based simulator.

These programs are tested on a Linux server with an Intel 2.4 GHz Xeon Quad-Core CPU chip. The host (CPU) side has a total of 60 GBytes memory available. Meanwhile, the server has three GPU cards (devices) as mentioned earlier and are repeated here: one Tesla K40c containing 2880 cores with 12 GBytes global memory, one Tesla K20c containing 2688 cores with 6 GBytes global memory and one Tesla C2075 containing 448 cores with up to 5 GBytes global memory. But we only use the Tesla K40c and K20c in the new multi-GPU GMRES solver.

5.1. segSpMV performance comparison on public matrices

To perform the comparisons, several mentioned algorithms have been implemented or obtained from the published sources as listed below:

Table 1
The matrices and their properties from UFL Sparse Matrix Collection.

Matrices	Row	nzsize	nzperrow	seg_length
scircuit	170,998	958,936	5.61	8
mac-econ-fwd500	2,06,500	12,73,389	6.17	8
cop20k-A	1,21,192	13,62,087	11.24	16
qcd5-4	49,152	19,16,928	39.00	32
cant	62,451	20,34,917	32.58	32
mc2depi	5,25,825	21,00,225	3.99	4
pdb1HYS	36,417	21,90,591	60.15	64
rma10	46,835	23,74,001	50.69	64
consph	83,334	30,46,907	36.56	32
webbase-1M	10,00,005	31,05,536	3.11	4
shipsec1	1,40,874	39,77,139	28.23	32
pwtk	2,17,918	59,26,171	27.19	32

- segSpMV, the proposed method.
- *P&S*, the *P&S* method.
- *B&G*-s, the *B&G* method using single thread per row [1].
- *B&G*-w, the *B&G* method using one warp per row [1].
- *cu*, the NVIDIA CUSPARSE library SpMV function.

We perform the comparison on the set of matrices from University of Florida Sparse Matrix Collection [11] as shown in Table 1 in which *nzsize* means number of nonzeros and *nzperrow* is the average number of nonzeros per row. *seg_length* is the segment length used for the proposed methods. All the matrices are ranked with increasing number of *nzsize* from top to bottom and those matrices represent various matrix structures from wide applications.

Table 2 first shows the performance comparison on the matrices in Table 1 on the latest Tesla K40c GPU for the five algorithms. It can be seen that the proposed segSpMV method beats all the other algorithms on *ALL* the matrices with various structures. The average speedups over *B&G*-s, *B&G*-w and *P&S* methods are $9.09\times$, $7.27\times$ and $3.88\times$, respectively. Speedup in some cases such as *webbase-1M* can be order of magnitude faster over three other algorithms. In addition, we also provide the comparison results between the proposed segSpMV method and NVIDIA CUSPARSE library function. The speedup ranges from $1.17\times$ to $2.01\times$, with average $1.58\times$. Although the speedup highly depends on the benchmark matrices, we see the $>1\times$ speedup on all the cases.

5.2. Multi-GPU segSpMV implementation and performance comparison

To further utilize the multiple GPU resources and make the proposed segSpMV method more scalable for handling much larger problems, we further extended segSpMV algorithm into the multi-GPU platforms.

Specifically, the segSpMV method can be easily divided into several tasks. First, we partition the two expanded vectors into several segment groups, and each group is managed by a CPU thread. The number of groups can be determined by the number of GPU devices on the server. Second, each CPU thread passes the corresponding segments to one GPU device, and GPU just finishes the computation of multiplication and addition phases with one kernel launch. Since the sparse matrix and vector are already expanded into several segments with a fixed length, the task partition and distribution become very simple. Furthermore, the segSpMV method is very multi-GPU friendly as there is no inter-GPU communication. Each GPU can still enjoy the full coalesced memory access and shared memory utilization.

Our multi-GPU server consists of one Tesla K40c, one Tesla K20c, and one C2075 GPUs. The server also consists of two 8-Core Xeon E5-2670 CPUs, DDR3-1600 64 GB memory. The Tesla K40c and K20c GPU are built on the NVIDIA Kepler compute architecture and have 2880 and 2688 CUDA parallel processing cores, respectively. The K40c is capable of running 4.29 Tflops per second of single precision processing performance while K20c has the peak 3.95 Tflops single precision floating performance. C2075 is based on previous Fermi architecture GPU with 448 cores and 1 Tflops peak single precision performance.

The resulting *multi-GPU segSpMV* method can gain further speedup as shown in Fig. Fig. 11 in addition to the added scalability. The performance comparison is based on the matrices in Table 1 for single GPU (K40c), 2-GPUs (K40c and K20c) and 3-GPUs (K40c, K20c and C2075). It can be seen that the performance differences are very small when the matrix size is small. It is due to the overhead of creating new CPU threads, starting GPU and performing synchronization. However, the speedups in larger cases are much better. For example, for the largest matrix *pwtk*, the 2-GPU and 3-GPU implementations are 66% and 87% faster

Table 2
The performance comparison over UFL matrices on K40c GPU.

1	2	3	4	5	6	7	8	9	10
Matrices name	Algorithm					Speedup			
	B&G-s (ms)	*B&G-w* (ms)	*P&S* (ms)	*cu* (ms)	*seg* (ms)	*B&G-s*	*B&G-w*	*P&S*	*cu*
scircuit	0.352	1.063	0.174	0.195	0.118	2.98	9.01	1.47	1.65
mac-econ-fwd500	0.435	1.757	0.242	0.254	0.153	2.84	11.48	1.58	1.66
cop20k-A	0.932	0.871	0.285	0.251	0.146	6.38	5.97	1.95	1.72
qcd5-4	1.903	0.762	0.523	0.263	0.188	10.12	4.05	2.78	1.40
cant	2.068	0.821	0.528	0.330	0.204	10.14	4.02	2.59	1.62
mc2depi	0.248	2.238	0.297	0.349	0.196	1.27	11.47	1.52	1.78
pdb1HYS	2.416	0.909	0.697	0.373	0.215	11.24	4.23	3.24	1.73
rma10	2.303	1.019	0.702	0.401	0.257	8.96	3.96	2.73	1.56
consph	3.087	1.218	0.791	0.401	0.303	10.19	4.02	2.61	1.32
webbase-1 M	14.502	10.843	11.439	1.066	0.531	27.31	20.42	21.54	2.01
shipsec1	3.198	1.704	0.882	0.512	0.378	8.46	4.51	2.33	1.35
pwtk	5.167	2.299	1.256	0.662	0.565	9.15	4.07	2.22	1.17
Average						9.09	7.27	3.88	1.58

than the single GPU implementation. We also notice that the 2-GPUs and 3-GPUs implementations have similar performance. We also notice that the K40c and K20c are much more powerful than the C2075. So the computing speed of 3-GPUs implementation is mainly determined by C2075, which limits the performance improvement. But the results from Fig. 11 clearly demonstrates the advantages and benefits of the proposed multi-GPU segSpMV over the single GPU segSpMV method.

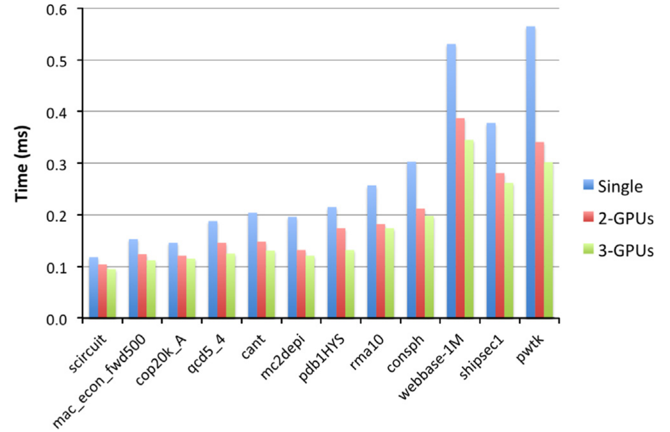


Fig. 11. The performance comparison of multi-GPU segSpMV method.

5.3. Accuracy comparison and discussion

We first test the accuracy and efficiency of our solver on the power grid circuits from IBM benchmark suite [2]. There are 6 benchmark circuits with sizes ranging from forty thousand to three million nodes in the interconnection. The information of these benchmarks can be retrieved from their website. We show the matrix sizes of their circuit MNA models in Table 3. Also in the same table, the running time spent in LU factorizations and LU solves of the backward Euler equations are also listed. The equation solved here is stated in Eq. (3). Since we use uniform discretization in the time domain, the time step length h remains the same on all the steps. In addition, all of our examples are linear circuits, and the matrices $\mathbf{G}$ and $\mathbf{C}$ do not change either. As a result, the LU factorization only needs to be calculated once on $\mathbf{G}+(1/h)\mathbf{C}$ and its triangular $\mathbf{L}$ and $\mathbf{U}$ are reused for all the transient steps. The time measurements in Column "LU fact." are the one time cost of LU factorization, and those in Column "LU solve" are time spent on LU triangular solve on one time step, i.e., solving $\mathbf{Ax}=\mathbf{b}$ with reuse of LU factors.

The error tolerance of all of our GMRES solvers is set to 10^{-7}. A smaller tolerance guarantees higher accuracy, but also leads to more iterations and longer solving time. During our extensive experiments with the benchmarks, we have found a 10^{-6} tolerance is good and accurate for most cases. Nonetheless, we use 10^{-7} for all experiments as this will give us statistics according to the same standard. We do not push our tool only for a demonstration of speed with the sacrifice of accuracy.

Fig. 12 shows the simulation results of a benchmark circuit ibmpg6t, from IBM. It is a voltage waveform at node n0_2679_17913. We plot the waveforms of the direct LU method and multi-GPU GMRES with preconditioner on the same figure, and the accuracy of GMRES result is quite satisfactory since the two curves are closely overlapped. To further show the accuracy, we plot the errors of the GMRES curve, i.e., the difference between GMRES result and LU result, in Fig. 13, which shows about 1% maximum relative error. We have verified all the examples, especially waveforms at the observation port nodes listed by `print` command in IBM netlists, and all the waveforms from GPU GMRES agree with the LU golden results.

5.4. Computing time comparison and discussion

Table 3 lists the running time measurements in the benchmarks. Column 5 (C5) gives the threshold value used for control the fill-ins in the ILU preconditioner. The column C6 lists preconditioner setup time, C9 is for the multi-GPU GMRES solving time without initial guess available, and C12 is for the multi-GPU GMRES solving time on each transient point, when good initial guess is available. The speedup of multi-GPU GMRES over LU on DC solving is listed in C13. The speedup of multi-GPU GMRES over LU on the whole simulation (1000 time steps) is listed in C14.

We first discuss the results on the IBM examples. Among the six IBM circuits, multi-GPU GMRES brings reasonable speedup over LU factorization. To make a fair competition with LU, the speedup on DC solving, i.e., the first GMRES solves without any good initial guess available, shall be calculated as (C3+C4)/(C6+C9). The biggest speedup for this initial DC solving is 97 ×, which happens in the case of ibmpg3t. We notice that the speedup does not always go up with the size of the circuit as shown in Table 3. We observe that these IBM benchmarks vary not just in sizes, but also in the circuit structure and thus the their matrix structures. But still the proposed parallel GMRES solver show decent speedup over the direct method on these industrial design examples. We also observe that the multi-GPU GMRES solver will have about 4–5 × speedup over their CPU version of GMRES solver

Table 3
Statistics of IBM power gird benchmarks and solver performance. Column 14 lists the speed up of GPU GMRES over LU method on all the 1000 time step points in a transient simulation calculated as (C3 + 1000 · C4)/(C6 + C9 + 1000 · C12).

1	2	3	4	5	6	7	8	9	10	11	12	13	14
		LU		GMRES								sp. up DC	sp. up
Circuit name	Matrix size	Fact.	Solve	ILU	Precond.	Solving on DC			Solving on tran. step (ave.)			(C3+C4)/(C6+C9)	
		(s)	(s)	thres.	setup (s)	# iter	CPU (s)	GPU (s)	# iter	CPU (s)	GPU (s)		
ibm1t	54,265	0.19	0.02	2.1	0.10	33	0.35	0.07	7	0.03	0.01	1.2	2.0
ibm2t	164,897	9.93	0.06	1.2	0.62	143	3.50	0.51	23	0.54	0.06	8.8	1.2
ibm3t	1,043,444	638.7	0.87	2.6	5.03	25	6.41	1.55	6	1.10	0.35	97	4.2
ibm4t	1,214,288	904.7	1.01	1.9	9.65	77	23.2	4.69	10	3.15	0.57	63	3.3
ibm5t	2,092,148	241.6	0.60	1.5	5.80	118	22.1	4.41	17	3.36	0.49	24	1.7
ibm6t	3,203,802	174.3	0.82	2.2	12.49	42	15.2	3.44	9	3.40	0.52	11	1.9
rlc80	32,064	6.97	0.01	1.8	0.12	29	0.12	0.28	4	0.01	0.003	17	5
rlc100	50,200	28.60	0.02	1.8	0.17	32	0.18	0.38	4	0.02	0.003	52	14
rlc120	72,384	102.2	0.05	1.9	0.26	32	0.28	0.44	4	0.02	0.008	146	18
rlc140	98,616	255.6	0.08	2.0	0.36	32	0.39	0.49	4	0.04	0.008	301	38
rlc160	128,896	726.3	0.15	2.0	0.48	34	0.51	0.52	4	0.06	0.008	726	97
rlc180	163,224	2033.6	0.28	2.0	0.68	34	0.65	0.63	4	0.10	0.008	1552	248
rlc200	201,600	4191.3	0.39	2.0	0.85	35	0.82	0.64	4	0.14	0.025	2813	173
rlc220	244,024	6750.9	0.54	2.1	1.09	35	1.01	0.78	4	0.19	0.017	3610	386
rlc800	3,235,200	–	–	2.1	25.93	35	16.86	0.63	5	1.56	0.13	–	–
rlc1000	5,056,000	–	–	2.0	91.88	36	28.09	0.75	6	3.66	0.22	–	–
rlc1200	7,281,600	–	–	2.2	139.06	36	46.43	1.39	5	5.12	0.33	–	–

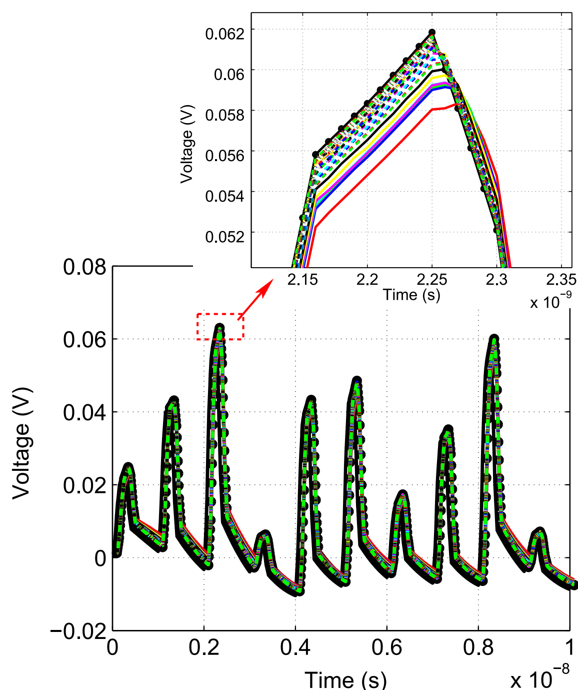


Fig. 12. Transient waveforms of LU and GPU GMRES at port node n0_5480720_1102640 in ibmpg6t. The black curve with dots is from LU direct method. All other colored curves are results of GMRES with preconditioners set to different ILU thresholds, i.e., from 0.1 to 3.0. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

on those IBM benchmark circuits (not shown in the table), which clearly shows the advantages and benefits of GPU based computing.

For transient analysis, we observe that when the LU factors are available, it seems cheaper for LU triangular solve than iterative methods to compute the solution. Since fixed time step is used in our simulator and the triangular LU factor matrices do not change as we mentioned in the previous sections, it is very understandable that GMRES does not superbly beat the triangular solve if the examples are relatively small. However, as the average running time listed in C12 of GMRES solve is smaller than C4 of LU solve, the total reduction of cost will still be favored when there are a lot of transient steps. If LU factorization has to be done many times, as happens in transient simulation with changing time steps, GMRES solver will be faster than the LU factorization.

Now we discuss the results on some RLC mesh circuits, which are the middle eight examples in Table 3 with "rlc" in circuit names. Those power grid networks are generated based on RLC mesh grid circuit model shown in Fig. 1. We observe that the speedups of the proposed method over LU factorizations in both DC and transient analysis are much larger (ranging from 5 to 3610) and speedup goes up with the sizes of the circuits. This indicates that the structures of the power grid networks have huge impacts on the solving efficiency and their final computing speed. Similarly, we observe that the multi-GPU GMRES solver will have about 3–12 × speedup over their CPU version of GMRES solver on those IBM benchmark circuits for transient analysis, although the speedup is marginal for DC analysis. As a result, it seems that IBM examples favor the LU based solver, while our mesh-structured RLC networks favor the proposed GMRES solver.

To show the added scalability of the new parallel GMRES solver on multi-GPU platforms, we also provide three very large RLC mesh circuits, which are the rlc800, rlc1000 and rlc1200. They are all million-sized circuits and cannot be handled in single GPU card with limited global memory. But our multi-GPU GMRES solver is able to handle such large circuits easily. We notice that the LU factorization method is too slow for these large circuits. As a result, we do not show the results and speedup comparison for the LU solver. We observe that the multi-GPU GMRES solver will have about 12–17 × speedup over their CPU version of GMRES solver on those large circuits for transient analysis, and the speedup for DC analysis is also between 30 and 60%, which is better than the speedup on small circuits.

5.5. *Preconditioner study and discussion*

Now, let us study the quality of an ILU preconditioner. The fill-in ratio is a good indicator about the quality of incomplete LU

preconditioner. It is calculated as the ratio of the number of fill-in elements in incomplete LU factors $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{U}}$ over the number of non-zero elements in the original coefficient matrix $\mathbf{A}$, i.e,

$$\text{fill-in ratio} = [\text{nnz}(\tilde{\mathbf{L}}) + \text{nnz}(\tilde{\mathbf{U}}) - n]/\text{nnz}(\mathbf{A}).$$

Notice that the diagonal of lower triangular factor $\text{nnz}(\tilde{\mathbf{L}})$ is unitary and need not be stored in practice. This also explains the subtraction of matrix size n in the equation above. For the simplest incomplete LU preconditioner ILU0, which computes the LU factorization but drops any fill-in elements in $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{U}}$ outside of the nonzero pattern of $\mathbf{A}$, the fill-in ratio is 1.0. This means the number of non-zero elements in ILU0 factors are equal to that of $\mathbf{A}$'s. To the best of our knowledge, NVIDIA has released a function of ILU0 factorization in the most recent CUSPARSE 5.0 version [29]. However, it has no fill-ins and does not support row/column permutation, and our experiments show that the two limitations hurt its applicability to the circuit cases here. Instead, we use the ILU package from [23], who allows different fill-in ratios by modifying the dropping threshold. This threshold parameter controls the dropping rule during incomplete LU factorization and affects the behavior of ILU preconditioner. The detailed description of the dropping rule can be found in [24]. Though low fill-in ratio implies a simple structure in the two triangular factors and a possibly faster computation in GPU's triangular solve, it results in more iterations in GMRES solver and may not be optimal in terms of overall computation time of GMRES. In addition, the time spent on preconditioner construction also grows up in order to compute more fill-in elements. Table 4 shows the relationship among the threshold, fill-in ratio, the iteration numbers, and the total GMRES CPU time. It can be seen that the CPU time reaches the minimum value when the threshold is 1.9. Fig. 14 depicts the aforementioned relationships. The data in this figure are measured from 30 runs of the same circuit *ibmpg4t*, where only the threshold is changed from 0.1 to 3.0 with 0.1 increment (only half of the data are shown). The effects of this change on fill-in ratio and GMRES time on each time step are shown by two curves.

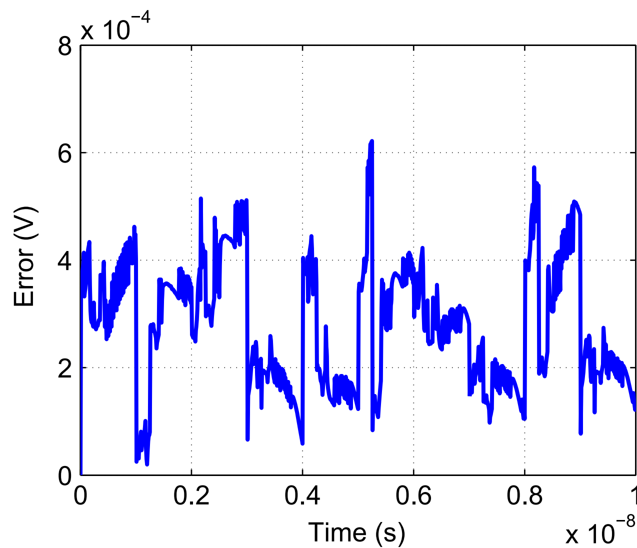


Fig. 13. The error of GPU GMRES result compared to LU golden result. This curve is calculated at node n0_5480720_1102640 of ibmpg6t, whose waveform is shown in Fig. 12.

Table 4
The performance comparison of ILU preconditioners with different fill-in ratios. The same circuit matrix from IBM power gird benchmark ibmpg4t is used in all the cases. GMRES convergence tolerance is set to 10^{-7}.

Threshold	Precond setup (s)	ILU fill-in	# Iter on DC	# Iter per tran step	Total time (s)
0.1	5.46	0.31	3447	913	12531.6
0.3	5.69	0.53	1469	440	6413.9
0.5	5.28	0.65	690	310	4773.2
0.7	5.93	0.92	480	115	1905.8
0.9	6.18	1.29	366	68	1358.9
1.1	6.67	1.70	237	32	812.6
1.3	6.92	1.99	210	26	821.8
1.5	7.28	2.35	126	19	720.5
1.7	7.74	2.77	109	16	664.3
1.9	9.65	4.06	77	10	645.6
2.1	12.38	5.42	47	7	727.5
2.3	16.05	6.81	39	6	753.3
2.5	20.83	8.18	30	5	804.8
2.7	27.57	9.78	37	5	941.3
2.9	37.30	11.61	37	4	1132.7
3.0	42.68	12.55	19	4	1233.4

6. Conclusion

In this paper, we have proposed an efficient parallel solver *GPU-GMRES* for large linear dynamic systems. The new solver is based on the preconditioned GMRES solver implemented CPU–GPU platforms. The proposed GPU-GMRES solver is based on the very general and robust incomplete LU based preconditioner. We have shown that by properly selecting the right amount of fill-ins in the LU factors, a good trade-off between GPU efficiency and convergence rate can be achieved for the overall best performance. In addition, a new fast parallel SpMV multiplication algorithm is proposed to further accelerate the GMRES solver. The new algorithm, called *segSpMV*, can enjoy full coalesced memory access. To further improve the scalability and efficiency, *segSpMV* method is further extended to multi-GPU platforms. The resulting *multi-GPU segSpMV* can deliver further performance enhancement for the resulting *multi-GPU-GMRES* solver. Furthermore, we have properly partitioned the major computing tasks in GMRES solver to minimize the data traffic between CPU and GPU, which further boosts performance of the proposed method. Experimental results on the set of the published IBM benchmark circuits and mesh-structured power grid networks have shown that the GPU-GMRES solver can deliver order of magnitudes speedup over one direct LU solver. The resulting *multi-GPU-GMRES* can also deliver 3–12 × speedup over

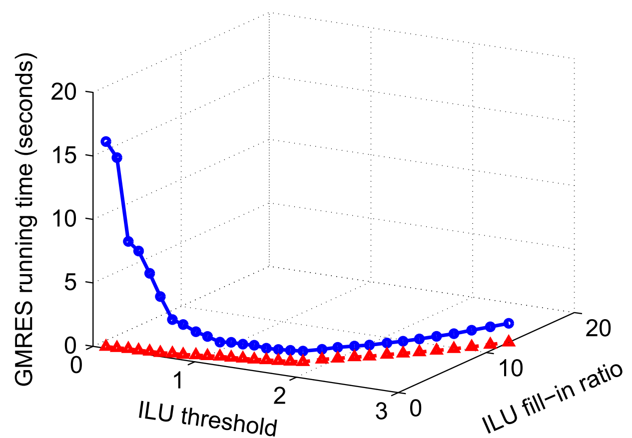
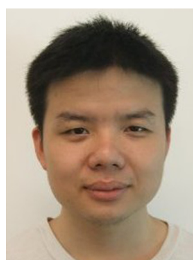


Fig. 14. The impact of ILU threshold on fill-in ratio and GMRES solving time. The blue curve in 3D space is GMRES solving time with respect to threshold and fill-in ratio, and the red curve on the bottom plane reflects the changes of fill-in ratio caused by different threshold values. All the measurements are from ibmpg4t. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

the CPU implementation of the same GMRES method on transient analysis.

References

[1] Cusp-library – generic parallel algoritgms for sparse matrix and graph computations, ⟨http://code.google.com/p/cusp-library⟩.
[2] IBM power grid benchmarks, ⟨http://dropzone.tamu.edu/pli/PGBench/⟩.
[3] NVIDIA Tesla's Servers and Workstations, ⟨http://www.nvidia.com/object/tesla-servers.html⟩.
[4] J.M. Bahi, R. Couturier, L.Z. Khodja, Parallel GMRES implementation for solving sparse linear systems on GPU clusters, in: Proceedings of the 19th High Performance Computing Symposia, Society for Computer Simulation International, ser. HPC '11. San Diego, CA, USA, 2011, pp. 12–19. [Online]. Available: ⟨http://dl.acm.org/citation.cfm?id=2048577.2048579⟩.
[5] J. Bahi, R. Couturier, L. Khodja, Parallel sparse linear solver GMRES for GPU clusters with compression of exchanged data, in: Euro-Par 2011: Parallel Processing Workshops, Lecture Notes in Computer Science, M. Alexander, P. D'Ambra, A. Belloum, G. Bosilca, M. Cannataro, M. Danelutto, B. DiMartino, M. Gerndt, E. Jeannot, R. Namyst, J. Roman, S. Scott, J. Traff, G. Valle, J. Weidendorfer (Eds), vol. 7155, Springer, Berlin, Heidelberg, 2012, pp. 471–480. [Online]. Available: ⟨http://dx.doi.org/10.1007/978-3-642-29737-3_52⟩.
[6] M.M. Baskaran, R. Bordawekar, Optimizing sparse matrix–vector multiplication on GPUs, IBM Research Division, IBM Research Report RC24704, April 2009.
[7] N. Bell, M. Garland, Efficient sparse matrix–vector multiplication on CUDA, NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, December 2008.
[8] L. Buatois, G. Caumon, B. Levy, Concurrent number cruncher: a GPU implementation of a general sparse linear solver, Int. J. Parallel Emerg. Distrib. Syst. 24 (3) (2009) 205–223.
[9] T. Chen, C.C. Chen, Efficient large-scale power grid analysis based on preconditioned Krylov-subspace iterative method, in: Proceedings of Design Automation Conference (DAC), 2001, pp. 559–562.
[10] K. Daloukas, N. Evmorfopoulos, G. Drasidis, M. Tsiampas, P. Tsompanopoulou, G. Stamoulis, Fast transform-based preconditioners for large-scale power grid analysis on massively parallel architectures, in: Proceedings of International Conference on Computer Aided Design (ICCAD), November 2012, pp. 384–391.
[11] T.Davis, The University of Florida sparse matrix collection, ⟨http://www.cise.ufl.edu/research/sparse/⟩.
[12] Y. Deng, B. Wang, S. Mu, Taming irregular EDA applications on GPUs, in: 2009 IEEE/ACM International Conference on Computer-Aided Design – Digest of Technical Papers, ICCAD 2009, 2009, pp. 539–546.
[13] Z. Feng, P. Li, Multigrid on GPU: tackling power grid analysis on parallel SIMT platforms, in: Proceedings of International Conference on Computer Aided Design (ICCAD), 2008, pp. 647–654.
[14] Z. Feng, Z. Zeng, P. Li, Parallel on-chip power distribution network analysis on multi-core–multi-GPU platforms, IEEE Trans. Very Larg. Scale Integr. (VLSI) Syst. 19 (10) (2011) 1823–1836.
[15] International technology roadmap for semiconductors (ITRS), 2012 update, 2012, ⟨http://public.itrs.net⟩.
[16] D.B. Kirk, W.-M. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, second ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, 2013.
[17] Y. Lee, Y. Cao, T. Chen, J. Wang, C. Chen, HiPRIME: Hierarchical and passivity preserved interconnect macromodeling engine for RLKC power delivery, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 24 (6) (2005) 797–806.
[18] R. Li, Y. Saad, GPU-accelerated preconditioned iterative linear solvers, J. Supercomput. 63 (2) (2010) 443–466 http://static.msi.umn.edu/rreports/2010/112.pdf.
[19] X. Liu, H. Wang, S.X.-D. Tan, Parallel power grid analysis using preconditioned gmres solvers on cpu–gpu platforms, in: Proceedings of International Conference on Computer Aided Design (ICCAD), November 2013, pp. 561–568.
[20] X. Liu, K. Zhai, Z. Liu, K. He, S.X.-D. Tan, W. Yu, Parallel thermal analysis of 3D integrated circuits with liquid cooling on CPU–GPU platforms, IEEE Trans. Very Larg. Scale Integr. (VLSI) Syst. 3 (March) (2015) 575–579.
[21] Z. Liu, S. Swarup, S.X.-D. Tan, H. Chen, H.Wang, Compact lateral thermal resistance model of TSVs for fast finite-difference based thermal analysis of 3D stacked ICs, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 33 (10) (Oct 2014).
[22] Z. Liu, S.X.-D. Tan, H. Wang, Y. Hua, A. Gupta, Compact thermal modeling for packaged microprocessor design with practical power maps, Integr. VLSI J. (47) (January (1)) 2014, [Online]. Available: ⟨http://www.sciencedirect.com/science/article/pii/S0167926013000412⟩.
[23] J. Mayer, ILU++ package, ⟨www.iluplusplus.de/⟩.
[24] A multilevel Crout ILU preconditioner with pivoting and row permutation, Numer. Linear Algebra Appl. 14 (10) (2007) 771–789.
[25] S.R. Nassif, J.N. Kozhaya, Fast power grid simulation, in: Proceedings of Design Automation Conference (DAC), 2000, pp. 156–161.
[26] M. Naumov, Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU, NVIDIA Technical Report NVR-2011-001, NVIDIA Corp., June 2011.
[27] NVIDIA Corporation, CUBLAS library v5.0, ⟨https://developer.nvidia.com/cublas⟩.
[28] NVIDIA Corporation, ⟨http://www.nvidia.com⟩, 2011.
[29] NVIDIA Corporation, CUSPARSE library v5.0, ⟨http://developer.nvidia.com/cuSPARSE⟩, October 2012.
[30] H.F. Qian, S.R. Nassif, S.S. Sapatnekar, Random walks in a supply network, in: Proceedings of Design Automation Conference (DAC), 2003, pp. 93–98.
[31] L. Ren, X. Chen, Y. Wang, C. Zhang, H. Yang, Sparse LU factorization for parallel circuit simulation on GPU, in: Proceedings of Design Automation Conference (DAC), 2012, pp. 1125–1130.
[32] Y. Saad, M.H. Schultz, GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. (1986) 856–869.
[33] Y. Saad, Iterative Methods for Sparse Linear Systems. PWS Publishing, 2000.
[34] A.M. Sridhar, A. Vincenzi, et al., 3D-ICE: Fast compact transient thermal modeling for 3D-ICs with inter-tier liquid cooling, in: Proceedings of International Conference on Computer Aided Design (ICCAD). IEEE Press, San Jose, CA, 2010, pp. 463–470.
[35] UMFPACK, ⟨http://www.cise.ufl.edu/research/sparse/umfpack/⟩.
[36] J.M. Wang, T.V. Nguyen, Extended Krylov subspace method for reduced order analysis of linear circuit with multiple sources, in: Proceedings of Design Automation Conference (DAC), 2000, pp. 247–252.
[37] J. Wang, Deterministic random walk preconditioning for power grid analysis, in: Proceedings of International Conference on Computer Aided Design (ICCAD), November 2012, pp. 392–398.
[38] M. Wang, H. Klie, et al., Solving sparse linear systems on NVIDIA Tesla GPUs, in: Proceedings of the 9th International Conference on Computational Science, 2009, pp. 864–873.
[39] S.-H. Weng, Q. Chen, N. Wong, C.-K. Cheng, Circuit simulation via matrix exponential method for stiffness handling and parallel processing, in: Proceedings of International Conference on Computer Aided Design (ICCAD), November 2012, pp. 407–414.
[40] T. Yu, Z. Xiao, M.D.F. Wong, Efficient parallel power grid analysis via additive Schwarz method, in Proceedings of International Conference on Computer Aided Design (ICCAD), 2012, pp. 399–406.
[41] L. Ziane Khodja, R. Couturier, A. Giersch, J.M. Bahi, Parallel sparse linear solver with GMRES method using minimization techniques of communications for gpu clusters, J. Supercomput. 69 (July (1)) (2014) 200–224. http://dx.doi.org/10.1007/s11227-014-1143-8.



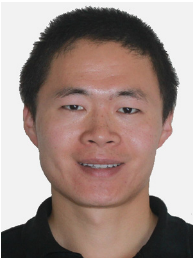
Kai He received the B.S. and M.S. degrees from Nanjing University, Nanjing, China, in 2009 and 2012. He is currently pursuing the Ph.D. degree from the Department of Electrical and Computer Engineering, University of California, Riverside, CA, USA. His current research interests include parallel computing, circuit simulation and hardware security.

Sheldon X.-D. Tan(S'96-M'99-SM'06) received his B.S. and M.S. degrees in electrical engineering from Fudan University, Shanghai, China in 1992 and 1995, respectively and the Ph.D. degree in electrical and computer engineering from the University of Iowa, Iowa City, in 1999. He is a Professor in the Department of Electrical Engineering, University of California, Riverside, CA. He is the Associate Director of Compute Engineering Program (CEN) at Bourn College of Engineering at UC Riverside since 2009. He also is a cooperative faculty member in the Department of Computer Science and Engineering at UCR. He is also a Guest Professor of Shanghai Jiao Tong University and a Guest Professor of University of Electronic Science and Technology of China.

Dr. Tan co-authored four books: Symbolic Analysis and Reduction of VLSI Circuits published by Springer/Kluwer in 2005, Advanced Model Order Reduction Techniques for VLSI Designs by Cambridge University Press published in 2007; Statistical Performance Analysis and Modeling Techniques for Nanometer VLSI Design by Springer Publishing in 2012 and Advanced Symbolic Analysis for VLSI Systems – Methods and Applications, Springer in 2014. He received Outstanding Oversea Investigator Award from the National Natural Science Foundation of China (NSFC) in 2008. He received NSF CAREER Award in 2004. He received NSF CAREER Award in 2004. Dr. Tan received the Best Paper Award from 2007 IEEE International Conference on Computer Design (ICCD'07), the Best Paper Award from 1999 IEEE/ACM Design Automation Conference. He also receives three Best Paper Award Nomination from IEEE/ACM Design Automation Conferences in 2005, 2009 and 2014 and one Best Paper Award nomination from ASPDAC in 2015. He now is serving as an Associate Editor for three journals: IEEE

Transaction on VLSI Systems (TVLSI), ACM Transaction on Design Automation of Electronic Systems (TODAE), Integration, The VLSI Journal.

Hengyang Zhao received the BS degree in computer science in 2011 and the MS degree in instrument and meter engineering in 2013, both from the Shanghai Jiao Tong University, P. R. China. He is currently studying as a Ph.D. student in department of electrical and computer engineering in University of Riverside, California. His current research interests are machine-learning based behavior modeling, fast and scale-able matrix calculations in computer-aided circuit design. His past research area includes swallowable video capsule design, image compression and FPGA-based embedded system design.



Xue-Xin Liu received his bachelor and master degrees from Fudan University, Shanghai, in 2005 and 2008 respectively, and Ph.D. degree from University of California, Riverside, in 2013. He is now with Synopsys, Inc. as senior software engineer. His research interests include device modeling, circuit simulation, and parallel computing.

Hai Wang received his B.S. degree from Huazhong University of Science and Technology, China in 2007, and his M.S. and Ph.D. degree from University of California, Riverside, USA in 2008 and 2012, respectively. After that, he has been an associate professor at University of Electronic Science and Technology of China. His research interests mainly lie in electrical/thermal verification and optimization of VLSI circuits and systems. He has published around 30 peer-reviewed international conference papers and journal articles in related research field. He serves as technical program committee member of several international conferences including DATE, ASP-DAC and ISQED, and also serves as reviewer of many journals including IEEE TC, IEEE TCAD, IEEE TCAS II and ACM TODEAS.

Guoyong Shi (S'99-M'02-SM'11) received the Bachelor's degree in applied mathematics from Fudan University, Shanghai, China, the Master of Science degree in electronics and information science from Kyoto Institute of Technology, Kyoto, Japan, and the Ph.D. degree in electrical engineering from Washington State University, Pullman, in 1987, 1997, and 2002, respectively.

He is currently a Professor with the Department of Micro/Nano-Electronics, School of Electronic, Information, and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China. Before joining the university, he worked as a post-doctor from 2002 to 2005 in the Department of Electrical Engineering, University of Washington, Seattle. He is the author or co-author of more than 60 research papers in the areas of systems, control, and integrated circuits. He is the co-author of the book “Advanced Symbolic Analysis for VLSI Systems – Methods and Applications” published by Springer in 2014. His current research interest is in the design automation of mixed-signal integrated circuits and systems and embedded systems. Dr. Shi was a co-recipient of the Donald O. Pederson Best Paper Award from the IEEE Circuits and Systems Society in 2007.