Parallel Power Grid Analysis Using Preconditioned GMRES Solver on CPU-GPU Platforms

Xue-Xin Liu*, Hai Wang†, and Sheldon X.-D. Tan*
*Department of Electrical Engineering, University of California, Riverside, CA 92521, USA
†School of Microelectronics & Solid-State Electronics, Univ. of Electronic Sci. & Tech. of China, Chengdu, 610054

***Abstract*—In this paper, we propose an efficient parallel dynamic linear solver, called *GPU-GMRES*, for transient analysis of 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 and other applications like thermal analysis. The proposed GPU-GMRES solver adopts the very general and robust incomplete LU (ILU) 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 GMRES convergence rate can be achieved for the best overall performance. Such a tunable feature makes this algorithm very adaptive to different problems. Furthermore, we properly partition 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 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. GPU-GMRES can also deliver 3–10× speedup over the CPU implementation of the same GMRES method on transient analysis.**

1. Introduction

The verification of today's large linear global networks such as on-chip large power grid networks remains challenging for chip designers. Fast verification of voltage drops and other noises on power delivery networks is critical for final design closure. The so-called power integrity verification becomes even more challenging due to increasing design complexity and lowered supply voltage as VLSI technologies advance [1]. Intensive studies 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 [2]–[6]. But most of those techniques are based on the homogeneous single-core architectures.

The recent leap from single-core to multi-core or many-core technologies has permanently altered the course of computing. Among them, the graphics processing units (GPUs), are one of the most powerful many-core computing systems in mass market use [7]. 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) and are connected by high-speed links 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 K20X chip has a peak performance of over 4 Fflops performance in comparison with about 80–100 Gflops of Intel i7 series quad-core CPUs [8]. 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.

To date, dense linear algebra support on GPU is well developed, with its own BLAS library [9], but sparse linear algebra support is still limited. In [10], a CPU based sparse LU solver was used to simulate the linear systems. However, sparse LU solvers have complicated data dependency compared to dense matrix solvers, and their implementation is considered to be difficult on GPU (although there are some recent efforts in this direction [11]). 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 [12]–[15], which confirm the practicality and effectiveness of iterative solvers in solving large linear networks like power grid networks. But the existing approaches primarily focus on the multi-core CPU. The investigation of GPU power on these iterative solvers receives less attention.

In this paper, we propose an efficient parallel dynamic linear solver with application on transient analysis of large power grid networks of VLSI systems. We aim at developing a dynamic linear solver, which is robust and can be applied to power grid networks with different structures and properties. Our new method, called *GPU-GMRES*, is based on the preconditioned generalized minimum residual (GMRES) iterative solver, and is implemented on heterogeneous CPU-GPU platforms. The GPU-GMRES solver adopts a very general and robust incomplete LU (ILU) 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 GMRES convergence rate can be made to achieve the best overall performance of the solver. Such a tunable feature can make this algorithm very adaptive and flexible for different problems. Furthermore, since many operations in the preconditioned GMRES solver, such as sparse matrix-vector multiplication (SpMV) and sparse triangular solves, 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

This work is supported in part by NSF grant under No. CCF-1017090, in part by NSF Grant under No. OISE-1130402 and a startup grant from UESTC.

978-1-4799-1071-7/13/$31.00 ©2013 IEEE

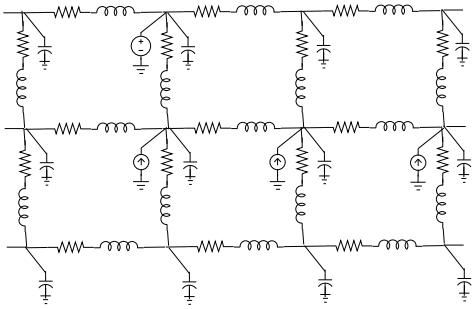


Fig. 1. An RLC model of power grid network.

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 [16]. GPU-GMRES can also deliver 3–10× speedup over the CPU implementation of the same GMRES method on transient analysis. We also show that matrix structure and property have huge impacts on the efficiency of GMRES solvers.

This paper is organized as follows. Section 2 reviews power grid analysis problem. Section 3 describes the proposed GPU-GMRES parallel algorithm and its ILU preconditioner. Several numerical examples and discussions are presented in section 4. Last, Section 5 concludes the paper.

2. The problem of power grid simulation

In this section, we review the problem of power grid analysis.

An on-chip power grid network can be modeled as RC or RLC networks with known time-variant current sources, which can be obtained by gate-level logic simulations of the circuits. A typical power grid model may contain several million nodes, and up to hundreds of thousands of input current sources. Some nodes in the grid are associated with known voltages, and are modeled as nodes connected with constant voltage sources. For C4 power grids, nodes with known voltages can be internal nodes inside the power grid. Fig. 1 shows an RLC power grid model.

Given the current source vector, $\mathbf{u}(t)$, as stimuli of the network, the node voltages can be obtained by solving the following differential equation, which is formulated using modified nodal analysis (MNA),

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

where $\mathbf{G} \in \mathbb{R}^{n\times n}$ 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.

Suppose the backward Euler method is applied to the integration of this dynamic system, the transient behavior of the power grid can be solved step by step from a given initial condition $\mathbf{x}(0)$ using

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

where h is the time step length and the new state variable $\mathbf{x}(t+h)$ is to be solved, either by direct LU method or iterative method. If a fixed time step h is chosen, then the left-hand side matrix, $\mathbf{G} + \frac{1}{h}\mathbf{C}$, will remain the same 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 $\mathbf{G} + \frac{1}{h}\mathbf{C}$, and they can be reused for all the triangular solves in the following time steps.

3. Parallel GMRES solver on the GPU-CPU platform

We have seen in the previous section that our problem is how to solve a linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{3}$$

In our application, the coefficient matrix is $\mathbf{A} = \mathbf{G} + (1/h)\mathbf{C}$, and the right-hand side vector is $\mathbf{b} = (1/h)\mathbf{C} \cdot \mathbf{x}_i + \mathbf{u}_{i+1}$. The subscript i here denotes the index of transient point, i.e., $\mathbf{x}_i = \mathbf{x}(t_i) = \mathbf{x}(i \cdot h)$. To solve linear equation like Eq. (3), one can apply the direct LU method or iterative methods. LU-factorization director solvers, however, are shown to be more difficult to be parallelized especially on GPUs, as the LU factorization has many inherent data dependency and irregular memory access. Iterative solvers, on the other hand, are more amenable for GPU computing as only sparse matrix-vector (SpMV) and triangular matrix solving (in our implementation) are required, which are GPU-friendly operations.

In this work, we investigate the GPU-accelerated GMRES iterative solver to solve the proposed power grid analysis problem. Considering the following system equivalent to $\mathbf{A}\mathbf{x} = \mathbf{b}$,

$$\mathbf{C}_{\mathrm{L}}\mathbf{A}\mathbf{C}_{\mathrm{R}}\mathbf{y} = \mathbf{C}_{\mathrm{L}}\mathbf{b}, \qquad \mathbf{y} = \mathbf{C}_{\mathrm{R}}^{-1}\mathbf{x}, \tag{4}$$

where $\mathbf{C}_{\mathrm{L}}$, $\mathbf{C}_{\mathrm{R}} \in \mathbb{R}^{n\times n}$ are non-singular. We refer $\mathbf{C}_{\mathrm{L}}$ and $\mathbf{C}_{\mathrm{R}}$ as left preconditioner and right preconditioner, respectively. The intuitive idea of preconditioning is to choose the matrices $\mathbf{C}_{\mathrm{L}}$ and $\mathbf{C}_{\mathrm{R}}$ such that $\mathbf{C}_{\mathrm{L}}\mathbf{A}\mathbf{C}_{\mathrm{R}}$ in some sense approximates the identity matrix. This is typically done by squeezing eigenvalues of $\mathbf{C}_{\mathrm{L}}\mathbf{A}\mathbf{C}_{\mathrm{R}}$ close to unity. Note that in Eq. (4) we express this preconditioning process in the form of matrix multiplication, while in practice, there can be other operations involved. For example, in our proposed solver, the two matrices $\mathbf{C}_{\mathrm{L}}$ and $\mathbf{C}_{\mathrm{R}}$ are actually the applications of lower and upper triangular solves 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 simple single-side precondition factor. Existing works have shown that simple preconditioners, e.g., diagonal (or Jacobi) preconditioner and approximate inverse preconditioner (AINV), do not have ideal preconditioning quality and they may even fail

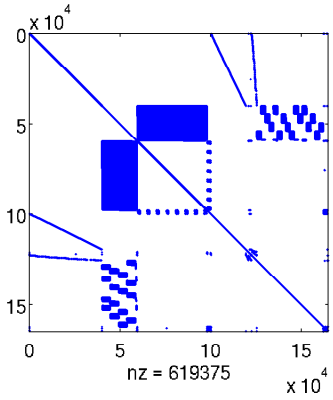


Fig. 2. Sparsity pattern of a circuit matrix from a power grid problem.

on some cases [17]. 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

$$\mathbf{C}_{\mathrm{L}}\mathbf{A}\mathbf{C}_{\mathrm{R}} = (\tilde{\mathbf{L}}^{-1}\mathbf{P})\mathbf{A}(\mathbf{Q}\tilde{\mathbf{U}}^{-1}) \tag{5}$$

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

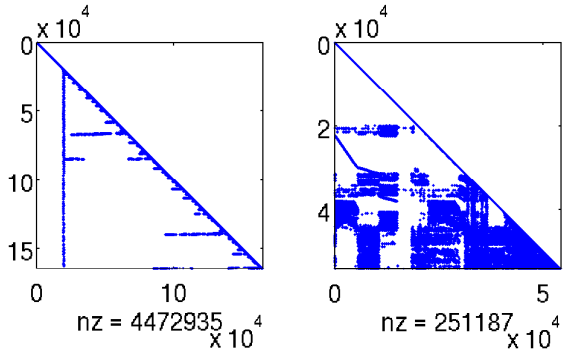
$$\mathbf{PAQ} \approx \tilde{\mathbf{L}}\tilde{\mathbf{U}},$$

which is equivalently to say that $\tilde{\mathbf{L}}^{-1}\mathbf{PAQ}\tilde{\mathbf{U}}^{-1}$ is an approximation to identity matrix $\mathbf{I}$. For a coefficient matrix generated in a power grid circuits, whose sparsity pattern is shown in Fig. 2, the sparsity patterns of its complete LU factors and ILU factors are drawn in Fig. 3. [1]

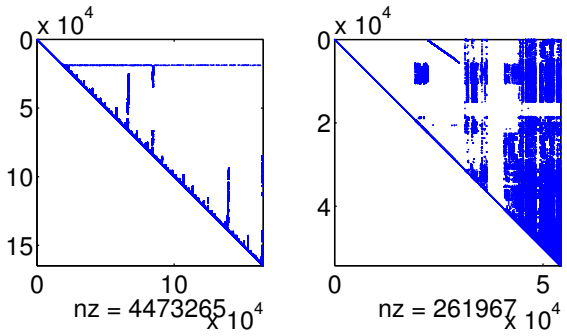
There is a critical trade-off between this approximation and the fill-in ratio of the ILU factors. An accurater approximation needs more efforts in factorization and results in high fill-in ratio of LU factors. Therefore, it will incur a high computation cost during the preconditioning process, i.e.,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

The generalized minimum residual (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

[1]Please be aware that MATLAB's plotting function makes ILU matrix patterns look more dense than the complete LU, which is not true.



(a) Lower triangular factors.



(b) Upper triangular factors.

Fig. 3. Comparison of sparsity patterns in complete and incomplete LU factors. The factors are derived from conductance matrix $\mathbf{G}$ in the example of "ibmpg1t." In both two figures, the left parts are complete LU factors, and the right parts are incomplete LU factors. Number of nonzero elements, "nz", is listed below each plot.

shows the standard Krylov-subspace based GMRES method with preconditioner [17], which uses projection method to form the m-th order Krylov-subspace [17], [18], e.g.,

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

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 $\mathbf{M}$ in Eq. (6), and from now on, all the occurrences of $\mathbf{MA}$ should denote the operation in Eq. (5), which contains two sparse triangular solves and one sparse matrix vector 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, which 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.

The least squares problem is usually solved by computing

Algorithm 1 GMRES with left and right preconditioning.

Input: $\mathbf{A} \in \mathbb{R}^{n\times n}$, $\mathbf{b} \in \mathbb{R}^n$, $\mathbf{x}_0 \in \mathbb{R}^n$ (initial guess), m (restart)

Output: $\mathbf{x} \in \mathbb{R}^n$: $\mathbf{Ax} \simeq \mathbf{b}$

1: $\mathbf{r}_0 = \mathbf{b} - \mathbf{Ax}_0$
2: $\tilde{\mathbf{r}}_0 = \mathbf{C}_\mathrm{L}\mathbf{r}_0$, $\beta = \|\tilde{\mathbf{r}}_0\|_2$, $\mathbf{v}_1 = \tilde{\mathbf{r}}_0/\beta$
3: **for** $j = 1, 2, \ldots, m$ **do** *// Arnoldi iteration on GPU*
4: $\mathbf{w} = \mathbf{C}_\mathrm{L}\mathbf{AC}_\mathrm{R}\mathbf{v}_j$ *// Eq. (5) using CUSPARSE csrmv and csrsv*
5: **for** $i = 1, 2, \ldots, j$ **do** *// using CUBLAS functions*
6: $h_{i,j} = \mathbf{w}_i^\mathrm{T}\mathbf{v}_j$
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: $\mathbf{y}_m = \mathrm{argmin}_\mathbf{y}\|\beta\mathbf{e}_1 - \tilde{\mathbf{H}}_m\mathbf{y}\|_2$
13: $\mathbf{x}_m = \mathbf{x}_0 + \mathbf{C}_\mathrm{R}\mathbf{V}_m\mathbf{y}_m$
14: **if** not converge **then**
15: $\mathbf{x}_0 = \mathbf{x}_m$, go to Line 1
16: **end if**

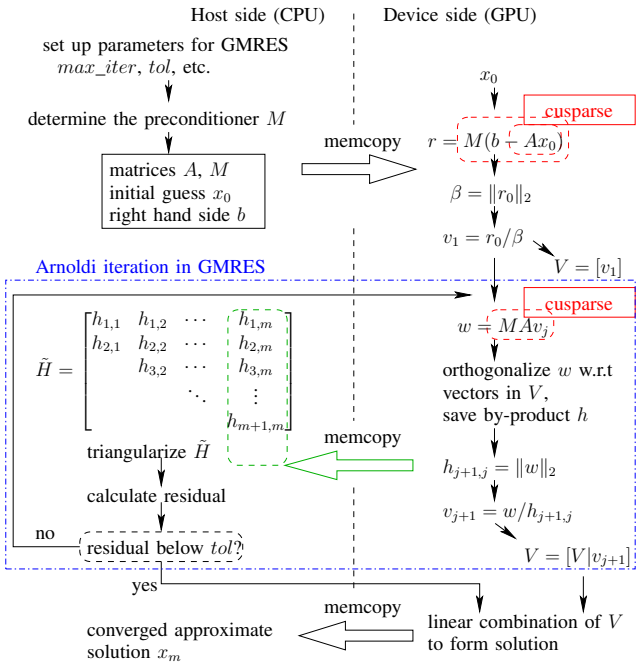


Fig. 4. The proposed GPU-accelerated parallel preconditioned GMRES solver. We also show the partitioning of the major computing tasks between CPU and GPU here.

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 implementation on parallel computers.

A. 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 sparse matrix-vector (SpMV) multiplication (`csrmv`). 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 [19]. Based on the examples we focus on, we have noticed that SpMV and triangular solves takes 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 [20].

GPU programming for many engineering problems are typically limited by the data transfer bandwidth as GPU favors computationally intensive algorithms [21]. This is especially true for operations such as sparse matrix-vector multiplication (SpMV) 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 basis vectors of the subspace $\mathbf{V}_m$ 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. Then, on CPU, 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.

B. GPU-friendly implementation of preconditioners

One important aspect of the iterative solver is the preconditioner. In this section, we discuss the implementation of ILU preconditioner for the GMRES solver on GPU platforms. We will show that proper tuning of ILU preconditioner is critical for the overall performances of the resulting GPU-accelerated GMRES solver.

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: (1) reducing the number of GMRES iterations, or attaining better convergence; and (2) accelerating the Arnoldi iteration, especially the computation in Line 4, on GPU. In order to satisfy the former requirement, 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 $\mathbf{I}$. This results in better convergence and fewer GMRES iterations, but causes more computation in Eq. (5).

To meet the latter requirement aforementioned, ILU factors with less fill-in elements are favored. When parallelizing the triangular solves of $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{U}}$ matrices on GPUs, the efficiency of the GPU solver requires less data dependency (less dependency among rows) [19]. As a result, less fill-ins benefit GPU triangular solvers [22], though they tend to hurt the convergence. Consider an extreme example, ILU0, for this case. 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.

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, so that 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, called `csrsv_solve`, 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 experimental section.

4. Experiments

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

To put our new simulator's performance into a right perspective, we compare GPU GMRES with CPU GMRES and a standard LU-based method based on UMFPACK [16]. We remark that we do not compare our 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 GPU GMRES is a general solver for any linear dynamic systems, which include but do not limit to the examples of power grid circuits and thermal circuit models. 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 24 GBytes memory available. Meanwhile, one GPU card is installed on this server, which is Tesla C2070 containing 448 cores running at 1.15 GHz and up to 5 GBytes global memory.

A. Accuracy comparison and discussions

We first test the accuracy and efficiency of our solver on the power grid circuits from IBM benchmark suite [23]. 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 I. 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 that even a loosened tolerance value, such as 10^{-6}, 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. 5 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 direct LU method and 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 error of GMRES curve, i.e., the difference between GMRES result and LU result, in Fig. 6, 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 LU golden results.

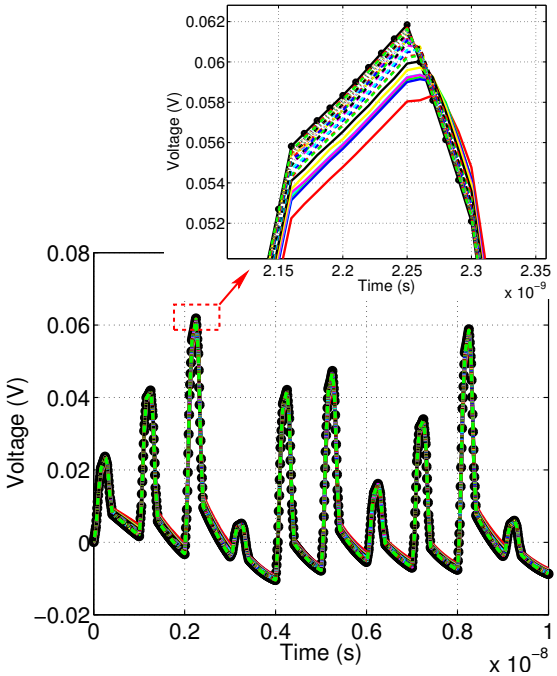


Fig. 5. Transient waveforms of LU and GPU GMRES at port node n0_5480720_1102640 in ibmpg6t. The black curve with dots is from LU direct method. All other colored curves are results of GMRES with preconditioners set to different ILU threshold, i.e., from 0.1 to 3.0.

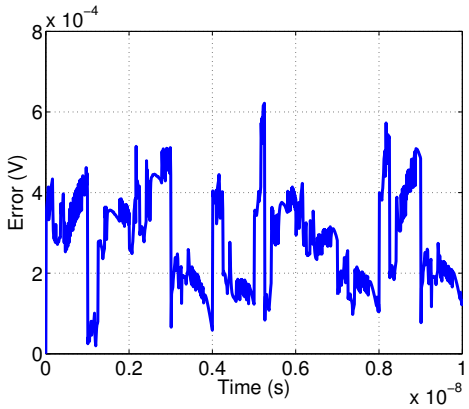


Fig. 6. The error of GPU GMRES result compared to LU golden result. This curve is calculated at node n0_5480720_1102640 of ibmpg6t, whose waveform is shown in Fig. 5.

B. Computing time comparison and discussions

Table I 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. C6 lists preconditioner setup time, C9 is for GPU GMRES solving time without initial guess available, and C12 is for GPU GMRES solving on each transient point, when good initial guess is available. The speedup of GPU GMRES over LU on DC solving is listed in C13. The speedup of GPU GMRES over LU on the whole simulation (1,000 time steps) is listed in C14, computed as $(\mathrm{C3} + 1000 \cdot \mathrm{C4})/(\mathrm{C6} + \mathrm{C9} + 1000 \cdot \mathrm{C12})$.

We first discuss the results on the IBM examples. Among the six IBM circuits, 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 solve without any good initial guess available, shall be calculated as $(\mathrm{C3} + \mathrm{C4})/(\mathrm{C6} + \mathrm{C9})$. The biggest speedup for this initial DC solving is 96 times, 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 I. 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 shows decent speedup over the direct method on these industrial design examples. We also observe that the GPU GMRES solver will have about 4–5×

TABLE I
STATISTICS OF IBM POWER GIRD BENCHMARKS AND SOLVER PERFORMANCE. COLUMN 14 LISTS THE SPEED UP OF GPU GMRES OVER LU METHOD ON ALL THE 1,000 TIME STEP POINTS IN A TRANSIENT SIMULATION CALCULATED AS $\frac{C3+1000 \cdot C4}{C6+C9+1000 \cdot C12}$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
		LU		GMRES								sp. up DC	overall
circuit	matrix	fact.	solve	ILU	precond.	solving on DC			solving on tran. step (ave.)			$\frac{C3+C4}{C6+C9}$	sp. up
name	size	(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	0.06	7	0.03	0.01	1.3	2.0
ibm2t	164,897	9.93	0.06	1.2	0.62	143	3.50	0.61	23	0.54	0.08	8.1	0.9
ibm3t	1,043,444	638.7	0.87	2.6	5.03	25	6.41	1.64	6	1.10	0.39	96	3.8
ibm4t	1,214,288	904.7	1.01	1.9	9.65	77	23.2	5.89	10	3.15	0.75	58	2.5
ibm5t	2,092,148	241.6	0.60	1.5	5.80	118	22.1	5.17	17	3.36	0.58	22	1.5
ibm6t	3,203,802	174.3	0.82	2.2	12.49	42	15.2	3.90	9	3.40	0.68	11	1.5
rlc80	32,064	6.97	0.01	1.8	0.12	29	0.12	0.34	4	0.01	0.003	15	11
rlc100	50,200	28.60	0.02	1.8	0.17	32	0.18	0.43	4	0.02	0.004	47	19
rlc120	72,384	102.2	0.05	1.9	0.26	32	0.28	0.48	4	0.02	0.01	137	11
rlc140	98,616	255.6	0.08	2.0	0.36	32	0.39	0.57	4	0.04	0.01	274	31
rlc160	128,896	726.3	0.15	2.0	0.48	34	0.51	0.64	4	0.06	0.01	648	78
rlc180	163,224	2,033.6	0.28	2.0	0.68	34	0.65	0.76	4	0.10	0.01	1410	160
rlc200	201,600	4,191.3	0.39	2.0	0.85	35	0.82	0.79	4	0.14	0.03	2555	145
rlc220	244,024	6,750.9	0.54	2.1	1.09	35	1.01	0.93	4	0.19	0.02	3213	243

speedup over their CPU version of GMRES solver on those IBM benchmark circuits, (this speedup ratio is not shown in the table), which clearly shows the advantage and benefits of GPU based computing.

For transient analysis, we observe that when the LU factors are available, it seems to be cheaper for LU triangular solve than iterative methods to compute the solution. Since fixed time step length is used in our simulator and the triangular LU factor matrices do not change, as we mentioned in previous sections, it is very understandable that GMRES does not superbly beat the direct LU solver 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 of nonlinear devices, GMRES solver will be faster than the LU factorization.

Now we discuss the results on some RLC mesh circuits, which are the last eight examples in Table I 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 is much larger (ranging from 11 to 3213) 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 GPU GMRES solver will have about 3–10$\times$ 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 GPU GMRES solver.

C. Preconditioner study and discussions

Now, let us study the quality of an ILU preconditioner. The fill-in ratio is a good indicator about the quality of ILU preconditioner. It is calculated as the ratio of the number of fill-in elements in ILU 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 drop 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 [20]. However, it has no fill-ins and does not support row/column permutation, and our experiments show that these two limitations hurt its applicability to the circuit cases here. Instead, we use the ILU++ package from [24], which 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 [25]. As we talked in Section 4-A, though low fill-in ratio implies a simple structure in the two triangular factors and a possibly faster computation in GPU's triangular solve in Eq. (5), it results in more iterations in GMRES solver and may not be optimal in term 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 II shows the relationship among the threshold, fill-in ratio, the iteration numbers, and the total GPU GMRES time. It can been seen that the CPU time reaches the minimum value when the threshold is 1.9. Fig. 7 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 in Table II). The effects of this change on fill-

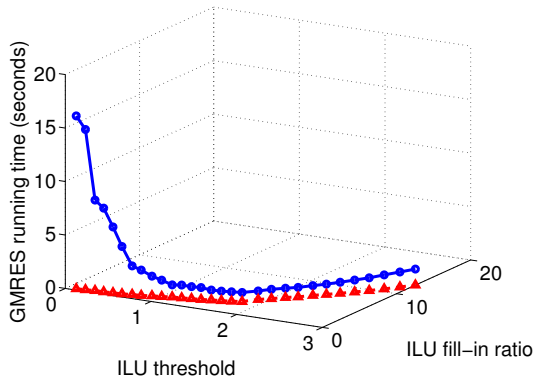


Fig. 7. The impact of ILU threshold on fill-in ratio and GPU GMRES solving time. The blue curve in 3D space is GMRES solving time with respect to threshold and fill-in ratio, and the red curve on the bottom plane reflects the changes of fill-in ratio caused by different threshold values. All the measurements are from ibmpg4t.

TABLE II
THE GPU GMRES 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	14080.5
0.3	5.69	0.53	1469	440	7288.6
0.5	5.28	0.65	690	310	5363.2
0.7	5.93	0.92	480	115	2215.8
0.9	6.18	1.29	366	68	1562.0
1.1	6.67	1.70	237	32	991.0
1.3	6.92	1.99	210	26	1014.8
1.5	7.28	2.35	126	19	857.7
1.7	7.74	2.77	109	16	820.1
1.9	9.65	4.06	77	10	749.5
2.1	12.38	5.42	47	7	815.2
2.3	16.05	6.81	39	6	866.3
2.5	20.83	8.18	30	5	992.8
2.7	27.57	9.78	37	5	1134.1
2.9	37.30	11.61	37	4	1332.6
3.0	42.68	12.55	19	4	1417.7

in ratio and GPU GMRES time on each time step are shown by two curves.

5. Conclusion

We have proposed an efficient parallel dynamic linear solver *GPU-GMRES*. The new solver is based on the preconditioned GMRES solver implemented on CPU-GPU platforms. To bring faster convergence, it adopts the very general and robust incomplete LU (ILU) preconditioner. We have shown that by properly selecting the right amount of fill-ins in the incomplete LU factors, a good trade-off between GPU efficiency and GMRES convergence rate can be achieved for the overall best performance. In addition, 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 the direct LU solver. GPU-GMRES can also deliver 3-10x speedup over the CPU implementation of the same GMRES method on transient analysis.

References

[1] International technology roadmap for semiconductors (ITRS), 2011edition, 2011. http://public.itrs.net.

[2] S. R. Nassif and J. N. Kozhaya. Fast power grid simulation. In *Proc. Design Automation Conf. (DAC)*, pages 156–161, 2000.

[3] J. M. Wang and T. V. Nguyen. Extended Krylov subspace method for reduced order analysis of linear circuit with multiple sources. In *Proc. Design Automation Conf. (DAC)*, pages 247–252, 2000.

[4] H. F. Qian, S. R. Nassif, and S. S. Sapatnekar. Random walks in a supply network. In *Proc. Design Automation Conf. (DAC)*, pages 93–98, 2003.

[5] T. Chen and C. C. Chen. Efficient large-scale power grid analysis based on preconditioned Krylov-subspace iterative method. In *Proc. Design Automation Conf. (DAC)*, pages 559–562, 2001.

[6] Y. Lee, Y. Cao, T. Chen, J. Wang, and C. Chen. HiPRIME: Hierarchical and passivity preserved interconnect macromodeling engine for RLKC power delivery. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 24(6):797–806, 2005.

[7] NVIDIA Corporation, 2011. http://www.nvidia.com.

[8] NVIDIA Tesla's Servers and Workstations. http://www.nvidia.com/object/tesla-servers.html.

[9] NVIDIA Corporation. CUBLAS library v5.0. https://developer.nvidia.com/cublas.

[10] A. M. Sridhar, A. Vincenzi, et al. 3D-ICE: Fast compact transient thermal modeling for 3D-ICs with inter-tier liquid cooling. In *Proc. Int. Conf. on Computer Aided Design (ICCAD)*, pages 463–470. IEEE Press, 2010.

[11] Ling Ren, Xiaoming Chen, Yu Wang, Chenxi Zhang, and Huazhong Yang. Sparse LU factorization for parallel circuit simulation on GPU. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1125–1130, 2012.

[12] K. Daloukas, N. Evmorfopoulos, G. Drasidis, M. Tsiampas, P. Tsompanopoulou, and G.I. Stamoulis. Fast transform-based preconditioners for large-scale power grid analysis on massively parallel architectures. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, pages 384–391, November 2012.

[13] Jia Wang. Deterministic random walk preconditioning for power grid analysis. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, pages 392–398, November 2012.

[14] Ting Yu, Zigang Xiao, and Martin D. F. Wong. Efficient parallel power grid analysis via additive schwarz method. In *ICCAD*, pages 399–406, 2012.

[15] Shih-Hung Weng, Quan Chen, Ngai Wong, and Chung-Kuan Cheng. Circuit simulation via matrix exponential method for stiffness handling and parallel processing. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, pages 407–414, November 2012.

[16] UMFPACK. http://www.cise.ufl.edu/research/sparse/umfpack/.

[17] Yousef Saad. *Iterative methods for linear systems*. PWS publishing, 2000.

[18] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. on Sci and Sta. Comp.*, pages 856–869, 1986.

[19] Maxim 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.

[20] NVIDIA Corporation. CUSPARSE library v5.0, October 2012. http://developer.nvidia.com/cuSPARSE.

[21] David B. Kirk and Wen-Mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach, 2ed*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2013.

[22] Ruipeng Li and Yousef Saad. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63(2):443–466, 2010.

[23] IBM power grid benchmarks. http://dropzone.tamu.edu/~pli/PGBench/.

[24] Jan Mayer. ILU++ package. www.iluplusplus.de/.

[25] Jan Mayer. A multilevel Crout ILU preconditioner with pivoting and row permutation. *Numer. Linear Algebra Appl.*, 14(10):771–789, December 2007.