Parallel Multigrid Preconditioning on Graphics Processing Units (GPUs) for Robust Power Grid Analysis

Zhuo Feng
Department of ECE
Michigan Technological University
Houghton, MI, 49931
zhuofeng@mtu.edu

Zhiyu Zeng
Department of ECE
Texas A&M University
College Station, TX, 77843
albertzeng@neo.tamu.edu

ABSTRACT

Leveraging the power of nowadays graphics processing units for robust power grid simulation remains a challenging task. Existing preconditioned iterative methods that require incomplete matrix factorizations can not be effectively accelerated on GPU due to its limited hardware resource as well as data parallel computing. This work presents an efficient GPU-based multigrid preconditioning algorithm for robust power grid analysis. An ELL-like sparse matrix data structure is adopted and implemented specifically for power grid analysis to assure coalesced GPU device memory access and high arithmetic intensity. By combining the fast geometrical multigrid solver with the robust Krylov-subspace iterative solver, power grid DC and transient analysis can be performed efficiently on GPU without loss of accuracy (largest errors < 0.5 mV). Unlike previous GPU-based algorithms that rely on good power grid regularities, the proposed algorithm can be applied for more general power grid structures. Experimental results show that the DC and transient analysis on GPU achieves more than $25X$ speedups over the best available CPU-based solvers. An industrial power grid with 10.5 million nodes can be accurately solved in 12 seconds.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids—*Simulation*

General Terms

Algorithms, Design, Performance, Verification

Keywords

P/G network, Multigrid, Iterative Method, GPU

1. INTRODUCTION

The design and verification of today's extremely large-scale power grid is truly challenging. Power grids are typically modeled as RC networks with up to tens of millions of nodes, which can not be easily solved by robust direct matrix solvers due to excessive runtime and memory costs.

DAC 2010, June 13-18, 2010, Anaheim, California, USA.

In recent years, a variety of research has focused on tackling large-scale power grid analysis problems [1, 2, 3], among which iterative solvers are preferred for achieving good scalabilities in runtime and memory consumption. Recently, there has been ever increasing interest in circuit simulations on data parallel computing platforms, such as graphics processing units [2, 3]. In [2], a CPU-GPU hybrid multigrid (HMD) algorithm is proposed to solve 3D irregular power grid structures, where the dominant geometrical multigrid operations are accelerated on GPU. In [3], a 2D Poisson solver using GPU-based FFT acceleration is proposed for solving 2D structured power grids. However, the above GPU algorithms are mostly suitable for solving power grids with good regularities, which can limit their applications in general power grid analysis. For instance, bad power grid designs having incorrect grid connections (such as isolated nets or shorted nets) may not be easily handled using the prior GPU-based power grid analysis algorithms.

Until recently, good progresses have been made in accelerating sparse matrix-vector operations on GPUs [4, 5], showing that more than 10G FLOPS (floating point operations per second) throughput can be achieved on the latest GPU models. However, efficient sparse matrix-vector operation on GPU does not immediately lead to faster power grid simulations if compared with the state of art direct solvers. For instance, running the conjugate gradient (CG) algorithm on GPU may easily bring 10X to 20X speedups over the CPU-based CG solver. Unfortunately, without an effective preconditioner, even accelerated on GPUs, running power grid analysis using such a plain CG solver can still be slower than using a good direct matrix solver (e.g. Cholmod [6]), especially when dealing with ill-conditioned problems. On the other hand, preconditioned conjugate gradient (PCG) algorithms using incomplete matrix factors can not be efficiently accelerated on GPUs, which is mainly due to GPU's very limited on-chip shared memory resources and data parallel computing scheme.

In this work, we propose a GPU-specific multigrid preconditioned iterative solver for robust (accurate) and efficient (fast convergence) power grid analysis. To allow efficient memory accessing and streaming data processing on GPUs, we adopt an ELL-like sparse matrix structure [7] that is particularly suitable for data-parallel power grid analysis, which enables rather efficient sparse-matrix product computation as well as the iterative relaxation operations (heavily used in the multigrid preconditioning procedure). We show that the by including multigrid preconditioning step into the Krylov-subspace iterative solver, power grid simulation can converge robustly in a few iterations, which therefore can be applied for general power grid analysis.

2. BACKGROUND

2.1 Power Grid Analysis

Power grid DC analysis for an n-node circuit can be formulated using the following linear system [1]:

$$Gx = b, \tag{1}$$

where the conductance matrix $G \in \mathbb{R}^{n\times n}$ is a symmetric positive definite (SPD) matrix representing all interconnected resistors, $x \in \mathbb{R}^{n\times 1}$ is the vector including all node voltage unknowns, while $b \in \mathbb{R}^{n\times 1}$ is an input vector including all excitation sources. The most accurate and robust way to solve such a system is to use the direct methods such as LU or Cholesky factorization algorithms, which can be rather expensive and memory inefficient for large scale circuits. Alternatively, iterative methods [1, 2] are memory efficient, thus preferred for attacking the very large power grid analysis problems.

On the other hand, transient analysis solves the dynamic system at multiple time points, taking into account energy-storage circuit components such as capacitors and inductors:

$$C\frac{dx(t)}{dt} + Gx(t) = b(t). \tag{2}$$

After applying the backward Euler's (BE) method, we obtain an alternative linear system equation for time step t:

$$\left(\frac{C}{h} + G\right) x(t) = b(t) + \frac{C}{h} x(t-h), \tag{3}$$

where h denotes the time step size. If direct methods are used, one-time matrix factorization is performed, while the solution of the following time steps can be obtained by reusing the matrix factors.

2.2 Basics of GPU Computing

Nowadays GPU computing for non-graphics applications becomes increasingly popular due to the significant performance boosts brought by the latest technologies. A recent GPU model integrates more than 200 streaming processors (SP) onto a single chip, achieving more than 700G FLOPS peak performance, while greater than 100Gb/s off-chip memory bandwidth can be observed. The above performance is much higher than the ones from the best available general purpose processors such as the quad-core CPUs. However, GPU's on-chip memory resource limitation as well as data parallel computing scheme make its programming rather challenging. We compare different memory resources of typical GPUs in Fig. 1, where the texture memory refers to the cached global memory that may lead to better memory access latency than the global memory access. To improve GPU computing efficiency, the following key factors should be considered:

1. *Data decomposition and sharing*: A GPU kernel should perform similar operations on different sets of data, minimize the dependencies among different tasks, avoid excessive global data sharing, use more read-only data sharing than the read-write data sharing to minimize synchronization times, avoid shared memory bank conflicts, and maximize the arithmetic intensity (defined as the total GPU computations per memory access).

2. *Control Flow*: A GPU algorithm should avoid thread branching (divergence), allow efficient parallel implementation that usually exploits formulating the original problems into multi-level hierarchical problems, and assure coalesced memory access.

	Local	Shared	Global	Texture
Read	Yes	Yes	Yes	Yes
Write	Yes	Yes	Yes	No
Size	Large	Small	Large	Large
BW	High	High	High	High
Cached?	No	Yes	No	Yes
Latency	500 cyc.	20 cyc.	500 cyc.	300 cyc.

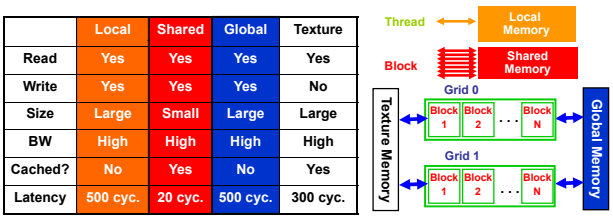

Figure 1: Device (GPU) Memory comparisons. "cyc." denotes the clock cycles.

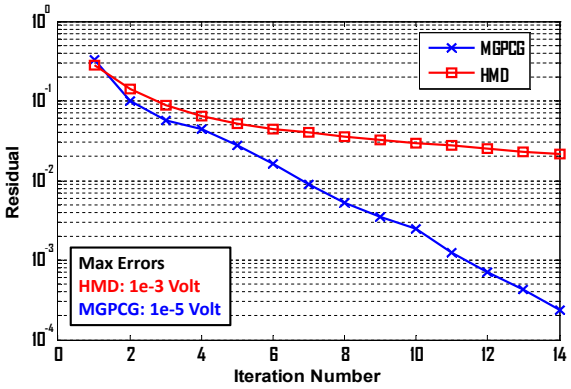

Figure 2: Convergence rate comparison of hybrid multigrid (HMD) [2] and multigrid preconditioned conjugate gradient methods (MGPCG).

2.3 Power Grid Analysis On GPU

2.3.1 Prior Approaches

Efficient power grid analysis on GPU requires good handling of irregular power grid data structures. As mentioned in Section 2.2, coalesced memory accessing and performing similar operations on different data sets are necessary for improving GPU's computing efficiency and minimizing the divergent threads. However, realistic power grid designs can be irregular and the resultant sparse matrix structures may not be regular too, so it is important to represent the irregular grid structure using regular data structures on GPU for improving the overall computing performance. Prior works on GPU-based power grid analysis [2, 3] exploit regular or regularized power grid structures for efficient GPU computation: a hybrid multigrid scheme is proposed in [2] to solve coarse level regular grid problem on GPU and correct the original irregular grid solution on CPU, while the work in [3] can be only applied to structured 2D power grid problems.

2.3.2 Proposed Approach

Although traditional Krylov-subspace iterative methods such as conjugate gradient method are very robust (always converge to the correct solution) and suitable for parallel computing, the number of CG iterations for power grid analysis can be still high, which may limit its application to large circuit problems. On the other hand, power grid analysis using existing preconditioned conjugate gradient solver [1] with incomplete matrix factors as preconditioners can converge faster in a few hundreds iterations. However the matrix factors that include many non-zeros elements can not be efficiently stored and processed on GPU's shared memory (shared memory size is up to 32KB per streaming multipro-

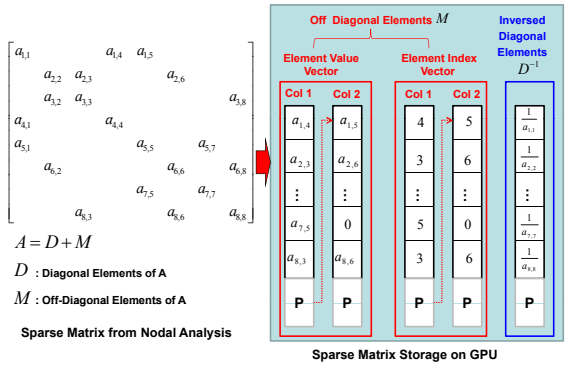

Figure 3: Sparse matrix storage for GPU-based power grid analysis. "P" are the padded dummy elements to make sure the vector dimension is a multiple of 16.

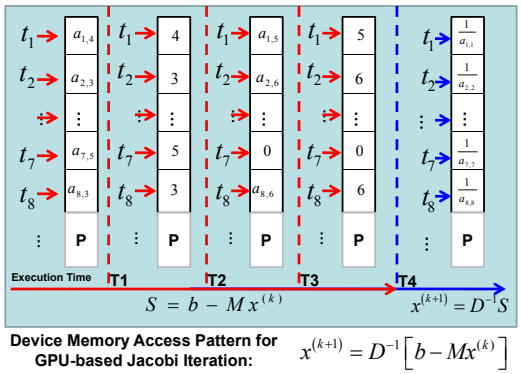

Figure 4: Matrix data access pattern during the Jacobi iterations on GPU. t_i denotes the GPU threads that are accessing the GPU device memory.

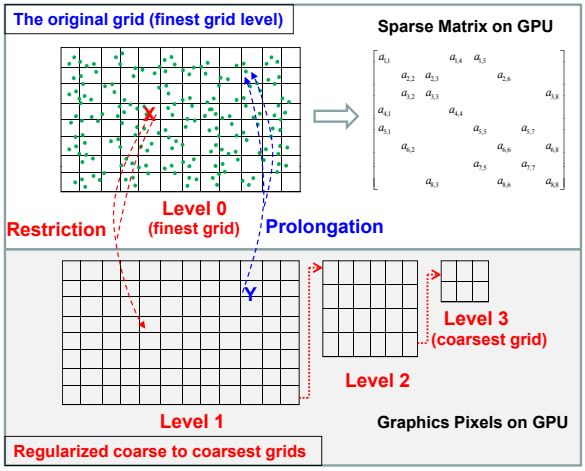

Figure 5: GPU memory layout for multigrid preconditioning.

cessor), which is therefore not suitable for GPU acceleration.

To gain high parallel computing efficiency and accuracy, a multigrid-preconditioned conjugate gradient method (MGPCG) has been proposed for groundwater flow simulation problems (elliptic PDE equations) [8], which outperforms the plain multigrid solver in terms of convergence rate and accuracy. Since multigrid methods have been shown to be well suited for GPU accelerations [2], in this work we propose a GPU-based MGPCG algorithm specifically for robust power grid analysis, and show that our algorithm can be efficiently accelerated on GPU. More importantly, this preconditioned iterative solver is more robust than the prior hybrid multigrid algorithm (HMD) [2], which retains near-SPICE accuracy after only a few iterations. In Fig. 2, we show the DC analysis results for an industrial power grid design with 1.6 million nodes. The residual versus the number of iterations are shown for both the MGPCG and HMD solvers. From the figure, it can be observed that the MGPCG solver converges to highly accurate solution in a much faster way, retaining two orders of magnitude better accuracy when compared with the HMD solver.

3. MGPCG SOLVER ON GPU

3.1 Sparse Matrix Operations on GPU

Each of the power grid nodes typically connects to a limited number of neighboring nodes (a node usually has no more than four neighbors). It is also observed that most of these grid nodes have similar numbers of neighboring nodes. The above observations allow us to efficiently store the sparse conductance matrix G that is obtained from power grid nodal analysis using an ELL-like [7] sparse matrix structure. As shown in Fig. 3, once given the maximum degree of freedom of all the nodes, we can use three one-dimensional vectors to fully represent the original conductance matrix: one vector stores the inverse of diagonal matrix elements, one stores the off-diagonal element values and the rest stores the neighboring node indices. With the above sparse matrix structure, matrix-vector operations can be performed rather efficiently on GPU with very good memory access pattern [4]. Since GPU memory prefers the vector sizes to be multiple of 16, we add dummy elements to the ends of all the vectors (represented by "P" in Fig. 3 and Fig. 4). We demonstrate the GPU device memory access patterns during Jacobi iterations in Fig. 4. As shown, at different kernel execution steps (time points), consecutive GPU threads will access consecutive memory addresses, which exactly follows the coalesced memory access principle (Section 2.2). We store the inverse of diagonal elements instead of the original diagonal elements, since the multiplication operation on GPU is less expensive than the division operation (used heavily in GPU multigrid preconditioning). It is observed in our experiments that using the above data structure, up to 3X better performance (throughput) can be achieved when compared with the implementations using other sparse matrix formats such as the compressed sparse row (CSR) format.

3.2 Multigrid Preconditioning on GPU

As demonstrated in Fig. 5, the basic idea behind the multigrid preconditioning method is to use the original irregular power grid as the finest grid (level 0) in the multigrid hierarchy, and adopt a set of regularized 2D grids as the coarser to coarsest level grids (level 1 to level 3). The regularized 2D grids can be obtained in several ways such as the ones introduced in [2]. Then the finest grid (level 0 grid) will be handled using efficient sparse matrix structure on GPU (proposed in Section 3.1) while the 2D coarser to coarsest grids (level 1 to level 3 grids) are handled directly as graph-

Algorithm 1 Multigrid Preconditioning Algorithm

Input: The sparse conductance matrix $\mathbf{G_f} \in \mathbb{R}^{\mathbf{n \times n}}$ for the original power grid ($\mathbf{Grid_f}$ with n nodes), the coarse grid $\mathbf{Grid_c}$ with m nodes, the coarse-to-fine grid prolongation operator $V_c^f \in \mathbb{R}^{n \times m}$, the fine-to-coarse grid restriction operator $V_f^c \in \mathbb{R}^{m \times n}$, the coarse grid geometrical multigrid solver [2] that computes the solution $\mathbf{x_c} = mgsolve(\mathbf{f_c})$ where the input and output vectors are $\mathbf{f_c}$ and $\mathbf{x_c} \in \mathbb{R}^{m \times 1}$, the original grid rhs vector $\mathbf{b_f} \in \mathbb{R}^{\mathbf{n \times 1}}$, the initial solution guess $\mathbf{x_f^{(0)}} \in \mathbb{R}^{\mathbf{n \times 1}}$, the maximum number of iterations $\mathbf{K}$, the number of weighted Jacobi iterations $\mathbf{s}$, as well as the error tolerance **tol**.

Output: The solution $\mathbf{x_f}$ for all grid nodes of $\mathbf{Grid_f}$.

Do $\mathbf{s}$ times relaxations on $\mathbf{Grid_f}$ to get $\mathbf{x_f^{(0)}}$;
Calculate the $\mathbf{r_f^{(0)}} = b_f - G_f x_f^{(0)}$;
for $(\mathbf{k = 0; k < K; k++})$: **do**
  $\mathbf{r_c^{(k)}} = \mathbf{V_f^c r_f^{(k)}}$;
  $\mathbf{e_c}^{(k)} = mgsolve(\mathbf{r_c^{(k)}})$;
  $\mathbf{e_f^{(k)}} = \mathbf{V_c^f e_c^{(k)}}$;
  $\mathbf{x_f}^{(k+1)} = \mathbf{x_f}^{(k)} + \mathbf{e_f}^{(k)}$;
  Do $\mathbf{s}$ times post relaxations on $\mathbf{Grid_f}$ to update $\mathbf{x_f^{(k+1)}}$;
  $\mathbf{r_f^{(k+1)}} = b_f - G_f x_f^{(k+1)}$;
  if $||\mathbf{r_f^{(k+1)}}|| < \mathbf{tol}$ **then**
    exit the loop and return the solution $\mathbf{x_f^{(k+1)}}$;
  end if
end for
Return the solution $\mathbf{x_f} = \mathbf{x_f^{(k+1)}}$;

ics pixels on GPU. In this way, we can eliminate most of the inefficient device memory access operations, avoid thread branchings, minimize the global data dependencies and increase the arithmetic intensity of multigrid operations for all the multigrid levels. The level 0 multigrid operations are performed using GPU-based sparse matrix-vector operations (Fig. 4), while the other multigrid operations (level 1 to level 3) are performed in a geometrical multigrid fashion (treat each grid node as a graphics pixel) that can achieve very high throughputs on GPU(over 100G FLOPS performance).

Denoting the finest power grid (level 0 irregular 3D grid in Fig. 5) by $Grid_f$ and the next coarser grid (level 1 regular 2D grid in Fig. 5) by $Grid_c$, the multigrid preconditioning algorithm is described in Algorithm 1. It should be noted that there can be many configurations (controlling parameters) for the multigrid preconditioning step. The most important controlling parameters include the number of Jacobi relaxations for the original grid s and the number of V cycles for the multigrid solver *mgsolve*. For different power grid analysis problems, the user can adjust these key parameters empirically. It is also expected that we automatically extract the optimal controlling parameters by using GPU performance modeling and optimization approaches. Due to the scope of this paper, we will not discuss the details about the parameter setting issue.

It should be emphasized that developing an algebraic multigrid (AMG) preconditioner on GPU is impractical due to the significant unbalanced workload during the computation. Another limiting factor is that the sparse matrix structures for different AMG levels can change significantly, which does not allow to use the ELL-like matrix format to gain efficiency. On comparison, the workload of the proposed preconditioning algorithm can be well balanced among different

Algorithm 2 Multigrid preconditioned CG (MGPCG) algorithm

Input: The sparse conductance matrix $\mathbf{G} \in \mathbb{R}^{\mathbf{n \times n}}$ for the original power grid with n nodes, the multigrid preconditioner that computes the approximate solution $\mathbf{x} = MGPrecond(\mathbf{b})$ (Algorithm 1) where the input and output vectors are $\mathbf{b}$ and $\mathbf{x} \in \mathbb{R}^{n \times 1}$, the rhs vector $\mathbf{b} \in \mathbb{R}^{\mathbf{n \times 1}}$, the maximum number of iterations $\mathbf{K}$, as well as the error tolerance **tol**.
Output: The solution for all grid nodes.

$\mathbf{x^{(0)}} = MGPrecond(\mathbf{b})$;
$\mathbf{r^{(0)}} = \mathbf{b} - \mathbf{G x^{(0)}}$;
$\mathbf{z^{(0)}} = MGPrecond(\mathbf{r^{(0)}})$;
$\mathbf{q^{(0)}} = \mathbf{z^{(0)}}$;
for $(\mathbf{k = 0; k < K; k++})$: **do**
  $\alpha_k = \frac{\mathbf{r^{(k)}}^T \mathbf{z^{(k)}}}{\mathbf{q^{(k)}}^T \mathbf{G q^{(k)}}}$;
  $\mathbf{x^{(k+1)}} = \mathbf{x^{(k)}} + \alpha_k \mathbf{q^{(k)}}$;
  $\mathbf{r^{(k+1)}} = \mathbf{r^{(k)}} - \alpha_k \mathbf{G q^{(k)}}$;
  if $\mathbf{r^{(k+1)}} < \mathbf{tol}$ **then**
    exit the loop and return the solution $\mathbf{x^{(k+1)}}$;
  end if
  $\mathbf{z^{(k+1)}} = MGPrecond(\mathbf{r^{(k+1)}})$;
  $\beta_k = \frac{\mathbf{r^{(k+1)}}^T \mathbf{z^{(k+1)}}}{\mathbf{r^{(k)}}^T \mathbf{z^{(k)}}}$;
  $\mathbf{q^{(k+1)}} = \mathbf{z^{(k+1)}} + \beta_k \mathbf{q^{(k)}}$;
end for
Return the solution $\mathbf{x^{(k+1)}}$.

GPU cores. Our algorithm does not use sparse matrices during the coarse grid operations.

3.3 The MGPCG Algorithm

A preconditioned conjugate gradient method that uses incomplete matrix factors as preconditioners to improve the convergence rate of CG is proposed in [1]. While the incomplete Cholesky factorization method has shown potentials of providing good preconditioners for power grid analysis, such preconditioning technique may not be suitable for GPU-based parallel computation, since there is not enough memory space on GPU for storing and processing the matrix factors (preconditioners). Additionally, the irregular matrix factors may cause excessive random memory accesses on GPU. Instead of using the "black-box" incomplete factorization methods, it has been shown in [8], by combining the faster but less robust multigrid solver with the slower but more robust conjugate gradient method, a more robust and highly parallelizable power grid solver can be created.

In this work, we propose a multigrid preconditioned conjugate gradient (MGPCG) solver on GPU for fast and robust power grid analysis. More specifically, we do not form an explicit preconditioner but rather use a GPU-based multigrid solver as an implicit preconditioner. As shown in our experiments, with such a GPU-accelerated multigrid preconditioner, power grid analysis requires significantly less number of iterations for converging to a satisfactory accuracy level. The number of required MGPCG iterations is much smaller than the traditional conjugate gradient iteration method as well as the hybrid multigrid iteration method [2]. The multigrid preconditioned conjugate gradient algorithm has been described in Algorithm 2 with more details. From our extensive experiments, it is observed that in most cases, the MGPCG solver can converge in 10 iterations (largest errors < 0.5 mV). It can be expected that if mixed precision multigrid algorithms [9] are adopted for the MGPCG solver, further convergence improvement can be made (GPU based solver mainly uses single-precision computations).

Table 1: Power grid circuit details. N_{node} is the number of nodes, N_{lay} is number of metal layers, N_{nz} is number of non-zeros of the conductance matrix, N_{res} is the number of resistors,N_{cur} is the number of current sources.

CKT	N_{node}	N_{lay}	N_{nz}	N_{res}	N_{cur}
$CKT1$	$127K$	5	$542.9K$	$209.7K$	$37.9K$
$CKT2$	$851.6K$	5	$3.7M$	$1.4M$	$201.1K$
$CKT3$	$953.6K$	6	$4.1M$	$1.5M$	$277.0K$
$CKT4$	$1.0M$	3	$4.3M$	$1.7M$	$540.8K$
$CKT5$	$1.7M$	3	$6.6M$	$2.5M$	$761.5K$
$CKT6$	$4.7M$	8	$18.8M$	$6.8M$	$185.5K$
$CKT7$	$6.7M$	8	$26.2M$	$9.7M$	$267.3K$
$CKT8$	$10.5M$	8	$40.8M$	$14.8M$	$419.3K$

The cost of each MGPCG iteration mainly comes from the multigrid preconditioning step. After analyzing the runtime of the multigrid preconditioning step, we found that the *mgsolve* time is similar to the *Jacobi* relaxation time. Consequently, we only need to optimize these two GPU kernels to achieve better performance.

4. EXPERIMENTAL RESULTS

Various experiments have been conducted to validate the proposed GPU-based multigrid preconditioned conjugate gradient (MGPCG) algorithm. A set of industrial power grids [10] (details are shown in Table 1) are used to test our MGPCG power grid solver. GPU-based MGPCG results are compared with several other GPU-based and CPU-based solvers: the GPU-based conjugate gradient (CG) solver, the GPU-based diagonal preconditioned conjugate gradient (DPCG) solver, the GPU-based hybrid multigrid (HMD) solver [2], and the CPU-based direct matrix solver CHOLMOD (Cholesky factorizations) [6]. All algorithms have been implemented using C++ and the GPU programming interface CUDA [11]. The hardware platform is a Linux PC with Intel Core 2 Quad CPU running at 2.66 GHz clock frequency and an NVIDIA's Geforce GTX 285 GPU (with 240 streaming processors). All running time results are measured in seconds.

4.1 Power Grid DC Analysis

We demonstrate the effectiveness of the multigrid preconditioned conjugate gradient (MGPCG) method proposed in Algorithm 2. In our experiments as shown in Table 2, we assure that the largest errors for all circuits are much smaller than 0.5 mV when applying the traditional CG method, the DPCG method, the MGPCG and the HMD methods. A significant reduction in the number of iterations are observed on all power grid designs when using the multigrid preconditioning technique. To obtain highly accurate results, the GPU-based MGPCG solver takes much less computation time than the GPU-based hybrid multigrid solver for all test cases. More importantly, our MGPCG method is more robust than the previous GPU-based algorithms. As shown in Table 2, our MGPCG solver can achieve very fast convergence (five to ten iterations), while the previous HMD solver can not converge to the desired accuracy in 30 iterations for CKT4 and CKT5.

From the experiments we can also find that the numbers of MGPCG iterations are almost constant, not varying significantly with circuit size. On the other hand, for larger circuits such as CKT4 and CKT5, the traditional CG solver and the diagonal preconditioned CG (DPCG) solver may take 500X to 1000X more iterations to converge.

It should be emphasized that the multigrid preconditioning technique can be efficiently implemented for the massively parallel computing platforms such as GPUs, while the other preconditioning techniques that are based on incomplete matrix factorizations can not be implemented for the GPU computing platform.

Table 3: 100 steps of transient analysis results of GPU-based MGPCG method (Algorithm 2). T_{GPU} is runtime of the GPU-based MGPCG solver, while T_{CPU} is the runtime of the direct solver based on the Cholesky factorization method [6]. N_{GPU} is the total number of the GPU-based MGPCG iterations. E_{avg} (E_{max}) is average (max) error of the GPU-based MGPCG solver. *Speedup* is the runtime ratio T_{GPU}/T_{CPU}.

CKT	T_{CPU}	T_{GPU}	N_{GPU}	E_{avg}	E_{max}	*Speedup*
$CKT1$	40.8	2.1	102	$3e-6$	$8e-4$	$20X$
$CKT2$	315.3	15.2	99	$1e-5$	$3e-4$	$21X$
$CKT3$	360.7	15.6	100	$5e-6$	$1e-4$	$23X$
$CKT4$	352.7	19.6	140	$3e-5$	$2e-4$	$18X$
$CKT5$	553.9	26.1	130	$2e-5$	$1e-4$	$21X$

Table 4: DC analysis for large power grids. N_{MGPCG} is the number of MGPCG iterations, T_{MGPCG} is the MGPCG runtime, and T_{CHOL} is the direct solver runtime. *Speedup* is the runtime ratio T_{CHOL}/T_{MGPCG}. Runtime results are shown in seconds.

CKT	N_{node}	N_{MGPCG}	T_{MGPCG}	T_{CHOL}	*Speedup*
$CKT6$	$4.7M$	7	4.9	131.5	$27X$
$CKT7$	$6.7M$	9	7.9	205.1	$26X$
$CKT8$	$10.5M$	11	11.6	N/A	N/A

4.2 Power Grid Transient Analysis

In this work, we use typical gate/decoupling capacitance values [12] in the power grid transient analysis. Loading current waveforms are modeled using the pulse-like functions while the switching activities are randomly assigned for different circuit blocks.

While the MGPCG solver for DC problems uses 30 to 50 Jacobi iterations for the smoothing step (Algorithm 1), in transient analysis, we only use 10 Jacobi iterations in each preconditioning step to reduce the simulation cost. We run the transient analysis using the MGPCG algorithm and assure it converges to the SPICE accuracy level at each time step. As shown in Table 3, the average MGPCG iteration number for each time step is between one and two. For the experiments with various current loading settings, we observe that the MGPCG solver always converges fast, using only one or two MGPCG iterations. Comparing the transient analysis results with the best available direct solver (CHOLMOD) solver results, GPU-based MGPCG solver can achieve up to 23X speedups, while retaining near-SPICE accuracy. The transient analysis waveforms for CKT5 is shown in Fig. 6.

4.3 DC Analysis for Large Power Grids

The proposed MGPCG algorithm can solve more general power grid problems. In this section, we test our algorithm on a variety of large scale power grid designs. Some of the designs do not have regular grid patterns (partial grids extracted from multi-layer power grid designs), and some

Table 2: DC analysis results of GPU-based MGPCG method (Algorithm 2). $N_{CG}(T_{CG})$, $N_{DPCG}(T_{DPCG})$, $N_{MGPCG}(T_{MGPCG})$, and $N_{HMD}(T_{HMD})$ are the numbers of iterations (runtime) using the GPU-based CG method, the GPU-based DPCG method, the GPU-based MGPCG method, and the GPU-based HMD method [2], respectively. T_{CHOL} is the runtime of the direct solver based on the CPU-based Cholesky factorization method [6]. E_{avg} (E_{max}) is average (max) errors of the GPU-based MGPCG method. *Speedup* is the runtime ratio T_{MGPCG}/T_{CHOL}.

CKT	N_{CG}	N_{DPCG}	N_{MGPCG}	N_{HMD}	T_{CG}	T_{DPCG}	T_{MGPCG}	T_{HMD}	T_{CHOL}	E_{avg}	E_{max}	*Speedup*
$CKT1$	$1,405$	400	3	7	0.2	0.12	0.05	0.1	1.7	$1e-4$	$4e-4$	$34X$
$CKT2$	$4,834$	$3,351$	4	13	5.9	3.9	0.5	0.78	20.2	$2e-6$	$2e-5$	$40X$
$CKT3$	$2,253$	681	3	8	5.4	2.1	0.4	0.72	21.6	$1e-5$	$1e-4$	$54X$
$CKT4$	$4,062$	$2,411$	5	> 30	5.3	3.7	0.9	> 2.4	19.4	$8e-5$	$7e-4$	$22X$
$CKT5$	$6,433$	$3,700$	6	> 30	15.2	9.5	1.1	> 4.5	25.6	$1e-4$	$5e-4$	$25X$

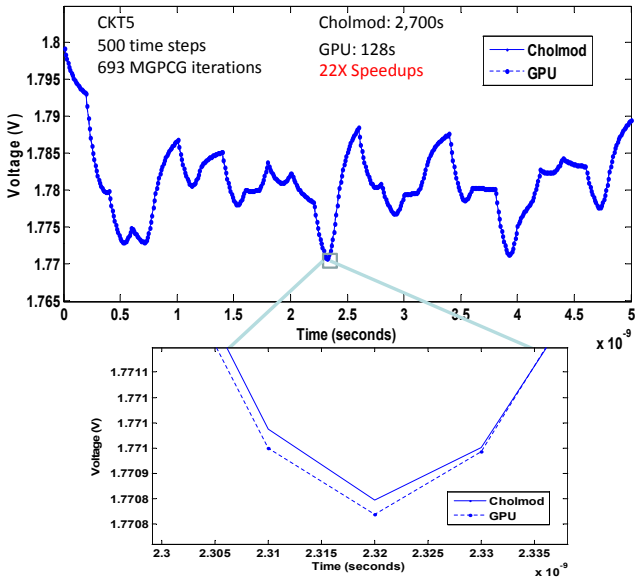

Figure 6: MGPCG based power grid transient simulation on GPU.

grids are not correctly connected (with floating or disconnected nets). The circuit size ranges from 4.7 million to 10.5 million. These test cases can not be handled by prior GPU based power grid analysis algorithms, but can be easily solved using the proposed method. The maximum errors of our iterative solver are much smaller than $0.5mV$. Details of all experiments are demonstrated in Table 4. As observed, even for these "bad" power grid designs with multi-million nodes, the proposed algorithm can scale well and achieve up to $28X$ speedups when compared with the state of art direct solver.

5. CONCLUSIONS

We propose an efficient GPU-based multigrid preconditioning algorithm for robust power grid analysis. An ELL-like sparse matrix data structure is adopted and implemented specifically for power grid analysis to assure coalesced GPU device memory access and good arithmetic intensity. By integrating the fast geometrical multigrid preconditioning step into the robust Krylov-subspace iterative algorithm, power grid DC and transient analysis can be performed efficiently on GPU without loss of accuracy. Unlike previous GPU-based algorithms that rely on good power grid regularities, the proposed algorithm can be applied for more general power grid structures. Extensive experimental results show that we can achieve more than $25X$ speedups over the best available CPU-based solvers for DC and transient power grid simulations. A 10.5 million industrial power grid DC analysis problem can be solved in 12 seconds.

6. REFERENCES

[1] T. H. Chen and C. C.-P Chen. Efficient large-scale power grid analysis based on preconditioned Krylov-subspace iterative methods. In *Proc. IEEE/ACM DAC*, pages 559–562, 2001.

[2] Z. Feng and P. Li. Multigrid on GPU: tackling power grid analysis on parallel SIMT platforms. In *Proc. IEEE/ACM ICCAD*, pages 647–654, 2008.

[3] J. Shi, Y. Cai, W. Hou, L. Ma, S. Tan, P. Ho, and X. Wang. GPU friendly fast Poisson solver for structured power grid network analysis. In *Proc. IEEE/ACM DAC*, pages 178–183, 2009.

[4] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on cuda. *NVIDIA Technical Report NVR-2008-004*, December 2008.

[5] Y. Deng, B. Wang, and S. Mu. Taming Irregular EDA Applications on GPUs. In *Proc. IEEE/ACM ICCAD*, pages 539–546, 2009.

[6] T. Davis. *CHOLMOD: sparse supernodal Cholesky factorization and update/downdate*. [Online]. Available: http://www.cise.ufl.edu/research/sparse/cholmod/, 2008.

[7] D. Kincaid and D. Young. The ITPACK Project: Past, Present, and Future. *Report CNA-180, Center for Numerical Analysis, University of Texas, Austin*, Mar. 1983.

[8] S. F. Ashby and R. D. Falgout. A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations. In *Nuclear Science and Engineering 124*, pages 145–159, 1996.

[9] D. Göddeke and R. Strzodka. Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid. *accepted to IEEE Trans. on Para. and Dist. Syst.*, Mar. 2010.

[10] S. R. Nassif. *IBM power grid benchmarks*. [Online]. Available: http://dropzone.tamu.edu/ pli/PGBench/, 2008.

[11] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*. [Online]. Available: http://www.nvidia.com/object/cuda.html, 2007.

[12] L. Smith, R. Anderson, and T. Roy. Chip-Package Resonance in Core Power Supply Structures for a High Power Microprocessor. In *Proceedings of IPACK*, 2001.