Parallel On-Chip Power Distribution Network Analysis on Multi-Core-Multi-GPU Platforms

Zhuo Feng, *Member, IEEE*, Zhiyu Zeng, *Student Member, IEEE*, and Peng Li, *Senior Member, IEEE*

***Abstract*—The challenging task of analyzing on-chip power (ground) distribution networks with multimillion node complexity and beyond is key to today's large chip designs. For the first time, we show how to exploit recent massively parallel single-instruction multiple-thread (SIMT)-based graphics processing unit (GPU) platforms to tackle large-scale power grid analysis with promising performance. Several key enablers including GPU-specific algorithm design, circuit topology transformation, workload partitioning, performance tuning are embodied in our GPU-accelerated hybrid multigrid (HMD) algorithm (GpuHMD) and its implementation. We also demonstrate that using the HMD solver as a preconditioner, the conjugate gradient solver can converge much faster to the true solution with good robustness. Extensive experiments on industrial and synthetic benchmarks have shown that for DC power grid analysis using one GPU, the proposed simulation engine achieves up to 100× runtime speedup over a state-of-the-art direct solver and more than 50× speedup over the CPU based multigrid implementation, while utilizing a four-core-four-GPU system, a grid with eight million nodes can be solved within about 1 s. It is observed that the proposed approach scales favorably with the circuit complexity, at a rate about 1 s per two million nodes on a single GPU card.**

***Index Terms*—Circuit simulation, graphics processing units (GPUs), interconnect modeling, multigrid method, parallel computing, power grid simulation, preconditioner.**

I. Introduction

THE sheer size of present day power/ground distribution networks makes their analysis and verification extremely runtime and memory consuming, and at the same time, limits the extent to which these networks can be optimized. In the past decade, on the standard general-purpose central processing unit (CPU) platform, a body of power grid analysis methods have been proposed [8], [14], [22], [19], [23], [17], [24], [20] with various tradeoffs. Recently, the emergence of massively parallel single-instruction multiple-data (SIMD), or more precisely, single-instruction multiple-thread (SIMT) [1], based graphics processing unit (GPU) platforms offers a promising opportunity to address the challenges in large scale power grid analysis. Today's commodity GPUs can deliver more than 380 GLOPS of theoretical computing power and 86 GB/s off-chip memory bandwidth, which are 3–4× greater than offered by modern day general-purpose quad-core microprocessors [1]. Moreover, the number of transistors integrated on GPU is doubling every year, which already exceeds the Moore's Law. The ongoing GPU performance scaling trend justifies the development of a suitable subset of computer-aided design (CAD) applications on such platform.

However, converting the impressive theoretical GPU computing power to usable design productivity can be rather nontrivial [11], [15], [10]. Deeply rooted in graphics applications, the GPU architecture is designed to deliver high-performance for data parallel computing. Except for straightforward general-purpose SIMD tasks such as parallel table lookups, rethinking and reengineering are required to express the data parallelism hidden in an application in a suitable form to be exploited on GPU. For power grid analysis, the above goal is achieved by the proposed GPU-accelerated hybrid multigrid (HMD) algorithm and its implementation via a careful interplay between algorithm design and SIMT architecture consideration. Such interplay is essential in the sense that it makes it possible to balance between computing and memory access, reduce random memory access patterns and simplify flow control, key to efficient GPU computing. To the best of our knowledge, our HMD solver is the first reported GPU-based power grid analysis tool.

As shown in Fig. 1, our HMD solver is built upon a custom multigrid (MG) algorithm as opposed to a direct solver. Despite the attempts to develop general-purpose direct matrix solvers on GPUs [12], so far the progress has been limited for large sparse problems due to the very nature of GPU such as the inefficiency in handling random complex data structures and memory access. Being a multilevel iterative numerical method, multigrid naturally provides a divide-and-conquer based solution that meets the stringent on-chip shared memory constraint in GPU. To further enhance the efficiency of our GPU-based multigrid algorithm, we keep the original 3-D irregular power grid as the finest grid, while for the next coarser grid, we propose a topology regularization step to obtain a regularized 2-D grid structure that is based on the irregular grid topology. Multigrid algorithm that is applied to the coarse grids can very well fit onto the GPU computing platform, in that the 2-D grid structure can lead to significantly reduced random memory access and thread divergence that are critical for high performance GPU computing. With our coarse grid construction, block smoothing

Manuscript received September 15, 2009; revised February 02, 2010; accepted June 20, 2010. Date of publication August 19, 2010; date of current version August 10, 2011. This work was supported in part by the C2S2 Focus Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation subsidiary and by the National Science Foundation under Grant 0903485 and Semiconductor Research Corporation under Contract 2009-TJ-1987.

Z. Feng is with the Department of Electrical and Computer Engineering, Michigan Technological University, Houghton, MI 49931 USA (e-mail: zhuofeng@mtu.edu).

Z. Zeng and P. Li are with the Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX 77843 USA (e-mail: albertzeng , pli)@neo.tamu.edu.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2010.2059718

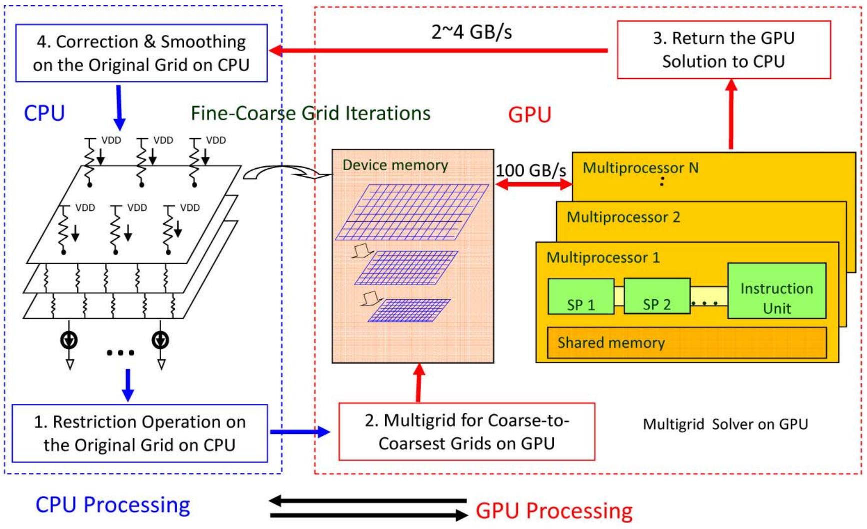
4. Correction & Smoothing on the Original Grid on CPU
2~4 GB/s
3. Return the GPU Solution to CPU
CPU
Fine-Coarse Grid Iterations
GPU
VDD
Device memory
100 GB/s
Multiprocessor N
Multiprocessor 2
Multiprocessor 1
SP 1
SP 2
Instruction Unit
Shared memory
1. Restriction Operation on the Original Grid on CPU
2. Multigrid for Coarse-to-Coarsest Grids on GPU
Multigrid Solver on GPU
CPU Processing
GPU Processing

Fig. 1. Overall HMD analysis flow.

strategies, restriction, and prolongation operators can be developed rather efficiently. The proposed multigrid method is referred to as a hybrid approach in the sense that the entire workload is split between the host (or CPUs) and the device (GPUs), in which the multigrid operations for the finest grid (original power grid) are performed on CPU, while the computations for the coarser to coarsest grids are done on GPU. The multigrid hierarchy is purposely made deep such that most of the work is pushed onto the GPU, and only a small fraction of finest grid residue computations and smoothing operations is conducted on the CPU. Through conducting extensive experiments, we show that in practice the required number of CPU-GPU HMD iterations is typically small and the power grid solutions can converge fast in linear time. Additionally, using the HMD solver as a preconditioner, more robust Krylov subspace iterative methods can be naturally exploited.

II. Background and Overview

We first review the power grid analysis problems and the GPU architecture. Next, an overview of the proposed GPU-based multigrid approach is provided with more details.

A. *Power Grid Analysis*

The power grid analysis covers two main aspects: DC and transient analysis. As for DC analysis, power grid problems are typically formulated into a linear system as [8], [23], [17], [14], [24]

$$Ax = b \tag{1}$$

where A is a symmetric positive definite matrix representing the interconnected resistors, x is a vector including all node voltages and b is a vector containing all independent sources. Directly solving such a large system using LU or Cholesky matrix factorizations is typically very expensive and requires huge memory resources [22], [20]. Iterative methods [8], [23], [14] are memory efficient, but may suffer from slow convergence.

Specifically, point relaxation methods update node voltage using its neighboring nodes repeatedly until achieving converged solution

$$x_j = \sum_{i \neq j} \frac{g_i}{\sum g_i} x_i - \frac{b_j}{\sum g_i} \tag{2}$$

where x_j is the node voltage to be updated and b_j is the current source flowing out the node, while g_i and x_i are the neighboring conductances and voltages.

In this work, we only focus on the power grid DC analysis, though transient analysis can be conducted in a similar way with promising performance.

B. *Challenges in Developing Solvers on GPU*

A basic understanding of GPU's hardware architecture is instrumental for evaluating the potential of applying GPU matrix solvers to large power grid problems. Consider a recent commodity GPU model, NVIDIA G80 series. Each card has 16 streaming multiprocessors (SMs) with each SM containing eight streaming processors (SPs) running at 1.35 GHz. An SP operates in single-instruction, multiple-thread (SIMT) fashion and has a 32-bit, single-precision floating-point, multiply-add arithmetic unit [18]. Additionally, an SM has 8192 registers which are dynamically shared by the threads running on it and can access global, shared, and constant memories. The bandwidth of the off-chip memory can be as high as 86 GB/s, but the memory bandwidth may reduce significantly under many random memory accesses. The following programming guidelines play very important roles for efficient GPU computing [1].

1) *Low control flow overhead:* Execute the same computation on many data elements in parallel.
2) *High floating point arithmetic intensity:* Perform as many as possible calculations per memory loading/writing.
3) *Minimum random memory access:* Pack data for coalesced memory access.

Due to the very nature of the SIMT architecture, it remains a challenge to implement efficient general-purpose sparse matrix solvers on GPU. In recent such attempts, it is reported that most of running time is spent on data fetching and writing, but not on data processing [5], [7]. For instance, traditional iterative methods such as (preconditioned) conjugate gradient and multigrid methods [5] involve many sparse matrix-vector computations, leading to complex control flows and a large number of random memory accesses that can result in extremely inefficient GPU implementations. On the other hand, a problem with a structured data and memory access pattern can be processed by GPU rather efficiently. For instance, the performance of a dense matrix-matrix multiplication kernel on GPU can reach a performance of over 90 GFLOPS, which is orders of magnitude faster than on CPU [18]. Considering the above facts, it is unlikely to facilitate efficient power grid analysis by building around immature general-purpose GPU matrix solvers or implementing existing CPU-oriented power grid analysis methods [8], [14], [17] on GPU.

C. Multigrid Methods for Power Grid Analysis

Multigrid methods are among the fastest numerical algorithms for solving large PDE-like problems [6], where a hierarchy of exact to coarse replicas (e.g., fine versus coarse grids) of the given linear problem are created. Via iterative updates, high and low frequency components of solution errors are quickly damped on the fine and coarse grids, respectively, contributing to the good efficiency of multigrid. When properly designed, multigrid methods can achieve a linear complexity in the number of unknowns.

1) Prior Multigrid Methods for Power Grid Analysis: Multigrid methods typically fall into two categories, geometric multigrid (GMD), and algebraic multigrid (AMG). AMG may be considered as a robust black-box method and requires an expensive setup phase while GMD may be implemented more efficiently if specific geometric structures of the problem can be exploited. The key operations of a generic multigrid method include the following.

1) *Smoothing:* Point or block iterative methods (e.g., Gauss-Seidel) applied to damp the solution error on a grid.
2) *Restriction:* Mapping from a fine grid to the next coarser grid (applied to map the fine grid residue to the next coarser grid).
3) *Prolongation:* Mapping from a coarse grid to the next finer grid (applied to map the coarse grid solution to the next finer grid).
4) *Correction:* Use the coarse grid solution to correct the fine grid solution.

On the kth level grid with an initial solution v_k, a typical multigrid cycle $\mathrm{MG}(k, v_k)$ has the following steps [6]:

1) apply a presmoothing step to update the kth level grid solution;
2) compute the residue on the kth level grid and map it to the $(k+1)$th level grid via restriction;
3) with the input that is mapped from the kth level grid, solve the $(k+1)$th level grid directly if the coarsest grid level is reached, otherwise apply a multigrid cycle $\mathrm{MG}(k+1, v_{k+1})$ with a zero initial guess $v_{k+1} = 0$;
4) map the $(k+1)$th level grid solution v_{k+1} to the kth level grid correction component via prolongation operation, and correct the kth level grid solution by: $v_k = v_k + v_{k+1}$;
5) apply a post-smoothing step to further improve solution quality of v_k and return the final solution result v_k.

Existing multigrid methods for power grid analysis fall into two categories, namely GMD-like methods [14], [21] or AMG-like methods [19], [24]. To create coarser grid in GMD-like algorithms, a two-step approach is typically adopted. The first step is to come up with a coarser grid by skipping power grid nodes geometrically, while the second step is to set up a corresponding conductance matrix for the coarser grid [14], [21]. By iterative applying the above procedure, all coarse level grids and their corresponding conductance matrices can be built before multigrid algorithms start. On the other hand, the AMG-like multigrid algorithms generate coarse grid levels merely based on the conductance matrix properties [19], [24].

Once the conductance matrices for coarse level grids are constructed, multigrid operations such as smoothing, restriction, prolongation, and correction can be performed using sparse matrix-vector operations. The coarsest grid can be solved directly or iteratively. It should be noted that compared with GMD-like algorithms, AMG-like algorithms may face more complicated setup phases and end with much denser matrices for the coarsest level grid, which may lead to less memory and runtime efficiencies.

As described above, all prior multigrid-like power grid simulation methods heavily depend on sparse matrix operations to achieve their scalability, whose implementations can bring huge challenges to GPU computing (see Section II-B). In this paper, we propose novel HMD algorithms and their implementations specifically for parallel GPU-based power grid analysis.

2) Our Approach—GPU-Specific HMD Method: To achieve good analysis efficiency on GPU's SIMT platforms, we propose a HMD method for GPU-based power grid analysis. The idea behind HMD method is to use the original 3-D irregular power grid as the finest grid in the multigrid hierarchy, and adopt a set of regularized 2-D grids as the coarser to coarsest level grids. By handling the irregular grid (finest grid in multigrid hierarchy) on CPU and the 2-D regularized grids on GPU, we can take the most advantages of GPU's SIMT computing capability. In general multigrid methods, relaxation (smoothing) steps are introduced to damp out the high frequency (short-range) solution errors, while the remaining lower frequency error components are mapped to the coarser level grids, becoming their high frequency errors. In our HMD method, we apply the smoothing step on the original irregular grid on CPU, while all other multigrid operations for the coarser grid levels are done on GPU.

It should be noted that, the 2-D grid regularity mentioned above means topological regularity (interconnect conductances may vary). By approximating the original irregular grid using a 2-D regularized grid, we ensure that each circuit node in such a 2-D approximation exactly connects to its four neighbors. This will allow us to use simple array storages to present the grid, leading to very regular data access patterns and control flows. These factors are crucial for achieving high computational efficiency on GPU.

Considering GPU's limited on-chip shared memory resources, the hierarchical iterative nature of multigrid is attractive to GPU platforms. A key enabler for GPU-based power grid simulation is an efficient multigrid solver on GPU

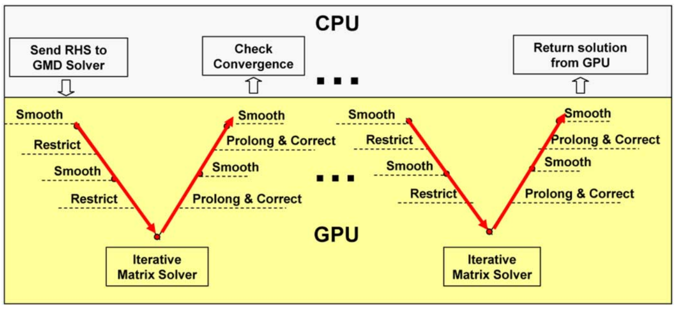

Fig. 2. Acceleration of multigrid solve on GPU.

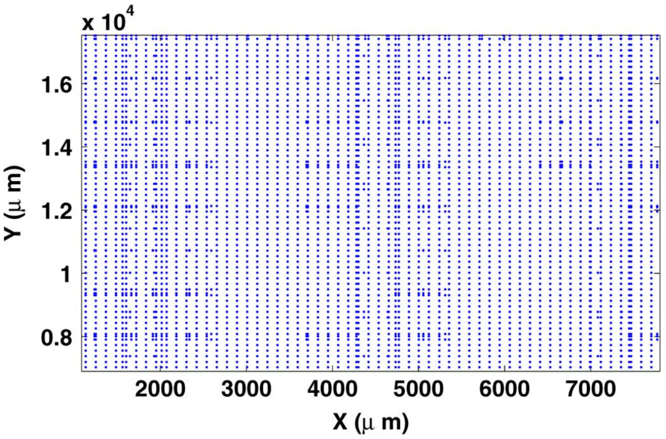

Fig. 3. Top view of node distribution of an industrial power grid design.

for handling the 2D coarser to coarsest grids that arise from the original 3-D power grid structure, for which we propose a GPU-specific multigrid algorithm. For the 2-D regularized coarse level grids, we perform all the multigrid computations on GPU as shown in Fig. 2, involving simple flow controls and highly regular memory access patterns that are favored by the GPU's streaming data processing.

Power grid solution can be efficiently computed using our custom multigrid solver (as shown in Fig. 1), with no explicit sparse matrix-vector operations. The computations associated with the GPU-based multigrid steps constitute the dominant workload of the entire HMD approach. As for the convergence rate, the algorithm shares the common properties of multigrid methods: solution will converge in linear time. Our experiments on a variety of industrial power grid designs show that after only a few HMD iterations, power grid error components can be damped out quickly (with maximum errors smaller than 1 mV and average errors smaller than 0.1 mV).

Denoting the true (original) power grid by Grid_O and the regularized grid by Grid_R, HMD iterations involve the following main steps (see Fig. 1).

1) *CPU:* Compute the original grid (Grid_O) residue using the latest solution, and map the residue to the input of grid Grid_R.
2) *GPU:* Solve the regularized grid Grid_R using GMD algorithm and then return its solution back to the original grid Grid_O.
3) *CPU:* Correct the original grid solution using the previous GPU result, and apply a post smoothing step.
4) *CPU:* If the maximum (average) error is below a threshold, exit; otherwise repeat the above steps.

Since general-purpose CPU is more efficient and flexible for handling the original (irregular) 3-D power grid, the bulk computation of the entire HMD algorithm is performed on GPU through solving the regularized 2-D grid (Step 2). Only a fraction of the work, such as the residue computation and smoothing steps, is preformed on the host.

Although we have observed good performance and robustness of the proposed HMD algorithm in extensive experiments, we also develop a preconditioned conjugate gradient (PCG) scheme, where our HMD solver is used in the inner loop as a preconditioner. This choice provides a theoretical convergence guarantee under the framework of preconditioned CG. With the high quality HMD preconditioner, the number of PCG iterations can be dramatically reduced. And as before, the bulk of the computation is accelerated on the GPU.

III. 3-D-to-2-D Grid Regularization

The number of our HMD iterations depends on how well the coarse level grids are constructed. If the coarse grids can properly approximate the original 3-D grid, HMD algorithm for power grid analysis can converge very fast. On the contrary, if not well designed, the number of multigrid iterations may increase. In this section, we propose a simple yet effective way for obtaining good regularized 2-D grid structure (coarse level grid) from the original power grid netlist.

A. Getting A Coarse Level 2-D Grid

The idea is to stamp the original 3-D power grid elements onto a 2-D regularized grid such that the electrical property of the original grid can be well preserved on the 2-D grid. As such, the regularized 2-D grid can provide good solution approximation for the original power grid, which therefore allows a fast convergence of the HMD algorithm.

Industrial power grid designs [16], [2] typically exhibit globally uniform grid structures (as shown in Fig. 3), except for some local grids that have irregular patterns. Consequently, we propose to generate the coarse level 2-D grid using typical wire pitch values (of the bottom metal layer) and stamp all the original multilayer irregular grid elements (resistors, current and voltage sources) onto the 2-D grid. In this way, the electrical properties of the original power supply network can be well preserved on the coarse level grid, while the resultant grid storage pattern allows highly efficient GPU data access and processing.

It should be emphasized that the purpose of the above element stamping procedure is to get a relatively good coarse level grid approximation for the HMD iterations. The 2-D regularized grid does not need to fully represent the original power grid structure, though a better approximation can lead to faster convergence. Since the via resistance is typically much smaller than the grid resistance, in the coarse grid generation step, we simply collapse the original 3-D grids to the 2-D grid and remove the via resistors connecting different layers. As we show in Table I, the 3-D grid nodes that have been connected through via resistors have negligible voltage difference (much smaller than 1

TABLE I
TOP AND BOTTOM LAYER NODE SOLUTIONS (N_l IS THE NUMBER OF METAL LAYERS, ΔV IS THE NODE VOLTAGE RANGES, E_{avg} IS THE AVERAGE VOLTAGE DIFFERENCE BETWEEN THE TOP AND BOTTOM METAL LAYERS, AND $E_{\text{rel}}\%$ IS THE RELATIVE VOLTAGE DIFFERENCE, FOR THE VDD/GND GRIDS, RESPECTIVELY.)

CKT	N_{node}	N_l	$\Delta V(mv)$	$E_{avg}(mv)$	$E_{rel}\%$
$ibmpg2$	127,238	5	337/270	0.87/0.63	0.06/0.20
$ibmpg3$	851,584	5	171/151	0.01/0.01	0.01/0.10
$ibmpg4$	953,583	6	5.3/2.4	0.02/0.02	0.00/0.78
$ibmpg5$	1,079,310	3	48/27	0.05/0.04	0.00/0.15
$ibmpg6$	1,670,494	3	154/112	0.1/0.06	0.01/0.10

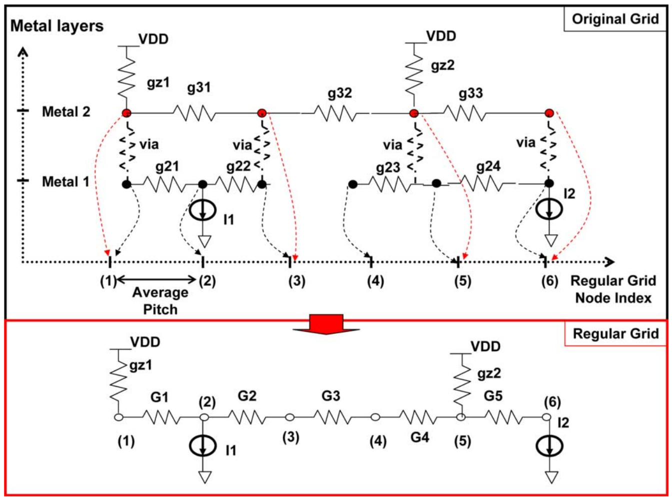

Fig. 4. Coarse grid generation using 3-D-to-2-D topological regularization.

mV). The above 2-D grid regularization procedure is shown as follows.

1) *Step 1*: By neglecting via resistances, all the metal layers in the network are overlapped on the same 2-D plane (X–Y plane), forming a collapsed 2-D irregular grid.
2) *Step 2*: By examining the original grid pitches in X- and Y-directions, fixed pitch values (for both X- and Y-directions) are selected, such that locations of the 2-D grid nodes can be determined.
3) *Step 3*: Circuit elements are stamped onto the above 2-D regularized grid. Elements that occupy multiple regular grid nodes have to be decomposed into smaller pieces (larger conductance values) during the stamping, while current and voltage sources are simply stamped according to their geometrical locations.

A simple example has been shown in Fig. 4, in which a simple two-metal-layer irregular power grid is stamped onto a single-layer regularized grid. The conductance values on the regularized grid can be determined as follows:

$$\begin{aligned} G_1 &= 2g_{31} + g_{21} \\ G_2 &= 2g_{31} + g_{22} \\ G_3 &= 2g_{32} \\ G_4 &= 2g_{32} + g_{23} \\ G_5 &= g_{33} + g_{24}. \end{aligned} \tag{3}$$

Note that the above procedure is not the only way to generate the coarse level (regularized) grid structure for multigrid processing. In this work, we notice that such a grid construction method can provide a good coarse level grid approximation for the original power grid circuit.

Due to the irregularity of the original 3-D grid, some of the coarse level grid nodes may not correspond to any of the original grid nodes. In this case, small dummy conductances ($G_{\min} = 1e - 6$) are inserted between such grid nodes. Note that the uniform pitches of the 2-D grid may be set to the average pitch values of the irregular grid, and can be adjusted. Typically, smaller uniform pitch values lead to increased 2-D grid sizes and better approximation of the original grid. Our experience shows that slightly changed coarse grid sizes do not significantly influence the overall runtime efficiency, which is due to the linear complexity of multigrid algorithms. For the GPU hardware used in this work, we set the 2-D regularized grid size to be slightly smaller than the original grid size (50% to 90% of the original grid size). It should be noted that if in the future, GPU computing power gets dramatically improved (compared with the CPU), larger 2-D regularized grid sizes can be used to ensure a faster multigrid convergence (fewer HMD iterations).

B. Regular Data Structure for 2-D Coarse Level Grids

The 2-D coarse grids obtained from the above 3-D-to-2-D regularization step, can be stored using regular data structures (e.g., 1-D or 2-D array), which ensures efficient coalesced memory access of GPU device memory. For instance, assume a regular 2-D coarse grid is stored in a 2-D lookup table (LUT), and the two table indices are the 2-D regular grid node indices as shown in Fig. 4 (only 1-D regular grid index is shown in the figure). Then the x-, y-, and z-directional interconnects as well as the current source that connect to a grid node $N[i,j]$, can be stored in the following four 2-D LUTs.

- $G_x[i,j]$: Horizontally connected conductance between node $N[i,j]$ and node $N[i+1,j]$.
- $G_y[i,j]$: Vertically connected conductance between node $N[i,j]$ and node $N[i,j+1]$.
- $G_z[i,j]$: The conductance that connects node $N[i,j]$ and the voltage sources.
- $I_z[i,j]$: The current source that flows out node $N[i,j]$.

Since the above 2-D grid resistors are stored in consecutive memory space (not in a sparse matrix), multigrid operations can be directly performed based on these 2-D LUTs, leading to GPU-friendly coalesced memory accesses and high-throughput GPU computing.

IV. MULTIGRID OPERATIONS FOR COARSE LEVEL GRIDS

While 2-D regularized grids can be obtained in a relatively straightforward manner, developing an efficient multigrid grid algorithm for GPU computing is nontrivial. Naive implementations for either data transferring or processing can lead to severe performance degradation. This section discusses several key steps of the GPU-based multigrid operations.

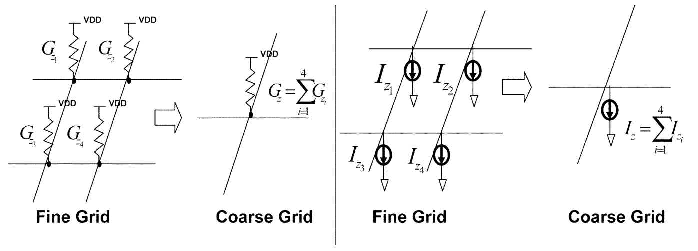

Fig. 5. VDD pads (G_z) and current sources (residues) in fine/coarse grids.

A. Coarse Grid Hierarchy

Based on the 2-D regularized grid obtained from the original 3-D grid (as shown in Section III), a set of increasingly coarser grids are created to occupy the higher grid levels. In this case, the 2-D grid produced by the previous stamping step serves as the second finest grid in our multigrid method, while the original 3-D grid serves as the finest grid sitting at the bottom of the multigrid hierarchy. Ideally, these coarse grids should be created such that the increasingly global behavior of the finest grid is well preserved using a decreasing number of grid nodes. Unlike in CPU based multigrid methods, here on GPU, it is critical to carry the good regularity of the coarse level grids throughout the multigrid hierarchy so as to achieve good efficiency on the GPU platform. The goal is achieved from the following view of the I/O characteristics of the power grid.

When creating the next coarser grid, we distinguish two types of wire resistances: resistances connecting a grid node to a VDD source (or VDD pad conductances) versus those connecting a grid node to one of its four neighboring nodes (or internal resistances) on the regular grid, as shown in Fig. 5. Importantly, the two types of resistances are handled differently. We maintain the same total current I_z that flows out the network and the same total wire conductance (G_z) that connects the grid to ideal voltage sources (e.g., total VDD pad conductance). In this way, the same pull-up and pull-down strengths are kept in the coarser grid of a power distribution network. Denote the voltages of M grid nodes that connect to ideal voltage sources via M wires by V_i for $i=1,\ldots,M$, and the N loading current sources by I_j for $j=1,\ldots,N$. The following equation holds:

$$\sum_{i=1}^{M}(\text{VDD}-V_i)\,G_{z_i}=\sum_{j=1}^{N} I_{z_j}. \tag{4}$$

To maintain approximately same node voltages V_i at VDD pad locations in the coarser grid, we ensure that $\sum_{i=1}^{M} G_{z_i}$ and $\sum_{j=1}^{N} I_{z_j}$ are unchanged. Consequently, as shown in Fig. 5, both the VDD pad conductance (G_z) and current loadings (or residues) are summed up when creating the coarser grid problem. Differently, internal conductances are averaged to create a coarser regular grid that approximately preserves the global behavior of the fine grid.

Using H and h to label the fine and coarser grid components, respectively, the coarser grid can be created as follows:

$$\begin{aligned}G_x^h[i,j]&=\frac{1}{4}\times\big(G_x^H[2i,2j]+G_x^H[2i+1,2j]\\&\quad+G_x^H[2i,2j+1]+G_x^H[2i+1,2j+1]\big)\\G_y^h[i,j]&=\frac{1}{4}\times\big(G_y^H[2i,2j]+G_y^H[2i+1,2j]\\&\quad+G_y^H[2i,2j+1]+G_y^H[2i+1,2j+1]\big)\\G_z^h[i,j]&=\big(G_z^H[2i,2j]+G_z^H[2i+1,2j]\\&\quad+G_z^H[2i,2j+1]+G_z^H[2i+1,2j+1]\big)\end{aligned} \tag{5}$$

where i and j denote 2-D grid location (2-D LUT indices), and the numbers of nodes along the horizontal and vertical directions are reduced by a factor of two in the coarser grid (total grid size reduced by a factor of four).

B. Inter-Grid Operators

The restriction and prolongation operators in this work are defined in the following way to allow more efficient GPU processing.

Restriction:

$$\begin{aligned}R^h[i,j]&=R^H[2i,2j]+R^H[2i+1,2j]\\&\quad+R^H[2i,2j+1]+R^H[2i+1,2j+1].\end{aligned} \tag{6}$$

Prolongation:

$$\begin{aligned}E^H[2i,2j]&=E^H[2i,2j+1]=E^H[2i+1,2j]\\&=E^H[2i+1,2j+1]=E^h[i,j]\end{aligned} \tag{7}$$

where residues and errors (solution corrections) are denoted by R and E, respectively. Apparently, the coarser level grids and the associated inter-grid operations are defined completely based on 2-D grid geometries that can be stored and processed in the regular data structure introduced in Section III.

In our multigrid flow, the coarsest grid (with a few hundreds of nodes) is solved using an iterative block-Jacobi algorithm which can be suitably implemented for GPU platform. To reduce the overhead of this coarsest grid solving, the multigrid hierarchy is purposely made deep. In our experiments, four to eight grid levels are used, such that the sizes of the coarsest problem vary from a few hundred to a few thousand times smaller than the original grid sizes. This choice may push more than 99% of overall multigrid computations onto GPU hardware (except a convergence check on CPU).

C. Point Versus Block Smoothers

The choice of smoother is critical in multigrid algorithm development. Typically, point Gauss–Seidel or weighted Jacobi smoothers are used for CPU-based multigrid methods. In this work, a block-based smoother is adopted to fully utilize GPU's computing resources. On GPU, a number (more precisely a warp [1]) of threads may be simultaneously executed in a SIMD fashion on each streaming multiprocessor (SM). This implies that multiple circuit nodes can be processed at the same time. For instance, assume a block of circuit nodes are loaded onto an SM at a time. Then, multiple treads are launched to simultaneously smooth the circuit nodes in the block using a number of iterations. Finally, the above processing step (almost) completely solves the circuit block, effectively leading to a block smoother. The above approach ensures that a meaningful amount of computing work is done before the data (in shared memory or registers) is released, and a new memory access takes place. In other words, it contributes to efficient GPU computing by increasing the arithmetic intensity. This block smoother will be discussed in detail in Section V.

V. ACCELERATING MULTIGRID ON GPU

To gain good efficiency on the GPU platform, care must be taken to facilitate thread organization, memory and register allocation, workload balancing as well as hardware-specific algorithm development.

A. Thread Organization

Through a suitable programming model (e.g., CUDA [1]), GPU threads shall be packed properly for efficient execution on SMs. On an SM, threads are organized in units of thread blocks, where the number of blocks should be properly chosen to maximize the performance. A GPU-friendly thread block typically includes multiples of 32 threads for a commercial GPU [1]. In our implementation, the actual optimal thread block size is chosen experimentally.

B. Memory and Register Allocation

Before the multigrid solve starts on GPU, 2-D LUTs are allocated on CPU (host) to store all the regularized grids in the multigrid hierarchy. Then, LUT data is transferred from host to device (GPU). We bind the conductance LUTs (G_x, G_y and G_z) to GPU's texture memory and other data to GPU's on-board device memory (DRAM). Since texture memory is cached, its access latency is significantly smaller (around two times smaller) than the global (device) memory latency. However, GPU's texture memory is read-only and cannot be used for grid solution updates. Therefore, residues, solution and error vectors are stored in the device memory (which is not cached), for which coalesced memory accesses should be employed to gain good memory bandwidth.

GPU's fast on-chip shared memory and registers are very limited. For instance, each SM of our GPU hardware has 16 kb shared memory and 8 kb registers. If the GPU kernel function consumes greater shared memory or register resources that exceed what are available, the application will fail. It is also important to allow (assign) more than one block of threads to run concurrently on each SM. This strategy helps hide GPU memory latency and will lead to a much higher performance throughput. In this work, all components of our GPU accelerated multigrid algorithm are developed carefully to fully utilize GPU hardware resources.

C. Mixed Block-Wise Smoother

In our multigrid algorithm flow, the relaxation (smoothing) step takes more than 90% computation time. Hence, an efficient implementation of the smoother is critical. For typical CPU-based multigrid solver, point-wise iterative methods such as Gauss–Seidel or weighted Jacobi smoothers are often adopted for good performance. For streaming data parallel computing on GPU, it is important to maximize the arithmetic intensity (computations/memory accesses), ensure efficient coalesced (block) memory accesses and simplify control flows. To this end, we propose a *global block Gauss–Seidel iteration* (GBG iteration) scheme and a *local block weighted Jacobi iteration* (LBJ iteration) scheme for the GPU-based multigrid algorithm. As illustrated in Fig. 6, during each GBG iteration, the whole 2-D regularized coarser to coarsest grids are partitioned into smaller blocks, and subsequently transferred to the SMs on GPU. Next, k times LBJ iterations are performed for the nodes within each circuit block locally. Since only the threads within the same thread block are able to share their data with each other (in the shared memory or registers), the node solutions of the block can not be shared by other blocks unless their solutions are sent back to GPU's global memory (DRAM). Using the latest neighboring block solutions in the global memory, the above local LBJ iterations can be repeatedly performed to iteratively update all the grid block solutions. Therefore, from a global point of view, the above iteration (relaxation) scheme can be considered as a global block Gauss–Seidel smoother (GBG), in which circuit block solutions are updated using the latest neighboring block solutions. It is also clear that the local block iteration scheme for the nodes within each grid block can be considered as a local block Jacobi smoother (LBJ), in which all the grid nodes within a block are updated locally by multiple threads simultaneously in a weighted Jacobi fashion. The above two iteration schemes have been carefully tailored for our GPU-based multigrid algorithm, particularly through the following considerations.

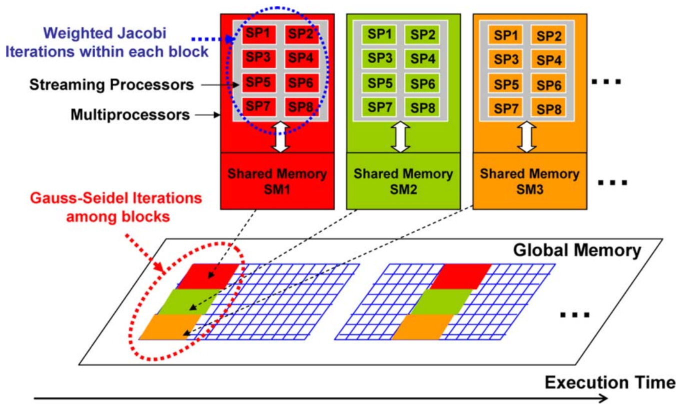

Fig. 6. Mixed block relaxation (smoother) on GPU.

1) To increase the arithmetic intensity, we perform k times LBJ iterations for each global memory access. k can be determined based upon the block size: larger block size requires to include more LBJ iterations. However, excessive local iterations may not help the overall convergence since block boundary information has not been immediately updated.
2) To hide the memory latency and thread synchronization time, we allow two or more grid blocks to run concurrently on each SM to avoid idle processors during GPU thread synchronization and device memory access. This method is especially important for improving the GPU computing performance of bandwidth limited applications.

The block size may impact the overall performance significantly. A too large block size can lead to slow convergence while a too small size may cause bad memory efficiency and shared memory bank conflicts. To minimize shared memory and register bank conflicts, block sizes such as 4×4 or 8×8

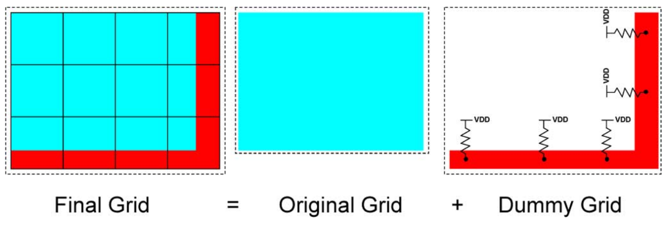

Fig. 7. Appending dummy grid nodes for a chosen block size.

are observed to offer good performance (good tradeoff between GPU computing efficiency and the multigrid algorithm efficiency). In our GPU algorithm implementation, we use a block of 8×8 (4×4) GPU threads to work on a block of 8×8 (4×4) regular grid nodes. In this way, at any time of GPU computing, 128 regular grid nodes are processed by 128 streaming processors (for NVIDIA's 8 series GPU cards), enabling highly efficient massively parallel computing capabilities.

D. Dummy Grid Nodes

As discussed before, GPU data processing favors block-based operations. If the 2-D regularized grid dimensions are not multiples of the thread block size, extra handling is required. For example, assume one smoothing kernel of the multigrid solver is executed on all coarse level grids using the 8×8 thread blocks. Then, the dimensions of the coarse level grids need to be multiples of the block size (8×8). To this end, certain dummy grids can be attached to the periphery of the coarse grids. It is very important to isolate these dummy grids from the original grid, as shown in Fig. 7. Otherwise, the multigrid convergence can be significantly impacted.

E. An Example of Block–Jacobi Iteration on GPU

By denoting regular grid node solution by V, and assuming that 2-D LUTs are used to store the 2-D regular grid (introduced in Section III), we show GPU memory access pattern of one mixed block-Jacobi iterations using shared memories and registers as follows (block size is 4×4 in this example).

1) Load 4×4 grid solution data V from the global memory to the shared memory and 4×4 regular grid data G_h, G_v, G_z, and RHS to the registers.
2) Load four boundary solutions and grid data to shared memory and registers.
3) Do k times local block Jacobi iterations.
4) Return the 4×4 grid solution data V to global memory for block solution update.

The above block Jacobi iteration is accomplished by using a 4×4 GPU thread block (including 16 threads), as shown in Fig. 8, where "X" represents the shared memory address that is accessed by a GPU thread, and the tiles in red represent the shared memory (registers) that stores the solution data (grid data). As shown, the central 4×4 regular grid block solution is updated through block Jacobi iterations (using four of its neighboring block solutions) by a 4×4 GPU thread block, which is followed by returning its latest block solution to GPU's global memory space. It is not hard to imagine that a few tens of such grid blocks (a few hundreds of power grid nodes) can be processed simultaneously using GPU's hundreds of cores (streaming processors), which is not possible for CPU computing.

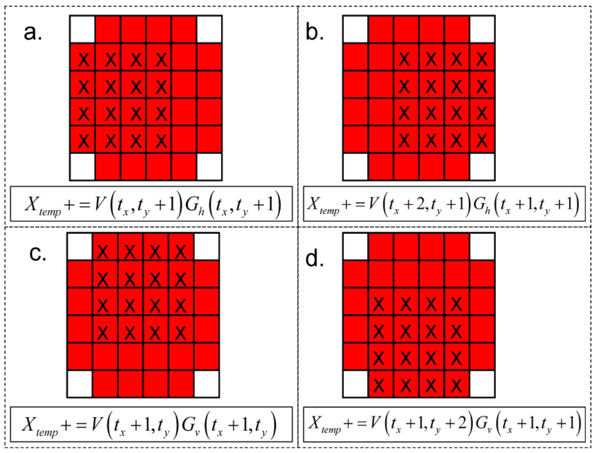
a.
$X_{temp} += V(t_x, t_y+1)G_h(t_x, t_y+1)$
b.
$X_{temp} += V(t_x+2, t_y+1)G_h(t_x+1, t_y+1)$
c.
$X_{temp} += V(t_x+1, t_y)G_v(t_x+1, t_y)$
d.
$X_{temp} += V(t_x+1, t_y+2)G_v(t_x+1, t_y+1)$

Fig. 8. GPU memory access pattern during a block Jacobi iteration.

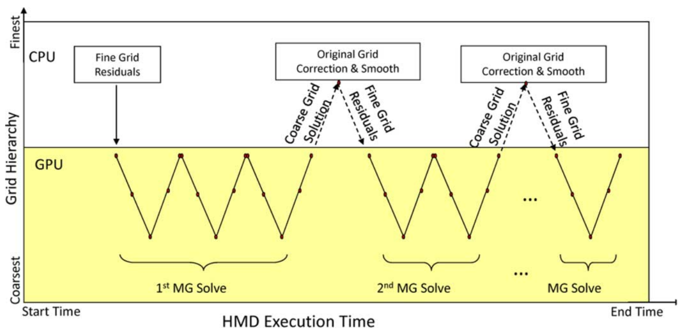

Fig. 9. Hybrid GPU-CPU HMD iterations.

VI. Hybrid Multigrid for Power Grid Analysis

After discussing the multigrid operations on GPU in Section V, we consider the full HMD flow in this section. By performing GPU-based multigrid operations for the 2-D coarse to coarsest grids on GPU, we can obtain approximate node solutions to the original power grid system. By introducing an extra smoothing and correction step for the original grid on CPU, more accurate solution can be computed. The above HMD iteration framework has been shown in Figs. 1 and 9, and also outlined in Section II-C.

A. Problem Formulation

Assume that for a 3-D irregular power grid (Grid_O), the following large linear system of equations need to be solved:

$$A \cdot x = b \tag{8}$$

where $A \in \mathbb{R}^{n \times n}$ is the original grid system matrix, representing a linear operator $A(x) : \mathbb{R}^n \rightarrow \mathbb{R}^n$, $x = x^* \in \mathbb{R}^n$ is the exact solution vector to be solved, and $b \in \mathbb{R}^n$ is the right-hand side (RHS). Denote the system matrix of the 2-D regularized grid (Grid_R) as $A_r \in \mathbb{R}^{m \times m}$, which is a linear

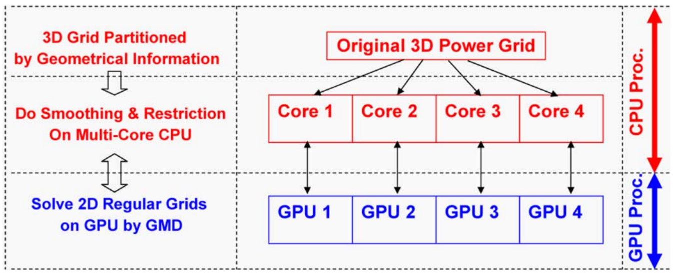
3D Grid Partitioned by Geometrical Information
Do Smoothing & Restriction On Multi-Core CPU
Solve 2D Regular Grids on GPU by GMD
Original 3D Power Grid
Core 1
Core 2
Core 3
Core 4
GPU 1
GPU 2
GPU 3
GPU 4
CPU Proc.
GPU Proc.

Fig. 10. Power grid simulation scheme on multi-core-multi-GPU platform.

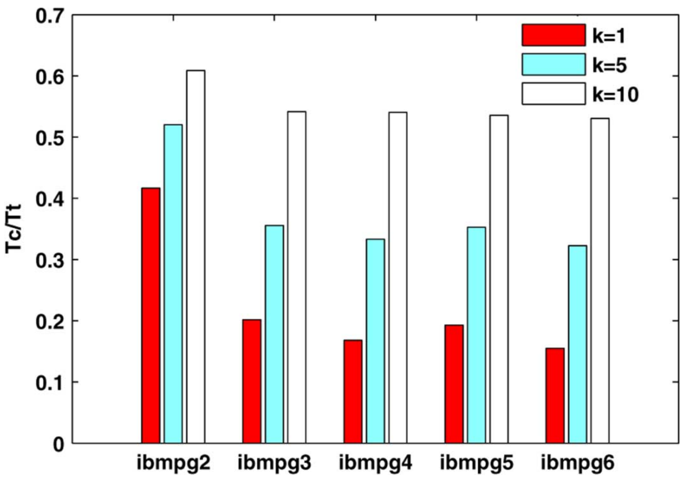

Fig. 11. Ratio of GPU computation time (T_c/T_t). T_c is the GPU computing time and T_t is the total computing time.

operator $A_r(x) : \mathbb{R}^m \rightarrow \mathbb{R}^m$. Denote the solution of the original grid in the kth HMD iteration as $x^{(k)} \in \mathbb{R}^n$. The following steps are performed in the kth HMD iteration. The residue $r^{(k)}$ associated with $x^{(k)}$ is computed and mapped onto $r_r^{(k)}$ on the 2-D grid (Grid_R) as

$$r^{(k)} = b - A \cdot x^{(k)}, \quad r_r^{(k)} = V_o^r \cdot r^{(k)} \tag{9}$$

where $V_o^r \in \mathbb{R}^{m\times n}$ is a proper linear operator $(\mathbb{R}^n \rightarrow \mathbb{R}^m)$. Note that the above simple computations are done on the CPU. With $r_r^{(k)}$, a solution correction $e_r^{(k)}$ is computed on the 2-D grid on GPU

$$A_r \cdot e_r^{(k)} = r_r^{(k)}. \tag{10}$$

$e_r^{(k)}$ is returned to the CPU host for further processing. $e_r^{(k)}$ is mapped back to the original grid (Grid_O) via

$$e^{(k)} = V_r^o \cdot e_r^{(k)} \tag{11}$$

where $V_r^o \in \mathbb{R}^{n\times m}$ is a proper linear operator $(\mathbb{R}^m \rightarrow \mathbb{R}^n)$. The solution for the finest grid is updated

$$x^{(k+1)} = x^{(k)} + e^{(k)}. \tag{12}$$

Finally, if the solution correction $(e^{(k)})$ is below a user-defined threshold, $x^{(k+1)}$ is returned as the final solution. Otherwise, proceed to the $(k+1)$th HMD iteration. The inter-grid $(\text{Grid}_O$ and $\text{Grid}_R)$ mapping operators V_o^r and V_r^o may be interpreted as a prolongation or restriction operator, respectively, as in a classical multigrid method, depending on the relative sizes of Grid_O and Grid_R. They are also constructed in a way similar to prolongation or restriction operators.

The HMD method has been described in detail in Algorithm 1. It should be emphasized that the proposed multigrid algorithm can converge fast to the true solution given a good coarse grid representation (such as the one proposed in Section III). The relaxation steps on the original grid Grid_O as well as the coarse level grids are important to successively damp out all the error components.

Algorithm 1 GPU-based HMD algorithm

Input: The conductance matrix $\mathbf{A} \in \mathbf{R}^{\mathbf{n}\times\mathbf{n}}$ obtained from the original irregular power grid (Grid_O with $\mathbf{n}$ nodes) sitting on CPU hardware, the 2-D regularized grid Grid_R with $\mathbf{m}$ nodes sitting on GPU device, the 2-D-to-3-D grid projection matrix $\mathbf{V_r^o} \in \mathbf{R}^{\mathbf{n}\times\mathbf{m}}$ (11), the 3-D-to-2-D grid projection matrix $\mathbf{V_o^r} \in \mathbf{R}^{\mathbf{m}\times\mathbf{n}}$ (12), the 2-D multigrid solver on GPU that computes the solution $\mathbf{x_r} = mgsolve(\mathbf{f_r})$ where the input and output vectors are $\mathbf{f_r}, \mathbf{x_r} \in \mathbf{R}^{\mathbf{m}\times 1}$, the RHS vector $\mathbf{b} \in \mathbb{R}^{\mathbf{n}\times\mathbf{1}}$ of conductance matrix $\mathbf{A}$, the initial solution guess $\mathbf{x_o^{(0)}} \in \mathbb{R}^{\mathbf{n}\times\mathbf{1}}$, the maximum number of iterations $\mathbf{K}$, the number of weighted Jacobi iterations $\mathbf{s}$, as well as the error tolerance $\mathbf{tol}$.

Output: The solution $\mathbf{x}$ for all grid nodes of Grid_O.

```
1:  Do $\mathbf{s}$ times relaxations for $\text{Grid}_O$ to get $\mathbf{x_o^{(0)}}$;
2:  Calculate the $\mathbf{r_o^{(0)}} = \mathbf{b} - \mathbf{A}\mathbf{x_o^{(0)}}$;
3:  for($\mathbf{k = 0; k < K; k++}$): do
4:     $\mathbf{r_r^{(k)}} = \mathbf{V_o^r r_o^{(k)}}$;
5:     $\mathbf{e_r}^{(k)} = mgsolve(\mathbf{r_r^{(k)}})$;
6:     $\mathbf{e_o^{(k)}} = \mathbf{V_r^o e_r^{(k)}}$;
7:     $\mathbf{x_o}^{(k+1)} = \mathbf{x_o}^{(k)} + \mathbf{e_o}^{(k)}$;
8:     Do $\mathbf{s}$ times relaxations on $\text{Grid}_O$ to get updated
       $\mathbf{x_o^{(k+1)}}$;
9:     $\mathbf{r_o^{(k+1)}} = \mathbf{b} - \mathbf{A}\mathbf{x_o^{(k+1)}}$;
10:    if$|\mathbf{r_o^{(k+1)}}| < \mathbf{tol}$ then
11:       exit the loop and return the solution $\mathbf{x_o^{(k+1)}}$;
12:    end if
13: end for
14: Return the solution $x = \mathbf{x_o^{(k+1)}}$;
```

B. HMD on Multi-Core-Multi-GPU System

The proposed HMD algorithm is highly parallelizable, and the workload can be easily partitioned based on the geometrical information of the power grid circuit. In this work, we propose to parallelize the HMD simulation on multi-core-multi-GPU system (see Fig. 10). As an example, assume we want to solve an N-node power grid on a quad-core machine with four GPUs, and we use each CPU core to talk with a GPU card, such that each small grid partition (e.g., a grid partition with about N/4 nodes) can be solved independently on a GPU by using the proposed multigrid method. When the solutions of all grid partitions are obtained and sent back to the full grid (in CPU's shared memory), a few smoothing steps can be performed and the grid

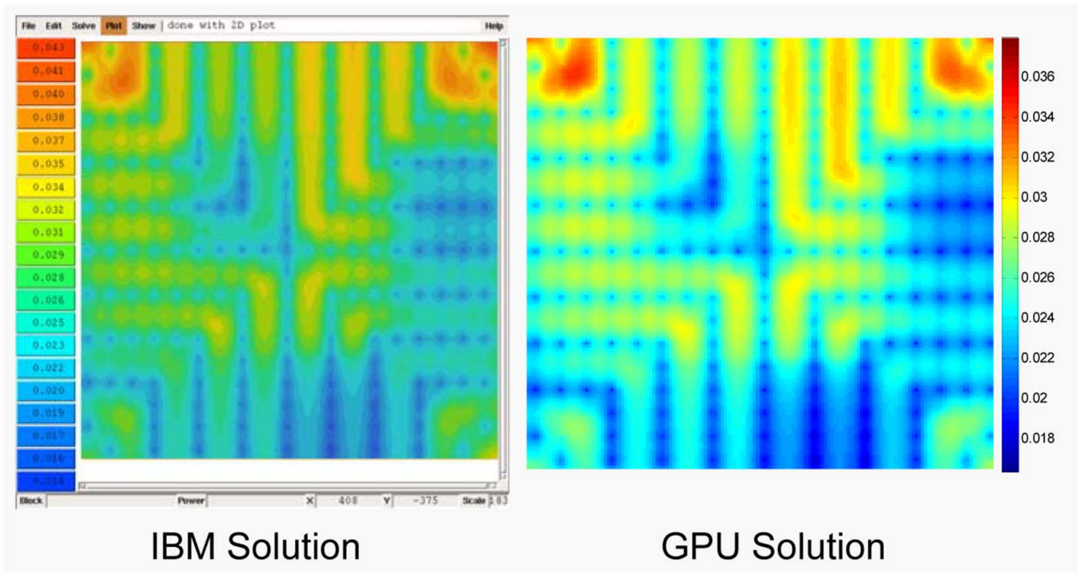

Fig. 12. Solution comparison of direct method and the GPU-based multigrid method (using one HMD iteration).

residuals can be computed subsequently. Next, new RHS vectors (residuals on full grid) are sent to GPUs again for subsequent multigrid iterations. In this manner, even very huge power grid designs can be handled efficiently. For instance, assume that each GPU can solve up to ten million grid nodes (with 512 Mb GPU memory), then a four-core-four-GPU system can solve up to 40 million nodes. It should be noted that each power grid partition should contain overlapping grid nodes of its neighboring partitions, to ensure a faster convergence rate.

To implement the multi-core-multi-GPU multigrid algorithm, the GPU code (written in NVIDIA's CUDA language [1]) is compiled to a static library that can be further invoked in the C/C++ code. Pthreads library [4] is used for the multithreading programming and each thread will control a single GPU card throughout the computation. Care must be given to data structure design of the GPU code: each power grid partition must be stored in the on-board memory (DRAM) of its own GPU device, while the full power grid solution are updated in the multi-core CPU's shared memory during each CPU-GPU iteration procedure. To minimize to the data communication between the host and device, only the boundary solution of each grid partition is transfered that only takes negligible time.

VII. Multigrid Preconditioned Conjugate Gradient Method

A preconditioned conjugate gradient method for power grid analysis has been proposed to improve the convergence rate [8]. However, such methods usually require finding good preconditioners for achieving decent performance. While the incomplete Cholesky factorization method has be shown to create good preconditioners for power grid analysis, such preconditioning technique may not be suitable for GPU-based parallel computation, in that there is not an efficient way to store and process the preconditioners (incomplete Cholesky factors) on GPU hardware. Instead of using the "black-box" incomplete factorization methods, in this work, we propose a GPU-accelerated multigrid-based preconditioner for power grid analysis.

TABLE II
Power Grid Specifications. Grid Size Is the Number of Nodes of the Original Power Grid, R. Grid Dim. Is the 2-D Coarse Grid Dimension, $N_{\mathrm{Lev.}}$ Is the Number of Multigrid Levels, N_{Vc} Is the Number of V-Cycles, ΔV (mV) Is the Difference Between the Maximum and Minimum Node Voltages $V_{\max} - V_{\min}$

CKT	$GridSize$	$R.GridDim.$	$N_{Lev.}$	N_{Vc}	$\Delta V(mv)$
$ibmpg2$	$127,238$	288×192	4	$4/4$	$347/275$
$ibmpg3$	$851,514$	896×432	5	$4/4$	$181/153$
$ibmpg4$	$953,583$	528×816	6	$8/8$	$5.3/2.6$
$ibmpg5$	$1,079,310$	544×992	6	$10/8$	$48/28$
$ibmpg6$	$1,670,494$	992×992	6	$10/10$	$154/86$

Multigrid-preconditioned conjugate gradient (MGPCG) methods [3], [13] have been proposed to combine the faster but less robust multigrid solver with the slower but more robust conjugate gradient method to form a more robust and parallelizable algorithm. In this work, we propose a GPU-accelerated multigrid preconditioned conjugate gradient method. More specifically, we do not form a preconditioner explicitly but rather use the GPU-based HMD solver as an implicit preconditioner. As shown in our experiments, with such a GPU-accelerated multigrid preconditioner, the power grid analysis requires significantly less iterations for achieving accurate solution than the traditional conjugate gradient method. In the current implementation, we accelerate the multigrid solver on GPU, while the rest of conjugate gradient computations are done on CPU.

VIII. Experimental Results

Extensive experiments are conducted to demonstrate the promising performance of the proposed GPU-based multigrid algorithm. A set of published industrial power grids [16], [2] and synthetic power grids are used to compare three solvers: the proposed GpuHMD, the CPU implementation of the same algorithm (CpuHMD), and the state-of-the-art CPU-based direct sparse matrix solver CHOLMOD [9]. We also show the results for the GPU-accelerated multigrid preconditioned

TABLE III
DC ANALYSIS RESULTS OF THE HMD SOLVER. $N_{\rm Iter}$ IS THE NUMBER OF HMD ITERATIONS, $E_{\rm avg}$ IS THE AVERAGE ERROR OF THE HMD SOLVER, AND $E_{\rm wst}$ IS THE WORST VOLTAGE DROP/BOUNCE ERROR. $T_{\rm gpu}/T_{\rm cpu}$ IS THE GPU/CPU RUNTIME OF HMD SOLVER. THE DATA FOR THE VDD AND GND GRIDS IS SHOWN IN THE FORM OF VDD/GND. SP. IS SPEEDUP OF THE GPUHMD SOLVER OVER THE CPUHMD SOLVER

CKT	N_{Iter}	$E_{avg}(mv)$	$E_{wst}(mv)$	$T_{gpu}(s)$	$T_{cpu}(s)$	$Sp.$	N_{Iter}	$E_{avg}(mv)$	$E_{wst}(mv)$	$T_{gpu}(s)$	$T_{cpu}(s)$	$Sp.$
$ibmpg2$	2/2	0.3/0.2	0.4/0.1	0.12	3.0	25X	3/3	0.0/0.0	0.0/0.0	0.18	4.3	24X
$ibmpg3$	2/2	0.6/0.5	1.2/1.0	0.72	31.0	43X	3/3	0.4/0.3	0.5/0.4	1.15	50.6	44X
$ibmpg4$	1/1	0.0/0.0	0.0/0.0	0.46	15.6	34X	2/2	0.0/0.0	0.0/0.0	0.62	21.1	34X
$ibmpg5$	2/2	0.6/0.2	1.4/1.2	0.83	39.8	48X	3/3	0.4/0.2	0.9/0.5	1.10	52.8	48X
$ibmpg6$	3/3	0.6/0.2	1.0/0.2	1.15	63.3	55X	4/4	0.5/0.2	0.0/0.2	1.58	81.6	51X

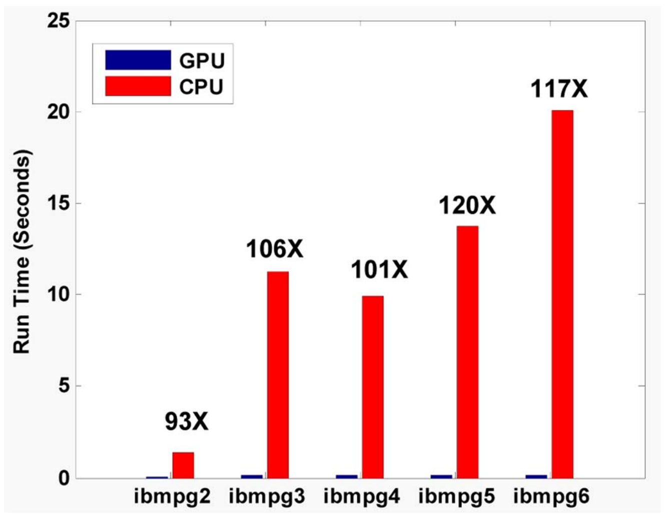

Fig. 13. Runtime of 1 K relaxations on CPU and GPU.

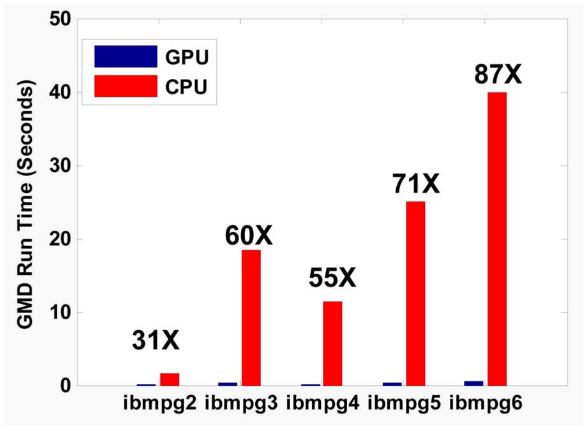

Fig. 14. Runtime of multigrid solvers on CPU and GPU.

conjugate gradient (MGPCG) method. All the algorithms are implemented using C++ and the GPU programming interface CUDA [1]. The hardware platform is a Linux PC with Intel Core 2 Quad CPU running at 2.66 GHz clock frequency and two NVIDIA Geforce 9800 GX2 cards (including four GPUs and each of them has a similar performance as Geforce 8800 GTX GPU).

A. Block Size Selection

As explained in Section V-C, GPU memory access (read/ write) latency can be dominant if the algorithm is not well implemented. When the block size is 4×4, for each choice of the

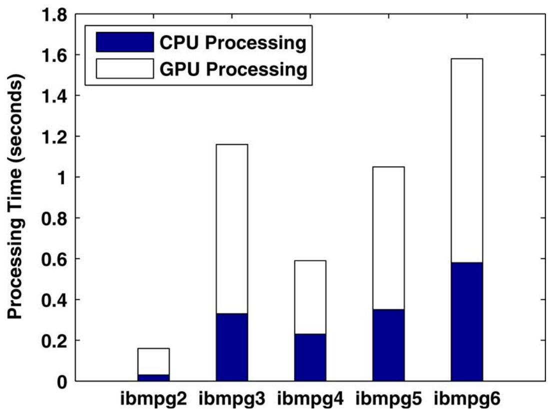

Fig. 15. Runtime composition of GPU-based multigrid solver.

local Jacobi iteration (LBJ) number k, the number of global iterations is empirically determined by $100/k$. The ratio of the pure GPU computing time T_c over the total GPU runtime T_t (computing time + memory read/write time) for each industrial benchmark circuit is shown in Fig. 11. From Fig. 11, we observe that the pure computation time T_c can only be a fraction (15% to 60%) of the total runtime T_t, while using more local LBJ iterations (larger k) can better hide the memory access latency. However, it is less useful to do excessive local iterations, since they may not help the convergence of the overall multigrid solving. Therefore, the number of local iterations (k) should be selected to trade off between the relaxation runtime (GPU computing efficiency) and global convergence rate (multigrid algorithm efficiency). We suggest $k = 10$ for the thread block size 4×4, and $k = 20$ for the thread block size 8×8 in practice.

B. DC Analysis Results

We list information of the industrial power grid designs in Table II, showing the sizes of the original irregular power grids, the sizes of the 2-D regularized grids that are obtained from the original power grids as discussed in Section III, the number of multigrid levels, the number of multigrid V-cycles, as well as the power grid solution range which is defined by the difference of the maximum and minimum node voltages.

1) HMD Results: The multigrid solvers are terminated when the residue is smaller than 10^{-4} of the original residual. Comprehensive results of the GPU-based multigrid solver for all the industrial power grid designs are shown in Table III. The results for VDD nets and GND nets are displayed as VDD/GND. In Fig. 12 we compare the solution of $ibmpg5$ circuit with the results obtained by GpuHMD solver (using one HMD iteration),

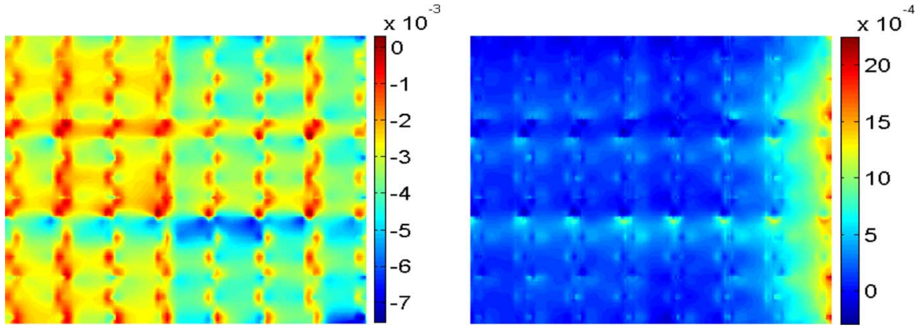

Fig. 16. Spatial error distributions after the (left) first and (right) second HMD iterations.

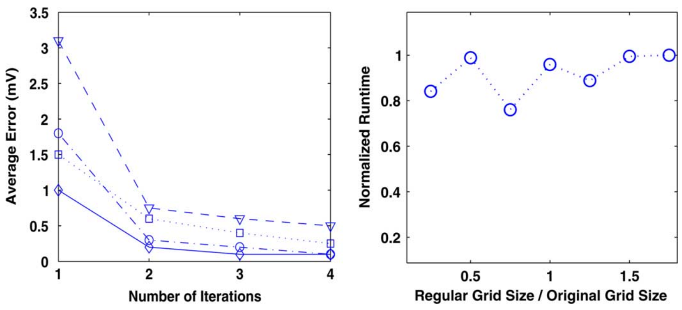

Fig. 17. (left) Average error versus HMD iteration count. (right) Regular grid size versus runtime.

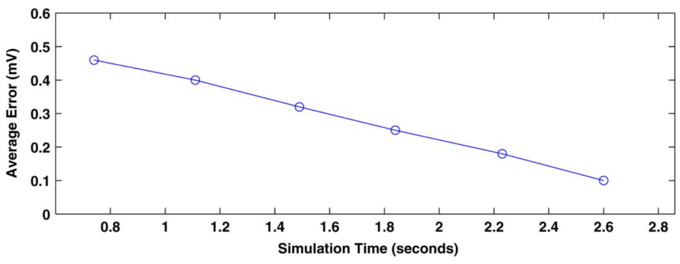

Fig. 18. Average error versus GPU-based HMD solving time for power grid ibmpg6.

showing that even after the first HMD iteration, a pretty accurate voltage distribution can be obtained on GPU. Additionally, in Table III, we show the runtime/accuracy results when using different numbers of HMD iterations. As observed, using one more iteration, the accuracy can be improved significantly. For most benchmarks, GpuHMD produces a less than 0.5 mV average node voltage error.

The following insightful experiments are also conducted. 1000 relaxations are run on both the CPU and GPU. As shown in Fig. 13, the GPU based computation achieves 93× to 117× speedups over its CPU counterpart. The runtimes of the multi-V-cycle multigrid operations are also compared between GPU and CPU. As shown in Fig. 14, our GPU implementation achieves roughly 31× speedup for small grid and 87× speedup for large grids. In Fig. 15, the runtimes of HMD iterations on CPU and GPU have been shown, where the GPU runtime takes more than 60% of the total runtime.

2) Errors of the HMD Iterations: To see the convergence behavior of the proposed HMD iterations, the spatial node voltage error distributions of the VDD net of a power grid design are shown in Fig. 16. The errors decrease drastically after two iterations, indicating a very fast convergence of HMD iterations. The average error versus the number of HMD iterations is shown for a few large power grid designs in Fig. 17 (left). The average errors of all four circuits can be damped very quickly after two or three HMD iterations. In Fig. 18, we also show the average error versus runtime plot of the GPU-based multigrid solver on a large power grid design, where the errors can be reduced quickly by increasing the number of multigrid iterations.

In Fig. 17 (right), the dependency of the total HMD runtime on the regular grid size is shown for a power grid design. When the coarse grid size is varied from 20% to 150% of the original grid size, the GPU-based HMD runtime does not vary significantly under the same accuracy tolerance. This indicates that the 2-D regularized grid (see Section III) is not required to be very accurate compared with the original grid. A reasonably 2-D regularized grid approximation is sufficient for fast HMD convergence.

C. Scalability of Multigrid Solvers

Fig. 19 shows the runtime and memory comparisons of GPU-based multigrid solver and the direct solver [9] for several large synthetic power grids (similar to the ones depicted in Table II). The multigrid solver is run on our four-core-four-GPU machine (one CPU and one GPU are used) while the direct solver is run on a more powerful computer (8-core Intel Xeon@2.33 GHz with 8G RAM running 64-bit Linux). The runtime and memory consumption of the multigrid solver increase linearly as the grid size increases: the 1-threading (8-threading) direct solver typically runs 100× (20×) slower and takes 20× more memory resources. The running times of the multigrid solver for solving very large grids are plotted in Fig. 20, where the thirty-million grid nodes have been solved in 8 s. Our GPU-based multigrid algorithm scales favorably with the circuit complexity, at a constant rate about one second (runtime) and 100 Mb (memory) per two million nodes.

D. Multi-Core-Multi-GPU Results

In previous section, we mentioned how to utilize the multi-core computers with multi-GPU cards to further accelerate the multigrid solver. Each CPU-GPU pair works on a smaller partition of the original grid, while the residual computation and smoothing steps are performed on the full grid. In this section, we show the results of solving large 2-D regular grids on

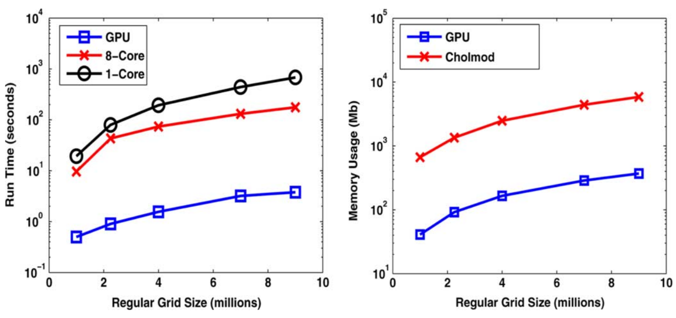

Fig. 19. (left) Runtime and (right) memory scalability of GPU-based multigrid solver compared with parallelized direct solver (CHOLMOD [9]).

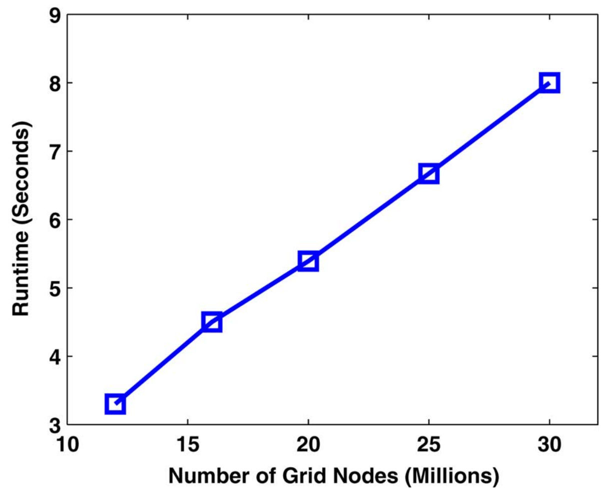

Fig. 20. Runtime of GPU-based multigrid solver for solving very large grids.

TABLE IV
GPU-Based Multigrid Results for Multi-Core-Multi-GPU System. Size Is Number of Nodes of the 2-D Regular Grid, While T_n Is the Runtime of the Multigrid Solver on N-Core-N-Gpu System. T_{Chol-n} Is the Runtime of the Direct Solver (CHOLMOD) Running on the N-Core CPU. Spd. Is Speedup of Four-GPU Multigrid Solver Over the Eight-Core Direct Solver. All the Runtime Results Are Shown in Seconds

$Size$	T_1	T_2	T_3	T_4	T_{Chol-1}	T_{Chol-8}	$Spd.$
$4M$	1.7	1.1	0.75	0.56	194.2	73.7	$132X$
$8M$	3.5	2.1	1.35	1.1	561.4	154.3	$140X$

multi-core-multi-GPU system. The synthetic 2-D regular grids are split into smaller partitions with similar sizes based on the grid geometries to well balance the workload. The runtime results of multigrid solver using different numbers of GPUs and CPUs are shown in Table IV, where it is observed that when using four-GPU system we can achieve up to 140× speedups over the 8-core direct solver.

E. Multigrid Preconditioned Conjugate Gradient Method

We demonstrate the results of the GPU-based multigrid preconditioning technique for conjugate gradient method (MGPCG) in Table V. From the experiments we can find that the numbers of MGPCG iterations are almost constant, not varying much with the circuit sizes, while the traditional CG solver for larger circuits may take 800× more iterations to

TABLE V
Results of Multigrid Preconditioned Conjugate Gradient Method (See Section VII). Size Is the Number of Nodes of the Circuit, $N_{\rm CG}/N_{\rm HMD}/N_{\rm PCG}$ Is the Number of Iterations Using the CG/HMD/PCG Method, $T_{\rm CG}/T_{\rm PCG}/T_{\rm HMD}$ Is the Runtime of the CG/PCG/HMD Solver

ckt	$Size$	N_{CG}	N_{HMD}/N_{PCG}	T_{CG}	T_{PCG}	T_{HMD}
$C1$	$65,229$	$1,790$	$5/3$	2.0	0.08	0.08
$C2$	$440,616$	$4,834$	$7/4$	54.3	0.60	0.78
$C3$	$478,059$	$2,253$	$3/2$	54.3	0.48	0.42
$C4$	$581,473$	$4,062$	$5/3$	53.7	0.56	0.65
$C5$	$862,418$	$6,433$	$6/4$	152.6	1.10	1.21

converge. Our undergoing research project shows that by accelerating both the sparse matrix-vector multiplication kernel and the multigrid solver on GPU, a significant amount of further speedups can be achieved in the future.

It should be emphasized that our multigrid preconditioning technique can be efficiently realized for the massively parallel computing platform, while the other preconditioning techniques that are based on incomplete matrix factorizations can not be efficiently applied to the GPU computing platform.

IX. Conclusion

In this work, we address the challenge of large-scale power grid analysis by developing a novel multi-core-multi-GPU acceleration engine. To properly exploit the massively parallel SIMT GPU architecture, a parallel multigrid algorithm is specially designed. To gain good efficiency on GPUs, we propose a hybrid multigrid (HMD) analysis framework to handle the original power grid on CPU, and a series of 2-D regularized coarser grids on GPU, so as to eliminate most of GPU random memory access patterns and simplify control flows. Novel coarse grid construction and block smoothing strategies are adopted to suit the SIMT GPU platform. The robustness of the algorithm is further enhanced by an efficient multigrid preconditioned conjugate gradient method. Careful performance fine tuning is conducted to gain good analysis efficiency on the GPU. Extensive experiments have shown that the DC analysis accelerated on a single-core-single-GPU system can achieve 100× runtime speedups over a state-of-art direct solver and 50× speedups over the CPU based multigrid solver. It is also demonstrated that when utilizing a four-core-four-GPU system, a grid with eight million nodes can be solved within about 1 s.

References

[1] Nvidia Corporation, Santa Clara, CA, "NVIDIA CUDA programming guide," 2007. [Online]. Available: http://www.nvidia.com/object/cuda.html

[2] IBM Austin Research Laboratory, Austin, TX, "IBM power grid benchmarks," 2008. [Online]. Available: http://dropzone.tamu.edu/pli/PGBench/

[3] S. F. Ashby and R. D. Falgout, "A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations," *Nucl. Sci. Eng.*, vol. 124(1), pp. 145–159, 1996.

[4] B. Barney, "POSIX threads programming," Lawrence Livermore National Laboratory, Livermore, CA, 2008. [Online]. Available: www.llnl.gov/computing/tutorials/pthreads/

[5] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder, "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 917–924, 2003.

[6] W. Briggs, *A Multigrid Tutorial*. Warrendale, PA: SIAM, 1987.

[7] L. Buatois, G. Caumon, and B. Levy, "Concurrent number cruncher: An efficient sparse linear solver on the GPU," in *Proc. HPCC, LNCS*, 2008, pp. 358–371.

[8] 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*, 2001, pp. 559–562.

[9] T. Davis, "CHOLMOD: Sparse supernodal cholesky factorization and update/downdate," 2008. [Online]. Available: http://www.cise.ufl.edu/research/sparse/cholmod/

[10] Y. Deng, B. Wang, and S. Mu, "Taming irregular EDA applications on GPUs," in *Proc. IEEE/ACM ICCAD*, 2009, pp. 539–546.

[11] Z. Feng and P. Li, "Multigrid on GPU: Tackling power grid analysis on parallel SIMT platforms," in *Proc. IEEE/ACM ICCAD*, 2008, pp. 647–654.

[12] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, "LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware," in *Proc. ACM SC*, 2005, vol. 22, no. 3, pp. 917–924.

[13] V. Henson and U. Yang, "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Appl. Numer. Math.*, pp. 155–177, 2002.

[14] J. N. Kozhaya, S. R. Nassif, and F. N. Najm, "A multigrid-like technique for power grid analysis," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 21, no. 10, pp. 1148–1160, Oct. 2002.

[15] Y. Liu and J. Hu, "GPU-based parallelization for fast circuit optimization," in *Proc. IEEE/ACM DAC*, 2009, pp. 943–946.

[16] S. R. Nassif, "Power grid analysis benchmarks," in *Proc. IEEE/ACM ASPDAC*, 2008, pp. 376–381.

[17] H. Qian, S. R. Nassif, and S. S. Sapatnekar, "Power grid analysis using random walks," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 24, no. 8, pp. 1204–1224, Aug. 2002.

[18] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. ACM PPOPP*, 2008, pp. 73–82.

[19] H. Su, E. Acar, and S. R. Nassif, "Power grid reduction based on algebraic multigrid principles," in *Proc. IEEE/ACM DAC*, 2003, pp. 109–112.

[20] K. Sun, Q. Zhou, K. Mohanram, and D. C. Sorensen, "Parallel domain decomposition for simulation of large-scale power grids," in *Proc. IEEE/ACM ICCAD*, 2007, pp. 54–59.

[21] K. Wang and M. M. Sadowska, "On-chip power-supply network optimization using multigrid-based technique," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 24, no. 3, pp. 407–417, Mar. 2005.

[22] M. Zhao, R. Panda, S. S. Sapatnekar, and D. T. Blaauw, "Hierarchical analysis of power distribution networks," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 21, no. 2, pp. 159–168, Feb. 2002.

[23] Y. Zhong and M. D. F. Wong, "Fast algorithms for IR drop analysis in large power grid," in *Proc. IEEE/ACM ICCAD*, 2005, pp. 351–357.

[24] C. Zhuo, J. Hu, M. Zhao, and K. Chen, "Power grid analysis and optimization using algebraic multigrid," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 27, no. 4, pp. 738–751, Apr. 2008.

Zhuo Feng (S'03–M'10) received the B.Eng. degree in information engineering from Xi'an Jiaotong University, Xi'an, China, in 2003, the M.Eng. degree in electrical engineering from National University of Singapore, Singapore, in 2005, and the Ph.D. degree in electrical and computer engineering from Texas A&M University, College Station, in 2009.

Since July 2009, he has been an Assistant Professor with the Department of Electrical and Computer Engineering, Michigan Technological University, Houghton. His research interests include VLSI computer-aided design and high performance computing on emerging parallel computing platforms. His past and present work has focused on parallel circuit simulation and parasitics modeling techniques, GPU computing, simulation and optimization of large-scale power delivery networks (PDNs), interconnect modeling, and statistical circuit modeling and analysis.

Dr. Feng served on the technical program committee of the International Symposium on Quality Electronic Design (ISQED) in 2009 and 2010. He was a recipient of two IEEE/ACM William J. McCalla ICCAD Best Paper Award Nominations in 2006 and 2008.

Zhiyu Zeng (S'10) received the B.S. degree in electronic and information engineering from Zhejiang University, Hangzhou, China, in 2006. He is currently pursuing the Ph.D. degree from the Department of Electrical and Computer Engineering, Texas A&M University.

His research interests include parallel simulation, optimization and verification of power grids, design and optimization for on-chip voltage regulation, and general parallel computing using GPU platforms.

Peng Li (S'02–M'04–SM'09) received the B.Eng. degree in information engineering and the M.Eng. degree in systems engineering from Xian Jiaotong University, Xian, China, in 1994 and 1997, respectively, and the Ph.D. degree in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, in 2003.

Since August 2004, he has been an Assistant Professor with the Department of Electrical and Computer Engineering, Texas A&M University, College Station. His current research interests include general areas of VLSI systems, design automation, and parallel computing. His past and present work has focused on analog and mixed-signal CAD and testing, circuit simulation, design and analysis of power and clock distributions, interconnect modeling, statistical circuit design, parallel algorithms, and implementations for solving large-scale computing problems in VLSI computer-aided design and emerging applications.

Dr. Li is the recipient of various awards, including two Design Automation Conference Best Paper Awards in 2003 and 2008, two Semiconductor Research Corporation Inventor Recognition Awards in 2001 and 2004, the MARCO Inventor Recognition Award in 2006, the National Science Foundation CAREER Award in 2008, and the ECE Outstanding Professor Award from Texas A&M University in 2008. He is an Associate Editor for the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS and the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—PART II: EXPRESS BRIEFS. He has been on the committees of many international conferences and workshops, including ICCAD, ISQED, ISCAS, TAU, and VLSI-DAT, as well as the selection committee for the ACM Outstanding Ph.D. Dissertation Award in Electronic Design Automation. He has served as the Technical Program Committee Chair for the ACM TAU Workshop in 2009 and the General Chair for the 2010 Workshop.