Multigrid on GPU: Tackling Power Grid Analysis on Parallel SIMT Platforms

Zhuo Feng and Peng Li
Department of Electrical and Computer Engineering
Texas A&M University, College Station, TX 77843
Email: {fengzhuo, pli}@neo.tamu.edu

***Abstract*—The challenging task of analyzing on-chip power (ground) distribution networks with multi-million 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 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 algorithm, GpuHMD, and its implementation. In particular, a proper interplay between algorithm design and SIMT architecture consideration is shown to be essential to achieve good runtime performance. Different from the standard CPU based CAD development, care must be taken to balance between computing and memory access, reduce random memory access patterns and simplify flow control to achieve efficiency on the GPU platform. Extensive experiments on industrial and synthetic benchmarks have shown that the proposed GpuHMD engine can achieve 100X runtime speedup over a state-of-the-art direct solver and be more than 15X faster than the CPU based multigrid implementation. The DC analysis of a 1.6 million-node industrial power grid benchmark can be accurately solved in three seconds with less than 50MB memory on a commodity GPU. It is observed that the proposed approach scales favorably with the circuit complexity, at a rate about one second per million nodes.**

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 CPU platform, a body of power grid analysis methods have been proposed [1], [2], [3], [4], [5], [6], [7], [8] with various tradeoffs. Recently, the emergence of massively parallel single-instruction multiple-data (SIMD), or more precisely, single-instruction multiple-thread (SIMT) [9], based 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 86GB/s off-chip memory bandwidth, which are 3-4X greater than offered by modern day general-purpose quad-core microprocessors [9]. The ongoing GPU performance scaling trend justifies the development of a suitable subset of CAD applications on such platform.

However, converting the impressive theoretical GPU computing power to usable design productivity can be rather nontrivial. Deeply rooted in graphics applications, the GPU architecture is designed to deliver high-performance for data parallelism parallel computing. Except for straightforward general-purpose SIMD tasks such as parallel table lookups, rethinking and re-engineering 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 in the proposed GPU-accelerated hybrid multigrid algorithm GpuHMD 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, GpuHMD is the first reported GPU based power grid analysis tool.

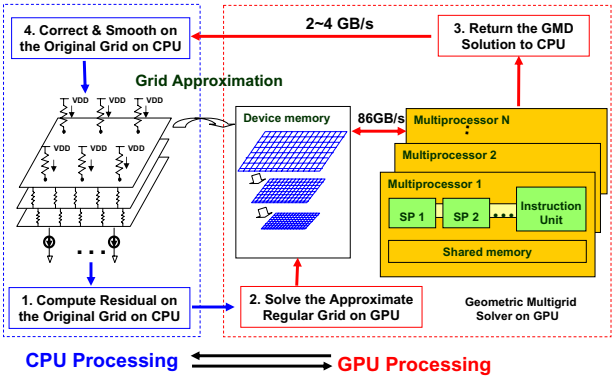

Fig. 1. Overall GpuHMD analysis flow.

As shown in Fig. 1, GpuHMD is built upon a custom geometric multigrid (MG) algorithm as opposed to a direct solver. Despite the attempts to develop general-purpose direct matrix solvers on GPUs [10], so far the progress has been limited for large sparse problems due to the very natures of GPU such as the inefficiency in handling random complex data structures and memory access. Being a multi-level 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 geometric multigrid, a topology regularization step is taken to convert a possibly irregular 3D grid into a regular 2D structure, thereby significantly reducing random memory access and thread divergence. New coarse grid construction, block smoothing strategies, restriction and prolongation operators are developed geometrically, maintaining the desirable regularity throughout the entire multigrid hierarchy. The proposed GpuHMD is referred to as a hybrid approach in the sense that the entire workload is split between the host (or CPU) and the GPU. The multigrid hierarchy is purposely made deep such that more than 95% of the work is pushed onto the GPU. Only a minimum of small matrix solve, residue computation and smoothing operation is conducted on the CPU. To remove the possible small error caused by topology regularization, a few iterative steps between the CPU and GPU may be performed. Through in-depth theoretical analysis and empirical data, we show that in practice the required number of CPU-GPU iterations is typically small and accurate power grid solutions converge fast.

In this paper, we focus only on the DC analysis of power grids, however, the same framework can be rather straightforwardly extended to transient analysis. Extensive experiments have shown the promising potential of GpuHMD: it is typically more than 100X faster than the state-of-art direct methods [11] on PC and 15X faster than the CPU-based multigrid implementation. We envision that with

978-1-4244-2820-5/08/$25.00 ©2008 IEEE

the future GPU performance improvement and the use of multiple GPU card systems, network analyses that were impossible in the past may become feasible, leading to a new level of verification and design. For instance, it may be possible to facilitate the analysis of large interconnect dominated nonlinear networks (e.g., power grids and mesh circuits coupled with devices), where the dominant linear portion of the problem is efficiently solved in a GpuHMD-like fashion.

II. Background and Overview

We first review the power grid analysis problems and the GPU architecture. Next, an overview of the proposed GpuHMD approach is provided.

A. Power grid analysis

At the heart of either DC or transient power grid analysis lies the solution of certain large matrix problems. For instance, a system of linear equations are formulated in the DC analysis [2]:

$$GV = I, \tag{1}$$

where when appropriately formulated, G is a symmetric positive definite matrix representing the interconnected resistors; V is the unknown vector of node voltages; and I is a vector of independent current sources. The feasible solution of such large linear systems with tens or even hundreds of millions of unknowns, as seen in today's industrial designs, are hampered by the excessive runtime and memory usage required.

B. GPU matrix solvers?

A basic understanding of the SIMT GPU architecture is instrumental for evaluating the potential in applying GPU matrix solvers to large power grid problems. Consider a recent commodity GPU model, NIVIDIA G80 series. Each card has 16 streaming multiprocessors (SMs) with each SM containing eight streaming processors (SPs) running at 1.35GHz. An SP operates in single-instruction, multiple-thread (SIMT) fashion and has a 32-bit, single-precision floating-point, multiply-add arithmetic unit [12]. 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 86GB/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 [9]:

1) Low control flow overhead: execute the same computation on many data elements in parallel;
2) High SP floating point arithmetic intensity: perform as many as possible calculations per memory access;
3) Minimum random memory access: pack data for coalesced memory access.

Due to the very nature of the SIMT architecture, it remains as a challenge to implement efficient general-purpose sparse matrix solvers on GPU. In recent such attempts, it is reported that most of runtime is spent on data fetching and writing, but not on data processing [13], [14]. For instance, traditional iterative methods such as conjugate gradient and multigrid [13] involve many sparse matrix-vector computations, leading to rather 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. 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 [12]. 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 [1], [2], [4] on GPU.

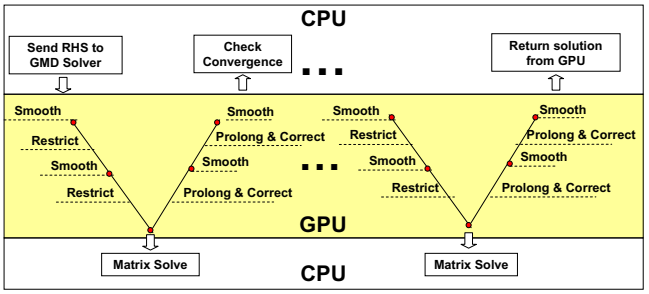

Fig. 2. Acceleration of GMD solve on GPU.

C. Our approach

1) Power grid uniformity: To achieve the best analysis efficiency on SIMT platforms, understanding the physical properties of practical power grid designs is critical. It can be expected that if the power grid can be stored and processed like pixel graphics, the GPU SIMT platform can be of a significant advantage over the general purpose CPU platform. Not surprisingly, after examining a set of published industrial power grids [15], [16], we have found that real-life designs have a high degree of global uniformity while exhibiting some local irregularity. Therefore, to maintain regularity on GPU, it is very natural for us to consider solving an approximate regular power grid that is close to the original grid. However, this brings up the need for developing "regular" numerical methods and correction schemes to guarantee solution accuracy.

2) GPU based geometric multigrid method: Mutigrid methods are among the fastest numerical algorithms for solving large PDE-like problems [17], where a hierarchy of exact to coarse replicas (e.g. fine vs. coarse grids) of the given linear problem are created. Via iterative updates, the high and low frequency components of the solution error are quickly damped on the fine and coarse grids, respectively, contributing to the efficiency of multigrid. When properly designed, multigrid methods can achieve a linear complexity in the number of unknowns. The hierarchical iterative nature of multigrid is attractive to GPU platforms since the GPU on-chip shared memory is rather limited. 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 multigrid method include:

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 coarse grid);
3) *Prolongation:* mapping from a coarse grid to the next finer grid (applied to map the coarse grid solution to the fine grid);
4) *Correction:* use the mapped coarse grid solution to correct the fine grid solution.

On the k-th level grid with an initial solution of v_k, a typical multigrid cycle $MG(k, v_k)$ has the following steps [17]:

1) Apply pre-smoothing to update the solution;
2) Compute the residue on the k-th grid and map it to the $k+1$-th coarser grid via restriction;

3) Using the mapped residue to solve the $k+1$-th grid directly if the coarsest level is reached, otherwise apply a multigrid cycle $MG(k+1, v_{k+1})$ with a zero initial guess $v_{k+1} = 0$;
4) Map the solution v_{k+1} to the k-th grid via prolongation, and correct the solution v_k by adding v_{k+1};
5) Apply post-smoothing to further improve v_k at the k-th level grid and return the final v_k.

A GPU-specific GMD method is developed in our approach. Starting from a regularized power grid, all the key components of multigrid are realized in a *geometrically regular* fashion across the entire multigrid hierarchy, leading to simple flow controls and highly regular memory access patterns, favoring the GPU implementation.

3) Hybrid multigrid (HMD) iterations: The approximate regular power grid is solved efficiently using our custom GMD method on GPU (Fig. 2), where no explicit sparse matrix-vector operations are needed. The work associated with the GMD constitutes the dominant workload of the entire GpuHMD approach. To guarantee the accuracy of the final power grid solution, we further apply HMD iterations between the GPU and host to remove any error that may arise from only solving the approximate regular grid. Denote the true (original) power grid by $Grid_O$ and the regularized grid by $Grid_R$, HMD iterations involve the following main steps (Fig. 1):

1) (CPU:) Compute the residue of the current solution on $Grid_O$ and map the residue to $Grid_R$;
2) (GPU:) Solve the $Grid_R$ problem under the mapped residue using GMD and return the solution to $Grid_O$;
3) (CPU:) Update the $Grid_O$ solution using the GPU result and apply additional smoothing;
4) (CPU:) If the solution error is small enough, exit; otherwise repeat the above steps.

The bulk workload of the entire GpuHMD approach is done on GPU via solving the regular grid (step-2). Only a fraction of the work such as simple residue computation and smooth steps is preformed on the host, where the general-purpose CPU is more efficient in terms of handling the original (irregular) power grid.

III. Regular Grid Approximation

We discuss several key issues in converting a three-dimensional irregular power grid to a two-dimensional regular approximation that can be processed efficiently on GPU.

A. Mapping to a regular grid

The goal is to map the original 3D irregular power grid to a 2D regular one such that the electrical property of the original grid can be well preserved. As such, the regular grid solution can be very close to the true solution, reducing the number of the GPU-CPU HMD iterations required.

The mapping procedure has two subsequent steps: 3D irregular to 2D irregular, and 2D irregular to 2D regular mappings. First, by neglecting via resistances, all the metal layers in the network are overlapped on the same 2D plane, forming a collapsed 2D irregular grid. By analyzing industrial power grid benchmarks, we found that neglecting via resistances typically does not alter the circuit solution in any significant way. Nevertheless, the error induced can be corrected through the HMD iterations. Then, by examining the pitches in the collapsed 2D irregular grid, a fixed uniform pitch is chosen for the X and Y directions for the final 2D regular grid, on which all the circuit elements are mapped to. Consider the simple example in Fig. 3, where a two-metal-layer irregular grid is mapped to a single-layer regular grid. The conductance values on the regular grid can be obtained as follows:

$$G_1 = 2g_{31} + g_{21}, G_2 = 2g_{31} + g_{22}, G_3 = 2g_{32}, \\ G_4 = 2g_{32} + g_{23}, G_5 = g_{33} + g_{24}. \quad (2)$$

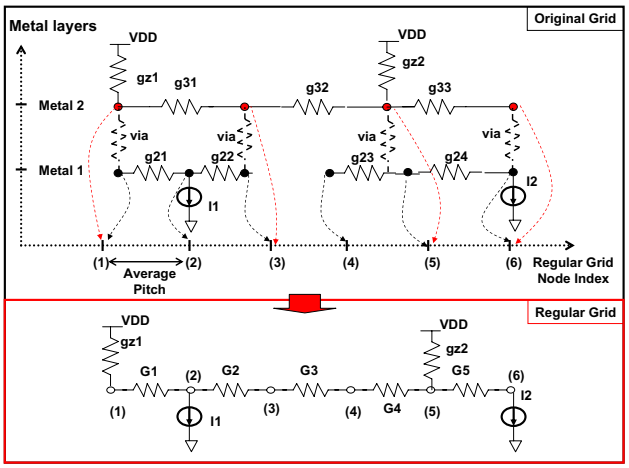

Fig. 3. Cross section view of mapping a two-layer irregular grid to a single-layer regular grid.

Note that because of irregularity of the original grid, some of the regular grid nodes may not correspond to any of the original nodes. In this case, small dummy conductances ($G_{min} = 1e-6$) are inserted between such regular grid node and its neighboring nodes. Note also that the uniform pitches of the regular grid may be set to the averaged pitch values in the irregular grid and can be adjusted when appropriate. Smaller uniform pitch values lead to increased regular grid size and improved grid approximation. The possible grid size increase in the regular grid does not significantly impact the overall runtime efficiency of our approach due to the linear complexity of the GPU GMD solver. The improved grid approximation, however, may contribute to faster HMD convergence. As will be demonstrated later, both the accuracy and efficiency of our GpuHMD algorithm are not sensitive to the regular grid size. This is the case even when the regular grid size is varied from 50% to 150% of the original grid size.

Algorithm 1 3D irregular-to-2D-regular grid mapping

Input: The original power grid netlist.
Output: The regular grid netlist consists of all the elements of G_h, G_v, G_z, I_z with their table indices.

1: Extract the horizontal and vertical node pitches from the netlist and compute the average pitches δX and δY;
2: For each circuit element except via resistors:
  a) Extract their node locations x_i and y_i;
  b) Compute their regular grid indices by:
    $I_{x_i} = floor[(x_i - x_{min})/\delta X]$, $I_{y_i} = floor[(y_i - y_{min})/\delta Y]$;
  c) Stamp the conductance values into 2D table based storage.

B. Table-based representation of the regular grid

The $2D$ regular grid is represented by several tables, denoted by G_h, G_v, G_z and I_z. The simple representation allows for efficient coalesced memory access to the device memory and is shown to be critical to the GPU implementation. For a regular grid node $N[i, j]$, the following four tables are adopted:

$G_h[i, j]$: Horizontally connected conductance between node $N[i, j]$ and node $N[i+1, j]$;
$G_v[i, j]$: 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 sources that flows out node $N[i,j]$.

The mapping procedure is summarized in Algorithm 1.

IV. Geometric Multigrid on GPU

While the 2D regular grid can be obtained in a relatively straightforward manner, developing an efficient regular grid solver on GPU is non-trivial. Naive implementations for either data transferring or processing can lead to severe performance degradation. The proposed GPU based GMD solver is described by covering the key issues concerning the discussion in Section II-B.

A. Coarse grid generation and inter-grid operators

With the mapped regular 2D grid sitting at the bottom of the multigrid hierarchy, a set of increasingly coarser grids shall be created to occupy the higher levels. In this case, the regular grid produced by the previous mapping step serves as the finest grid in our GMD method. 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, it is critical to carry the regularity of the finest grid 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) vs. those connecting a grid node to one of its four neighboring nodes (or internal resistances) on the regular grid, as shown in Fig. 4. 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 pullup and pulldown strengths are kept in the coarser grid of a power distribution network. Denote the voltages of M grid nodes that connect to an ideal voltage source via a wire resistance by V_i for $i = 1, ..., M$, and the N loading current sources by I_j for $j = 1, ..., N$. The following equation holds:

$$\sum_{i=1}^{M} (VDD - V_i)\, G_{z_i} = \sum_{j=1}^{N} I_{z_j}. \tag{3}$$

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. 4, 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.

Use H and h to indicate the fine and coarser grid components, respectively, the coarser grid is created as follows:

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

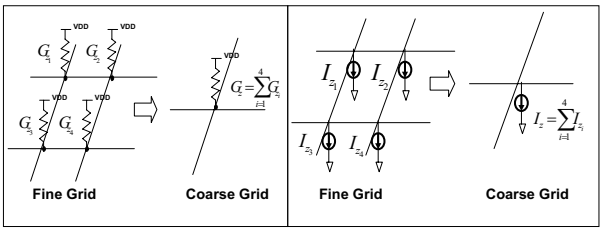

Fig. 4. VDD pads (G_z) and current sources (residues) in fine/coarse grids.

where i and j denote grid locations, and the numbers of nodes along the horizontal and vertical directions are reduced by a factor of two in the coarser grid. The restriction and prolongation operators are:

$$\begin{aligned}
&R^h[i,j] = R^H[2i,2j] + R^H[2i+1,2j] + \\
&R^H[2i,2j+1] + R^H[2i+1,2j+1],
\end{aligned} \tag{5}$$

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

where residues and errors (solution corrections) are denoted by R and E, respectively. Apparently, the coarser grid problem is defined completely based on geometry and can be stored in the same regular table-based representation. In our GMD implementation, the coarsest grid is solved via a direct method on the host. To reduce the overhead of this sparse matrix solve on CPU and fully utilize the GPU computing power, the GMD hierarchy is purposely made deep. In our implementation, four to five grid levels are used, making the size of the coarsest problem vary from a few hundred to a few thousand times smaller than the finest grid. This choice may push, say 95%, of the overall computation onto the GPU.

B. Point vs. block smoothers

The choice of smoother is critical in GMD. Typically, point Gauss-Seidel or weighted Jacobi smoothers are used for CPU based GMD methods However, a block based smoother is adopted in our approach to fully utilize the SIMT GPU computer power. On GPU, a number (more precisely a warp [9]) of threads may be simultaneously executed in a single-instruction multiple-data fashion on a multiprocessor. This implies that multiple circuit nodes can be processed in the smoothing step at the same time. In our approach, a block of circuit nodes are loaded into a multiprocessor at a time. Then, multiple treads are launched to simultaneously smooth the circuit nodes in the block for a number of iterations. As a result, such processing step (almost) completely solves the circuit block, effectively leading to a block smoother. This approach ensures that a meaningful amount of compute work is done before the data 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 is discussed in detail in Section V.

V. Accelerating GMD 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 algorithms.

A. Thread organization

Through a suitable programming model (e.g. CUDA [9]), threads shall be packed properly for efficient execution on multiprocessors. On a multiprocessor, threads are organized in units of blocks, where the number of blocks should be properly chosen to maximize the performance. The optimal block size shall be multiples of 32 threads

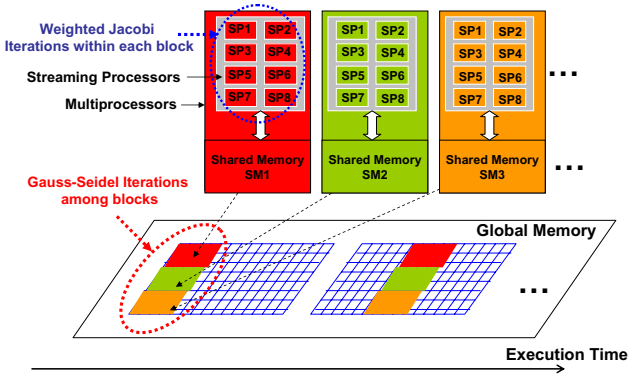

Fig. 5. Mixed block relaxation (smoother) on GPU.

for a commercial GPU [9]. In our implementation, the actual optimal block size is chosen experimentally.

B. Memory and register allocation

Before the GMD solve starts on GPU, $1D$ tables are allocated on the CPU to store all the regular grids in the multigrid hierarchy. Then, the data are transferred to the device (CPU). We bind the conductance tables (G_h, G_v and G_z) to the texture memory and other data to the on-board GPU device memory. Texture memory is cached, so its access latency is significantly smaller than the device memory. However, the texture memory is read-only and cannot be used for solution updates. Therefore, residues, solution and error vectors are stored in the device memory. Since the device memory is not cached, coalesced memory accesses are employed to achieve the best memory bandwidth.

The fast on-chip shared memory and registers are very limited resources on GPU. If the required shared memory and registers exceed what are available, an application will fail. On the other hand, more than one block of threads should be run on the same stream multiprocessor (SM). This will hide the memory read/write latency in a better way, leading to a much higher performance throughput. With this in mind, all components of our GPU GMD method are developed carefully to fully utilize GPU resources. As an example, in the smoothing steps, the solution and right hand side (RHS) vectors are loaded from the global memory to the shared memory, while the resistance grid data are loaded from the the texture memory to the registers. The above scheme allows more than two blocks of threads to be launched concurrently within the resource limitation on an SM. Otherwise, if the grid data were loaded to the shared memory, only one block of threads could be run, making the memory access latency a higher impact.

C. Mixed block-wise smoother

In our GMD solver, the relaxation (smoothing) steps dominate the overall computation. Hence, an efficient implementation of the smoother is critical. On CPU, point-wise iterative methods such as Gauss-Seidel or weighted Jacobi are often adopted. However, to improve the arithmetic intensity and work better with efficient coalesced (block) memory accesses and control flows, *global block Gauss-Seidel iteration* (GBG iteration) and *local block weighted Jacobi iteration* (LBJ iteration) schemes are introduced.

As illustrated in Fig. 5, during each GBG iteration, the whole $2D$ regular grid is partitioned into small blocks which are subsequently transferred to streaming processors. Next, k times LBJ iterations are conducted within each block locally. Since only the threads within the same thread block can share the data with each other, the solution of this local block can not be shared by others unless it is sent back to the global memory. As processed block solutions are written back to the global memory, the smoothing of subsequent blocks will be based upon the most recent solutions of the neighboring blocks. Therefore, from this global point of view, the smoother is a block Gauss-Seidel iterative (or GBG) method. On the other hand, when each block is being smoothed, all its nodes are processed by multiple threads simultaneously in a weighted Jacobi fashion, referred to as LBJ iterations. The above mixed block-iteration scheme has been carefully tailored for our GPU based GMD engine, particulary through the following considerations:

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 may include more local iterations. However, excessive local iterations may not help the overall convergence since the boundary information is not updated.
2) To hide the memory latency and thread synchronization time, we allow two or more blocks to run concurrently on each multiprocessor to avoid idle processors during the thread synchronization and device memory access.

The block size may impact the overall performance significantly. A too large block size may 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 are observed to offer good performance.

D. Dummy grid nodes

As discussed before, GPU data processing favors block-like operations. If the grid dimensions are not multiples of the block size, extra handling is required. For example, assume one smoothing kernel of the GMD solver is executed on all multigrid levels based on 8×8 thread blocks. Then, all the grid widths and heights need to be modified to be multiples of the block size. To this end, certain dummy grids can be attached to the periphery of the original grid. It is important to isolate these dummy grids from the original grid, as shown in Fig. 6. Otherwise, the GMD convergence can be significantly impacted.

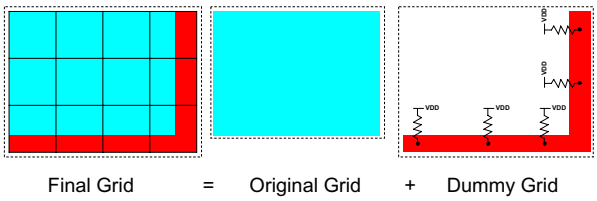

Fig. 6. Appending dummy grid nodes for a chosen block size.

VI. Hybrid Multigrid for Power Grid Analysis

Although solving the mapped 2D regular grids on GPU typically provides pretty accurate results, the solution quality may not be completely guaranteed since grid approximations can lead to various accuracy levels. To have a robust error control scheme, interactions between the 2D regular grid and the original 3D irregular grid are important. In this work, we propose a hybrid multigrid (HMD) analysis framework to iteratively correct the error components that are caused by grid approximation. The main steps of our HMD flow is shown in Fig. 1 and Fig. 7, and also outlined in Section II-C.

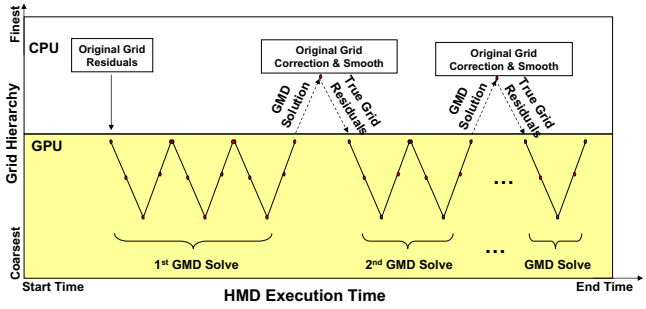

Fig. 7. Hybrid GPU-CPU multigrid iterations.

A. Problem formulation

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

$$A \cdot x = b, \tag{7}$$

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

$$r^{(k)} = b - A \cdot x^{(k)}, \quad r_r^{(k)} = V_o^r \cdot r^{(k)}, \tag{8}$$

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 regular grid using the GPU GMD method:

$$A_r \cdot e_r^{(k)} = r_r^{(k)}. \tag{9}$$

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

$$e^{(k)} = V_r^o \cdot e_r^{(k)}, \tag{10}$$

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 original grid is updated:

$$x^{(k+1)} = x^{(k)} + e^{(k)}. \tag{11}$$

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 ($Grid_O$ and $Grid_R$) mapping operators V_o^r and V_r^o may be interpreted as an 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.

B. Convergence analysis

Experimentally, it is observed that the proposed HMD approach can converge in a few iterations. To gain further insights on the converge property, the following theoretical result is proved.

Theorem 1: Denote the spectral radius of an $l \times l$ matrix M by $\rho(M)$, where $\rho(M) = max_{i=1,\cdots,l}|\lambda_i|$, λ_i is an eigenvalue of M. The HMD iteration converges to the true solution x^* for any chosen initial guess $x^{(0)}$ if and only if:

$$\rho\left(I - V_r^o \left(A_r\right)^{-1} V_o^r A\right) < 1. \tag{12}$$

Proof: At the k-th HMD iteration, the residue on $Grid_O$ ((8)) can be written as:

$$r^{(k)} = A\varepsilon^{(k)} = A\left[x^* - x^{(k)}\right], \tag{13}$$

where $\varepsilon^{(k)}$ is the solution error w.r.t. x^* and shall not be confused with $e^{(k)}$ in (10). Combining (8), (9) and (10) leads to

$$e^{(k)} = V_r^o A_r^{-1} V_o^r r^{(k)}. \tag{14}$$

From (11), (13) and (14), we have

$$\begin{aligned} x^{(k+1)} &= x^{(k)} + e^{(k)} \\ &= x^{(k)} + V_r^o A_r^{-1} V_o^r A\left[x^* - x^{(k)}\right]. \end{aligned} \tag{15}$$

Let $B = I - V_r^o A_r^{-1} V_o^r A$. Substituting the definitions $\varepsilon^{(k)} = x^* - x^{(k)}$ and $\varepsilon^{(k+1)} = x^* - x^{(k+1)}$ into (15), we have:

$$\begin{aligned} \varepsilon^{(k+1)} &= \left[I - V_r^o A_r^{-1} V_o^r A\right] \varepsilon^{(k)} \\ &= \left[I - V_r^o A_r^{-1} V_o^r A\right]^{k+1} \varepsilon^{(0)} \\ &= B^{k+1} \varepsilon^{(0)}. \end{aligned} \tag{16}$$

It is not difficult to see that for any $\varepsilon^{(0)}$ (or $x^{(0)}$), if $\rho(B) < 1$, $\varepsilon^{(k)}$ converges to a zero vector. Furthermore,

$$\left\|\varepsilon^{(k+1)}\right\| \leq \|B\|^{k+1} \left\|\varepsilon^{(0)}\right\| \geq \left[\rho(B)\right]^{k+1} \left\|\varepsilon^{(0)}\right\|. \tag{17}$$

It implies that if $\varepsilon^{(k)}$ converges to zero for any $\varepsilon^{(0)}$ (or $x^{(0)}$), it must be true that $\rho(B) < 1$. ■

Theorem 1 provides a very intuitive understanding of the convergence property of the HMD iterations and offers a theoretical basis to further improve the convergence rate. Let $C = V_r^o A_r^{-1} V_o^r$, $C \in \mathbb{R}^{n \times n}$. C can be interpreted as a linear operator ($\mathbb{R}^n \rightarrow \mathbb{R}^n$), which corresponds to the correction operator of the $Grid_O$ solution $x^{(k)}$ by solving an approximate problem defined by $Grid_R$. If there exists no grid approximation in $Grid_R$, then the original power grid can be solved exactly on the regular grid: $C = A^{-1}$. In this ideal case, $\rho(B) = \rho(I - CA) = 0$, implying that HMD converges in one iteration. In practice, the regular grid problem needs not to be exactly identical to the original grid problem to have HMD converge fast, as long as it is sufficiently close. Here, the *closeness* is measured by $\rho(B)$ (the smaller the better).

In our implementation, we have found that applying a few, say m, additional simple point Gauss-Jacobi relaxations to further improve the solution obtained in (11) is very beneficial. In this case, the spectral radius (16) of the HMD iterations becomes $\rho\left(B\left(I - D^{-1}A\right)^m\right)$, where D is the diagonal matrix corresponding to Gauss-Jacobi iterations. This only adds a small additional cost on the host, but makes the spectral radius even smaller, improving the overall convergence rate.

C. Accuracy and overhead

The HMD iteration scheme favorably enhances the robustness of the proposed algorithm and relaxes the need for a very accurate 3D-irregular-to-2D-regular grid mapping (Section III). For a set of power grid benchmarks, as the regular grid size is varied from 80% to 120% of the original grid size, running two HMD iterations can always reach a very satisfactory accuracy level of less than $1mV$ average node voltage error. Increasing to three iterations will cut the average error down to less than $0.5mV$. In addition to the solution of a small sparse matrix problem (the coarsest grid), required by the GMD method, the host (CPU) also conducts simple smoothing, correction and residue computation steps. The CPU runtime is typically only $1/3$ to $1/10$ of the total GpuHMD runtime.

TABLE I

DC ANALYSIS RESULTS OF THE GMD SOLVER. $GridSize$ IS THE NUMBER OF NODES OF THE ORIGINAL POWER GRID, N_l IS THE NUMBER OF MULTIGRID LEVELS, $N_{bot.}$ IS THE BOTTOM LEVEL REGULAR GRID SIZE, N_{Vc} IS THE NUMBER OF V-CYCLES, $\Delta V(mv)$ IS THE SOLUTION RANGE ($V_{max} - V_{min}$), E_{avg} IS THE AVERAGE ERROR, AND E_{max} IS THE MAXIMUM ERROR (THE DATA FOR THE VDD AND GND GRIDS ARE SHOWN IN THE FORM OF VDD/GND). T_C/M_C IS THE RUNTIME/MEMORY USING CHOLMOD SOLVER, T_{CPU} (T_{GPU}) IS THE RUNTIME USING GMD ON CPU (GPU) AND $Spd.$ IS SPEEDUP T_{CPU}/T_{GPU} (THE ABOVE T_C, T_{CPU} AND T_{GPU} ARE THE TOTAL RUNTIME FOR SOLVING BOTH THE VDD AND GND GRIDS.).

CKT	$GridSize$	N_l	$N_{bot.}$	N_{Vc}	$\Delta V(mv)$	$E_{avg}(mv)$	$E_{max}(mv)$	$T_C(s)/M_C$	$T_{CPU}(s)$	$T_{GPU}(s)$	$Spd.$
$ibmpg2$	127, 238	4	535	4/4	347/275	3.7/2.5	21/8.3	30/155M	2.4	0.24	10X
$ibmpg3$	851, 514	5	533	4/4	181/153	4.2/2.9	32/20	362/799M	16.1	0.80	20X
$ibmpg4$	953, 583	5	882	8/8	5.3/2.6	0.1/0.1	0.6/0.3	194/1.01G	28.5	1.48	19X
$ibmpg5$	1, 079, 310	5	629	10/8	48/28	1.5/1.0	4.4/4.6	N/A	25.8	1.53	17X
$ibmpg6$	1, 670, 494	5	1160	10/10	154/86	3.6/1.4	20.1/10.3	N/A	48.5	2.78	18X

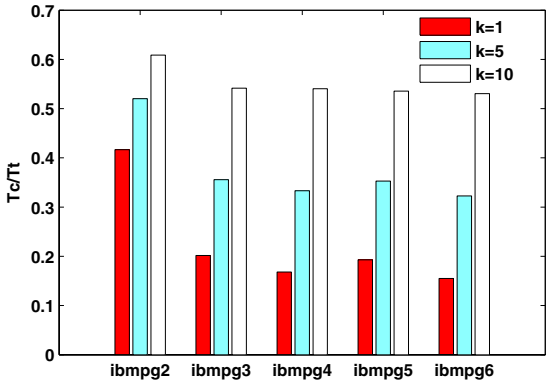

Fig. 8. Ratio of GPU computation time (T_c/T_t).

VII. EXPERIMENTAL RESULTS

Extensive experiments are conducted to demonstrate the promising performance of the proposed GpuHMD engine. A set of published industrial power grids [15], [16] and synthetic benchmarks are used to compare five solvers: proposed GPU accelerated GMD solver (GpuGMD), the CPU implementation of the same algorithm (CpuGMD), proposed GPU accelerated hybrid solver (GpuHMD), the GPU implementation of the same algorithm (CpuHMD), and the state-of-the-art CPU-based direct sparse matrix solver CHOLMOD [11]. All the algorithms are implemented using C++ and the GPU programming interface CUDA [9]. The hardware platform is a Linux PC running at 3.0 GHz clock frequency with an NIVDIA GeForce 8800 Ultra GPU.

The GMD solvers are terminated when the residue reaches 0.1% (0.5% for the HMD iteration) of the initial residue. The HMD solvers are stopped when the (estimated) average node voltage error is less then $0.5mV$. The comprehensive results of GpuGMD and GpuHMD for all the industrial benchmarks are shown in Table I and Table II. The results for VDD nets and GND nets are displayed as VDD/GND. When the grid size reaches over one millon nodes, the direct solver CHOLMOD failed due to memory limitation. For the benchmarks to which CHOLMOD can be applied, GpuGMD and GpuHMD are about 100X to 350X faster than CHOLMOD. GpuGMD is up to 20X faster than CpuGMD while GpuHMD is up to 16X faster than CpuHMD. Additionally, in Table II, 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 $0.5mV$ average node voltage error and a less than $5mV$ maximum node voltage error.

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 4×4, for each choice of the local Jacobi (LBJ) iteration number k, the number of global iterations is empirically determined by $100/k$. The runtimes and ratios of the pure GPU computing time T_c over the total GPU runtime T_t (computing time+memory read/write time) for all industrial benchmark circuits are shown in Table III and Fig. 8. From Fig. 8, we observe that the pure computation time T_c can only be a fraction (15% to 60%) of the total runtime T_t, while 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 GMD solve. Therefore, the number of local iterations (k) should be selected to tradeoff between the relaxation runtime and global convergence rate. We suggest $k = 10$ for the block size of 4×4 and $k = 20$ for the block size of 8×8 in practice.

TABLE III

RUNTIME (MS) COMPOSITION OF 100 RELAXATIONS ON GPU. THE PURE COMPUTATION TIME T_c AND TOTAL RUNTIME T_t ARE LISTED AS T_c/T_t. K IS THE NUMBER OF LOCAL BLOCK-WISE JACOBI (LBJ) ITERATIONS.

CKT	$ibmpg2$	$ibmpg3$	$ibmpg4$	$ibmpg5$	$ibmpg6$
$k = 1$	25/60	45/223	60/357	49/254	73/471
$k = 5$	6.4/12.3	16/45	23/69	18/51	30/93
$k = 10$	4.2/6.9	13/24	20/37	15/28	26/49

The following insightful experiments are also conducted. 500 smoothing steps are run on both the CPU and GPU. As shown in Fig. 9 (a), the GpuGMD engine achieves $18X$ to $32X$ speedups over its CPU counterpart. The runtimes of the multi-V-cycle GMD solve are also compared on the GPU and CPU. As in Fig. 9 (b), our GPU implementation achieves roughly $10X$ speedup for small grid and $20X$ speedup for large grids.

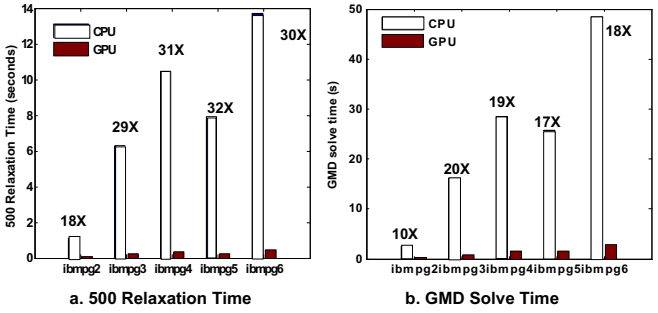

Fig. 9. CpuGMD vs. GpuGMD: runtimes of 500 relaxations and complete GMD solve.

The average solution error as a function of the number of HMD iterations is shown for the largest four industrial benchmarks in Fig.

TABLE II

DC ANALYSIS RESULTS OF THE HMD SOLVERS. N_{Iter} IS THE NUMBER OF HMD ITERATIONS, E_{avg} IS THE AVERAGE ERRORS OF THE HMD SOLVERS, E_{max} IS THE MAXIMUM ERRORS OF THE HMD SOLVERS, AND E_{wst} IS THE WORST VOLTAGE DROP/BOUNCE ERROR. T_{HMD} IS THE RUNTIME OF HMD SOLVE AND $Spd.$ IS SPEEDUP OF THE GPU ACCELERATED HMD (GPUHMD) SOLVER OVER THE SERIAL CPU IMPLEMENTATION (CPUHMD). THE DATA FOR THE VDD AND GND GRIDS ARE SHOWN IN THE FORM OF VDD/GND.

CKT	N_{Iter}	$E_{avg}(mv)$	$E_{max}(mv)$	$E_{wst}(mv)$	$T_H(s)$	N_{Iter}	$E_{avg}(mv)$	$E_{max}(mv)$	$E_{wst}(mv)$	$T_H(s)$	$Spd.$
$ibmpg2$	2/2	0.3/0.2	2.7/3.5	0.4/0.1	0.25	3/3	0.0/0.0	2.3/1.2	0.0/0.0	0.35	10X
$ibmpg3$	2/2	2.1/1.0	12.0/8.2	3.0/1.2	0.82	3/3	1.0/0.6	10.0/5.4	1.1/0.9	0.96	16X
$ibmpg4$	1/1	0.0/0.0	0.3/0.3	0.0/0.0	0.80	2/2	0.0/0.0	0.2/0.1	0.0/0.0	1.48	16X
$ibmpg5$	2/2	0.6/0.2	4.4/2.8	2.4/2.9	1.20	3/3	0.4/0.2	3.0/2.9	1.5/1.5	1.50	15X
$ibmpg6$	3/3	0.6/0.2	5.5/1.8	1.4/0.2	2.10	4/4	0.5/0.2	4.3/1.5	0.0/0.2	2.80	15X

10 (left). The average errors of all four benchmarks can be damped very quickly after two or three HMD iterations.

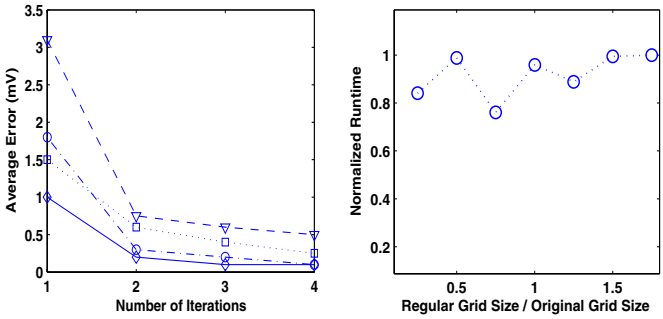

Fig. 10. Average error vs. HMD iteration count (left); regular grid size vs. runtime (right).

In Fig. 10 (right), the dependency of the total GpuHMD runtime on the regular grid size is shown for IBM benchmark $ibmpg6$. As the regular grid size is varied from 20% to 150% of the original grid size, the GpuHMD HMD runtime does not vary significantly under the same accuracy tolerance. This indicates that high accuracy in the 2D regular grid approximation is not needed. A reasonable regular grid approximation is sufficient for fast HMD convergence.

Fig. 11 shows the runtimes and memory usages of GpuGMD for several large synthetic 2D (topologically) regular power grids. These power grids are generated by using the typical wire/pad conductance values and current loadings observed in realistic industrial benchmarks. The runtime and memory consumption of GpuGMD increase rather linearly as the grid size increases. GpuGMD, the key component of GpuHMD, scales favorably with the circuit complexity, at a constant rate about one second (runtime) and $35Mb$ (memory) per million nodes.

VIII. CONCLUSIONS

In this work, we address the challenge of large-scale power grid analysis by developing a graphics processing unit (GPU) acceleration engine. To gain good efficiency on GPUs, we propose to transform an irregular grid to a regular structure so as to eliminate most of random memory access patterns and simplify control flows. To properly exploit the massively parallel single instruction multiple thread (SIMT) GPU architecture, a parallel geometrical multigrid algorithm is specially designed. New coarse grid construction and block smoothing strategies are adopted to suit the SIMT GPU platform. The robustness of the algorithm is well enhanced by an efficient CPU-GPU hybrid multigrid iteration scheme. Careful performance fine tuning is conducted to gain good analysis efficiency on the GPU. Extensive experiments have shown that our GPU engine can achieve more than $100X$ runtime speedup over a state-of-art direct solver and be $15X$ faster than the CPU based multigrid solver.

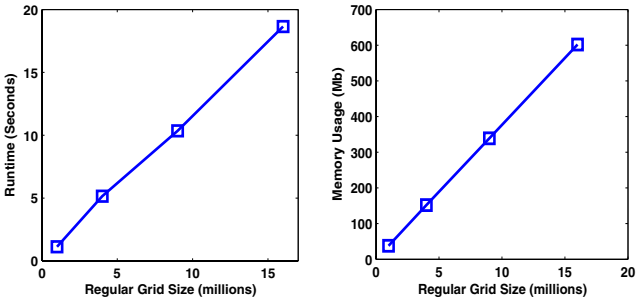

Fig. 11. Runtime and memory scalability of GpuGMD.

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] J. N. Kozhaya, S. R. Nassif, and F. N. Najm. A multigrid-like technique for power grid analysis. *IEEE Trans. on Computer-Aided Design*, 21(10):1148–1160, 2002.

[3] M. Zhao, R. Panda, S. S. Sapatnekar, and D. T. Blaauw. Hierarchical analysis of power distribution networks. *IEEE Trans. on Computer-Aided Design*, 21(2):159–168, 2002.

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

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

[6] H. Qian, S. R. Nassif, and S. S. Sapatnekar. Power grid analysis using random walks. *IEEE Trans. on Computer-Aided Design*, 24(8):1204–1224, 2005.

[7] C. Zhuo, J. Hu, M. Zhao, and K. Chen. Power grid analysis and optimization using algebraic multigrid. *IEEE Trans. on Computer-Aided Design*, 27(4):738–751, 2008.

[8] 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*, pages 54–59, 2007.

[9] NVIDIA CUDA programming guide. http://www.nvidia.com/object/cuda.html.

[10] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. *Proc. ACM SC*, 22(3):917–924, 2005.

[11] CHOLMOD. http://www.cise.ufl.edu/research/sparse/cholmod/.

[12] 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*, pages 73–82, 2008.

[13] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. on Graphics*, 22(3):917–924, 2003.

[14] L. Buatois, G. Caumon, and Bruno Levy. Concurrent number cruncher: An efficient sparse linear solver on the GPU. *HPCC, LNCS*, pages 358–371, 2008.

[15] S. R. Nassif. Power grid analysis benchmarks. In *Proc. IEEE/ACM ASPDAC*, pages 376–381, 2008.

[16] IBM power grid benchmarks. http://dropzone.tamu.edu/ pli/pgbench/.

[17] W. Briggs. *A multigrid tutorial*. SIAM Press, 1987.