Title	A GPU-accelerated parallel shooting algorithm for analysis of radio frequency and microwave integrated circuits
Author(s)	Liu, Xue-Xin; Yu, Hao; Tan, Sheldon X.-D.
Citation	Liu, X.-X., Yu, H., & Tan, S. X.-D. (2014). A GPU-accelerated parallel shooting algorithm for analysis of radio frequency and microwave integrated circuits. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, (99),1-1.
Date	2013
URL	http://hdl.handle.net/10220/19253
Rights	© 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The published version is available at: [http://dx.doi.org/10.1109/TVLSI.2014.2309606].

A GPU-Accelerated Parallel Shooting Algorithm for Analysis of Radio Frequency and Microwave Integrated Circuits

Xue-Xin Liu, Hao Yu *Member*, Sheldon X.-D. Tan *Senior Member*

***Abstract*—This paper presents a new parallel shooting-Newton method based on a GPU-accelerated periodic Arnoldi shooting solver, called *GAPAS*, for fast periodic steady state analysis of radio-frequency/millimeter-wave (RF/MM) integrated circuits. The new algorithm first explores a periodic structure of the state matrix by using a periodic Arnoldi algorithm for computing the resulting structured Krylov subspace in the generalized minimal residual (GMRES) solver. The resulting periodic Arnoldi shooting method is very amenable for massive parallel computing such as GPUs. Secondly, the periodic Arnoldi based GMRES solver in the shooting-Newton method is parallelized on the recent NVIDIA Tesla GPU platforms. We further explore CUDA GPU's features, such as coalesced memory access and overlapping transfers with computation to boost the efficiency of the resulting parallel GAPAS method. Experimental results from several industrial examples show that when compared to the state-of-the-art implicit GMRES method under the same accuracy, the new parallel shooting-Newton method can lead to up to 8 times speedup.**

***Index Terms*—periodic steady state analysis, shooting-Newton method, GMRES, Arnoldi iteration, structured Krylov-subspace, GPU parallelization.**

I. Introduction

The recent advance in radio-frequency and millimeter-wave integrated circuits (RF/MM IC) operating at 60 GHz [1]–[4] can provide much higher data rate than today's mobile devices for future smart mobile applications. However, the design for RF/MM-IC front-end circuits at this scale is very challenging [5]–[11]. At the scale of 60 GHz, all active and passive devices are closely coupled and the resulting post-layout circuit model has drastically increased complexity. Moreover, in order to follow the high-frequency carrier, traditional transient analysis requires using small time steps and hence results in a long simulation time.

The analysis of RF/MM-IC systems is notoriously difficult for speedup as accuracy can not be compromised due to high precision requirement. What is worse, its design complexity increases drastically since all active and passive devices are no longer separated but coupled. There are three approaches to numerically find the periodic steady state (PSS) solutions [5]–[10]: the finite difference time domain (FDTD) method, harmonic balance (HB) method, and shooting-Newton method. The shooting method has a better convergence for strongly nonlinear circuits because the underlying transient analysis has an adaptive time step control. However, the resulting Jacobian (sensitivity matrix) during the shooting-Newton method is usually a large-scale dense matrix. The iterative generalized minimal residual (GMRES) solver with the use of a standard Krylov subspace [12], [13] and an implicit matrix formulation [8] partially alleviates high computational costs of the shooting-Newton method.

To further improve the efficiency of shooting-Newton algorithm, one needs to explore the massive parallelism in today's muti-core and many core computing platforms. Modern computer architecture has shifted towards designs that employ multiple processor cores on a chip, so called multi-core processors or chip-multiprocessors (CMP) [14], [15]. Among these new chips, the graphic processing unit (GPU) is one of the most powerful many-core computing systems in massive use today. For instance, NVIDIA Telsa C2070 GPU has a peak performance of over 1 TFLOPS versus about 80–100 GFLOPS of Intel i5 series quad-core CPUs [16]. Recently a single Tesla K20 GPU card can deliver 4 TFLOPS with 2688 cores in a single chip, which rivals the performance of a small super-computer. Meanwhile, the introduction of new parallel programming interfaces for general purpose computations, such as Compute Unified Device Architecture (CUDA) [17], Stream SDK, and emerging OpenCL [17]–[19], has made GPUs powerful and attractive choice for solving large engineering problems. The general purpose GPU computing has been around for decades and we have many well-developed tools to improve the performance scalability of developing new scientific computing programs.

GPU parallel computing has been explored recently in many design automation algorithms [20] such as Bool satisfiability checking, statistical timing analysis, fault simulation, and so on. Within the past several years, a number of GPU-based parallel circuit simulation techniques have been proposed. They include on-chip power grid analysis methods [21]–[23], logic simulation [24] and general SPICE simulation [20], [25], where only the device model evaluation has been parallelized on GPUs. However, the shooting-based RF/MW simulation methods have not been explored on GPU platforms.

In this paper, we presents a new parallel shooting-Newton method based on a GPU-accelerated periodic Arnoldi shooting solver, called *GAPAS*, for fast periodic steady state analysis of radio-frequency/millimeter-wave (RF/MM) integrated circuits. We have the following new contributions:

This work was funded in part by NSF grants NSF OISE-1130402, CCF-1017090. The work of Hao Yu was funded in part by NRF2010NRF-POC001-001 (Singapore), and MOE ACRF Tier-1 (Singapore).

Xue-Xin Liu and Sheldon X.-D. Tan are with Department of Electrical Engineering, University of California, Riverside, CA 92521 USA. Please send the comments to Sheldon Tan (email:stan@ee.ucr.edu).

Hao Yu is with School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore. (email: haoyu@ntu.edu.sg).

- Firstly, instead of simply parallelizing the traditional shooting-Newton method, we first explore the periodic structure of the state matrix and the corresponding structured Krylov-subspace for better parallelization implementation. Since most RF/MM ICs are with periodic inputs and can be characterized as a periodic steady state (PSS) problem, the state matrix generally shows a *periodic-block-matrix structure*, or periodic structure, which will be shown to be more amenable for parallel computing solutions such as GPUs than the traditional shooting method.
- Secondly, we parallelize the periodic Arnoldi based GMRES solver in the shooting-Newton method on the recent NVIDIA Tesla GPU platform. We explore the host/device collaboration, coalesced memory access, and overlapping of memory transfer and GPU kernel computing, to further boost the efficiency of the GAPAS method.

Experimental results from several industrial examples show that when compared to the state-of-the-art implicit GMRES method under the same accuracy, GAPAS based shooting-Newton method can have up to 8 times speedup. We notice that earlier study in [26] has an initial efforts to explore the structured Krylov-subspace and its parallelization. However, it failed to identify the periodic structured Krylov-subspace during the shooting-Newton process, and nor did it explore such structure for parallelization on GPU platforms, which is the major focus of this work.

We organize the remaining parts of the paper as follows. Section II reviews the background of the periodic steady state analysis and shooting-Newton method. Then we discuss the conventional GMRES algorithm with non-structured Krylov subspace with the matrix-free method as our baseline. Next, Section III presents the GPU-accelerated periodic Arnoldi shooting (GAPAS) algorithm. GAPAS' better data independence for parallelization is discussed, and the implementation issues, such as how to explore the coalesced memory access and memory copy/kernel execution overlapping, for efficiency boost on GPU platforms are talked about in Section IV. We present the numerical experimental results in Section V, and finally conclude the paper in Section VI.

II. Background

For the completeness of the presentation of the proposed work, we would like to first review the basic mathematical concepts of periodic steady state (PSS) analysis and its solution by shooting-Newton method. We then review basic ideas of the conventional GMRES algorithm based on normal (non-structured) Krylov subspace with the matrix-free method. Those reviews will help readers to put newly proposed algorithm and new contributions into better perspectives.

A. Review of the shooting-based PSS analysis methods

1) Periodic steady state: First, terminology and definitions are given to develop the problem. In general, the time domain response of a nonlinear RF/MM integrated circuit can be obtained by solving its differential algebra equation (DAE) shown below,

$$\mathbf{f}(\mathbf{x}(t),t) = \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{q}(\mathbf{x}(t)) + \mathbf{j}(\mathbf{x}(t)) + \mathbf{u}(t) = 0, \qquad (1)$$

where $\mathbf{x}(t)\mathbb{R} \rightarrow \mathbb{R}^N$ is the state variable vector including nodal voltages and possibly several branch currents, $\mathbf{j}(\cdot)$ is a function that maps the state variable vector to a vector of N entries most of which are sums of resistive currents at a node, $\mathbf{q}(\cdot)$ is a function which maps the state variable vector to a vector of N entries that are mostly sums of capacitive charges or inductive fluxes at a node, and $\mathbf{u}(t)$ is for an external periodic source which functions as a stimulus input.

Definition 1: The circuit has a *periodic* solution of period T, if $\mathbf{x}(t_0) = \mathbf{x}(t_0 + T)$ for all $t_0 \in \mathbb{R}$.

However, it is neither practical nor necessary to verify or enforce this constraint on all $t_0 \in \mathbb{R}$. In a circuit DAE whose nonlinearities satisfy smoothness conditions and whose input is periodic with period T, finding a periodic solution is equivalent to solving a two point boundary condition

$$\mathbf{x}(T) = \mathbf{x}(0).$$

A solution $\mathbf{x}(0)$ from the above equation is referred to as the *periodic steady state* (PSS), since it has the property that if the circuit is in the state $\mathbf{x}(0)$ at $t = 0$, then the $\mathbf{x}(t)$ simulated from this initial condition will be periodic of T.

Definition 2: A *state transition* function $\boldsymbol{\phi}_T(\mathbf{x}(t_0), t_0)$ is the solution of Eq. (1) at $t_0 + T$, starting from a guessed initial state $\mathbf{x}(t_0)$ at t_0, or

$$\mathbf{x}(t_0 + T) = \boldsymbol{\phi}_T(\mathbf{x}(t_0), t_0). \qquad (2)$$

Using the state transition function, the two point boundary constraint is reformulated as

$$\boldsymbol{\phi}_T(\mathbf{x}(0), 0) = \mathbf{x}(0) \qquad (3)$$

for one period with $t_0 = 0$.

Definition 3: A *shooting sensitivity*, or called shooting Jacobian, can be further defined by

$$\mathbf{J}_{\boldsymbol{\phi}_T}(\mathbf{x}(0), 0) = \frac{\mathrm{d}\mathbf{x}(T)}{\mathrm{d}\mathbf{x}(0)} = \frac{\mathrm{d}\boldsymbol{\phi}_T(\mathbf{x}(0), 0)}{\mathrm{d}\mathbf{x}(0)}. \qquad (4)$$

As discussed next, $\mathbf{J}_{\boldsymbol{\phi}_T}$ works as the derivative in the shooting-Newton method. In each shooting cycle $[0, T]$, the Jacobian matrix records the sensitivity of the final state $\mathbf{x}(T)$ according to the perturbation of initial state $\mathbf{x}(0)$. It provides information for the solver to determine how much update is needed on the initial state $\mathbf{x}(0)$ for the next shooting cycle, in order that the difference between the initial state and the final state will be reduced in the next cycle. This shooting update operation is also described mathematically in Eq. (5). With a number of shooting iterations, the difference will converge to zero, i.e., the PSS criterion $\mathbf{x}(T) = \mathbf{x}(0)$ is attained.

For simplicity, $\mathbf{x}(0)$ will be denoted as $\mathbf{x}_0$, and $\mathbf{x}(T)$ as $\mathbf{x}_T$, in the remaining parts of this paper.

2) The shooting-Newton method: To solve for the periodic steady state $\mathbf{x}_0$ from the two point boundary condition in Eq. (3) in nonlinear form, Newton iteration is deployed. With the definitions of state transition function and shooting sensitivity, it is very straightforward to have the following Newton iteration,

$$\mathbf{x}_0^k = \mathbf{x}_0^{k-1} + \left[\mathbf{I} - \mathbf{J}_{\phi_T}(\mathbf{x}_0^{k-1}, 0)\right]^{-1} \left[\boldsymbol{\phi}_T(\mathbf{x}_0^{k-1}, 0) - \mathbf{x}_0^{k-1}\right], \tag{5}$$

where $\mathbf{I}$ is the identity matrix and k is the index of Newton iteration. Such a method is called shooting-Newton algorithm [5], [7], [8], [10]

Specifically, $\mathbf{J}_{\phi_T}$ is obtained by the following manner. One first integrates the DAE of Eq. (1) in one period T using a chosen integration scheme. Here, the backward Euler method is applied. Given $\mathbf{x}_{j-1}$ as a known state variable at current time step, backward Euler method solves the following equation to calculate the state variable $\mathbf{x}_j$ at the next time step,

$$\frac{1}{h_j}\left[\mathbf{q}(\mathbf{x}_j) - \mathbf{q}(\mathbf{x}_{j-1})\right] + \mathbf{j}(\mathbf{x}_j) + \mathbf{u}_j = 0, \tag{6}$$

where h_j is the j-th time step length, i.e., $h_j = t_j - t_{j-1}$, in the time discretization of one period $[0, T]$ into M time steps, $0 = t_0 < t_1 < t_2 < \cdots < t_M = T$.

Again, Eq. (6) is nonlinear and needs linearization to solve for $\mathbf{x}_j$. Assume that the linearization is converged at l-th transient Newton iteration, and bears the form

$$\left[\mathbf{G}(\mathbf{x}_j^{l-1}) + \frac{1}{h_j}\mathbf{C}(\mathbf{x}_j^{l-1})\right](\mathbf{x}_j^l - \mathbf{x}_j^{l-1}) = - \frac{1}{h_j}\left(\mathbf{q}(\mathbf{x}_j^{l-1}) - \mathbf{q}(\mathbf{x}_{j-1})\right) - \mathbf{j}(\mathbf{x}_j^{l-1}) - \mathbf{u}_j, \tag{7}$$

where $\mathbf{G}(\mathbf{x}_j^{l-1}) = \mathrm{d}\mathbf{j}(\mathbf{x}_j^{l-1})/\,\mathrm{d}\mathbf{x}$ is called conductance matrix, and $\mathbf{C}(\mathbf{x}_j^{l-1}) = \mathrm{d}\mathbf{q}(\mathbf{x}_j^{l-1})/\,\mathrm{d}\mathbf{x}$ is called capacitance matrix. They are obtained after the linearization of device models each time in transient Newton iteration. For simplicity, we will use $\mathbf{G}_j$ and $\mathbf{C}_j$ to represent the conductance and capacitance matrices after the linearization converged at the j-th time step.

Note that both the solving of backward Euler equations (6) and transient Newton iterations for linearization on each time step (7) are also required by standard transient analysis in SPICE at all discretized time steps.

With these conductance matrices and capacitance matrices available, it is the time to see how to calculate the sensitivity matrix, or shooting Jacobian, for shooting-Newton update equation. Recognizing that the backward Euler equation is the key to relate state variables $\mathbf{x}_{j-1}$ and $\mathbf{x}_j$ at two consecutive time steps, a differentiation of (6) with respect to the initial condition $\mathbf{x}_0$ generates the following relation

$$\left[\mathbf{G}_j + \frac{1}{h_j}\mathbf{C}_j\right]\frac{\mathrm{d}\mathbf{x}_j}{\mathrm{d}\mathbf{x}_0} = \frac{\mathbf{C}_{j-1}}{h_j}\frac{\mathrm{d}\mathbf{x}_{j-1}}{\mathrm{d}\mathbf{x}_0}. \tag{8}$$

When this is applied recursively following the chain rule for all time steps in one period, one can eventually obtain the shooting Jacobian by

$$\mathbf{J}_{\phi_T} = \prod_{j=1}^{M}\left[\mathbf{G}_j + \frac{1}{h_j}\mathbf{C}_j\right]^{-1}\frac{1}{h_j}\mathbf{C}_{j-1}. \tag{9}$$

Note that as the transient analysis is first applied, the matrices $\mathbf{G}_j + \mathbf{C}_j/h_j$, $j = 1, \ldots, M$, are already available and are stored as LU-factored sparse matrices. Therefore, if the shooting-Newton equation is to be solved with an explicitly formed sensitivity matrix using direct LU method, the computational cost is mainly from the $O(n^3 M)$ backward and forward substitutions in forming the dense $\mathbf{J}_{\phi_T}$, and the $O(n^3)$ of the direct LU factorization of (5).

3) Conventional Arnoldi method in GMRES: Since the shooting Jacobian matrix in the PSS shooting-Newton update equation is a dense matrix, the computational cost of its solving by LU factorization is expensive for large RF/MM ICs. Hence, to reduce the cost when solving the shooting-Newton update equation, the traditional non-structured Krylov subspace based GMRES method can be applied. The GMRES method is an iterative method for solving a system of large scale linear equations

$$\mathbf{A}\mathbf{x} = \mathbf{b},$$

where, in the PSS problem, the left-hand side dense matrix and the right-hand side vector are

$$\mathbf{A} = \mathbf{I} - \mathbf{J}_{\phi_T}(\mathbf{x}_0^{k-1}, 0), \text{ and } \mathbf{b} = \boldsymbol{\phi}_T(\mathbf{x}_0^{k-1}, 0) - \mathbf{x}_0^{k-1},$$

from the shooting-Newton update equation in (5).

Algorithm 1 shows one standard GMRES method [12], [13]. The approximate solution, which will minimize the residual in its 2-norm, i.e., $||\mathbf{b} - \mathbf{A}\mathbf{x}^m||_2$, is constructed from the m-th order Krylov subspace

$$\mathcal{K}^m = \mathrm{span}(\mathbf{b}, \mathbf{A}\mathbf{b}, \mathbf{A}^2\mathbf{b}, \ldots, \mathbf{A}^{m-1}\mathbf{b}).$$

To generate this Krylov subspace, the Arnoldi method [12], [13] can be used to form the orthonormal basis $\mathbf{V}^m$ that spans the Krylov subspace. Each Arnoldi iteration generates a new vector and is then appended to the previous Krylov subspace basis $\mathbf{V}^{m-1}$ to form a new orthonormal basis $\mathbf{V}^m$, such as

$$\mathbf{V}^m = [\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_m],$$

and the approximate solution calculated in GMRES is a linear combination of these basis vectors,

$$\mathbf{x}^m = \mathbf{V}^m\mathbf{y}.$$

Hence, to solve the linear equation is equivalent to finding the optimal coefficient $\mathbf{y}$ to calculate approximate solution.

Note that the orthogonalization and normalization process in Arnoldi iteration stored the related coefficients, such as inner products and norms of basis vectors, in an upper Hessenberg matrix $\mathbf{H}^m$ of size $(m+1)$-by-m. And $\mathbf{H}^m$ takes part in forming least squares problem, which is solved in the final step to calculate $\mathbf{y}$ and thus the solution $\mathbf{x}^m$.

4) Matrix-free in GMRES: To further reduce the computational cost, one needs to avoid the explicit formulation of matrix $\mathbf{A}$, since forming $\mathbf{A}$ involves $O(n^3 M)$ floating point operations and multiplying $\mathbf{A}\mathbf{v}^i$ needs another $O(n^2)$ operations for each Arnoldi iteration. The work in [8], [10] introduced a matrix-free approach to directly calculate the matrix-vector products (MVP) $\mathbf{A}\mathbf{v}^i$ without forming $\mathbf{A}$.

Algorithm 1 Non-Structured GMRES with conventional Arnoldi method

Input: matrix $\mathbf{A}$, RHS $\mathbf{b}$, and guess solution $\mathbf{x}^0$
Output: $\mathbf{x} \in \mathbb{R}^N$) such that $\mathbf{Ax} \simeq \mathbf{b}$

1: $\mathbf{r}^0 = \mathbf{b} - \mathbf{Ax}^0$
2: $h^0 = ||\mathbf{r}^0||_2$, $\mathbf{v}^0 = \mathbf{r}^0/h^0$ *// the 1st basis vector*
3: $i = 0$
4: **while** $h^i > tol$ & $i < maxIter$ **do**
5: $\mathbf{w} = \mathbf{Av}^i$
6: $\mathbf{h}^{i+1} = (\mathbf{V}^i)^{\mathrm{T}}\mathbf{w}$
7: $\mathbf{w} = \mathbf{w} - \mathbf{V}^i\mathbf{h}^{i+1}$ *// orthogonalize*
8: $g = ||\mathbf{w}||_2$, $\mathbf{v}^{i+1} = \mathbf{w}/g$ *// normalize*
9: $\mathbf{H}^{i+1} = \begin{bmatrix} \mathbf{H}^i & \mathbf{h}^{i+1} \\ 0 & g \end{bmatrix}$ *// Hessenberg*
10: $\mathbf{V}^{i+1} = [\mathbf{V}^i, \mathbf{v}^{i+1}]$ *// orthonormal basis*
11: Minimize: $||h^0\mathbf{e}_1 - \mathbf{H}^{i+1}\mathbf{y}^i||_2$ to find $\mathbf{y}^i$
12: $\mathbf{x}^{i+1} = \mathbf{x}^0 + \mathbf{V}^{i+1}\mathbf{y}^i$
13: $i = i + 1$
14: **end while**

Algorithm 2 Matrix-free MVP to replace Line 5 in Algorithm 1

Input: Previous basis $\mathbf{v}^i$ and pre-factorized matrices of (10)
Output: Implicitly calculated $\mathbf{w}$, equal to $\mathbf{Av}^i$

1: $\mathbf{w} = \mathbf{v}^i$
2: **for** $j = 1 : M$ **do** *// M is the number of time steps in one signal cycle.*
3: $\mathbf{w} = \left(\mathbf{G}_j + \frac{1}{h_j}\mathbf{C}_j\right)^{-1} \frac{1}{h_j}\mathbf{C}_{j-1}\mathbf{w}$
4: **end for**
5: $\mathbf{w} = \mathbf{v}^i - \mathbf{w}$

This is realized in Algorithm 2. This matrix-free method calculates the product of $\mathbf{J}_{\phi_T}\mathbf{v}^i$ instead of $\mathbf{Av}^i$ by iteratively reusing the pre-factored matrices of all circuit Jacobians

$$\mathbf{G}_j + \frac{1}{h_j}\mathbf{C}_j, \quad j = 1, \ldots, M. \tag{10}$$

Here, the shift of $\mathbf{I}$ to form $\mathbf{A}$ from $\mathbf{J}_{\phi_T}$ can be easily corrected when updating $\mathbf{v}^i$, as is done in Line 5 in Algorithm 2.

However, as discussed in the later part of the paper, neither the standard GMRES procedure in Section II-A3 nor the matrix-free GMRES in Section II-A4 takes into consideration the periodic structure of the matrix for the PSS RF problem. As we show in the next section, the exploration of the periodic structure of the state matrix can reveal more data independency for more efficient parallelization as shown in [27].

B. Review of GPU Architecture and CUDA programming

In this subsection, we review the GPU architecrture and CUDA programming. CUDA, short for Compute Unified Device Architecture, is the parallel programming model for NVIDIA's general-purpose GPUs. The architecture of a typical CUDA-capable GPU is consisted of an array of highly threaded streaming multiprocessors (SM) and comes with up to a huge amount of DRAM, referred to as global memory. Take the Tesla C2070 GPU for example. It contains 14 SMs,

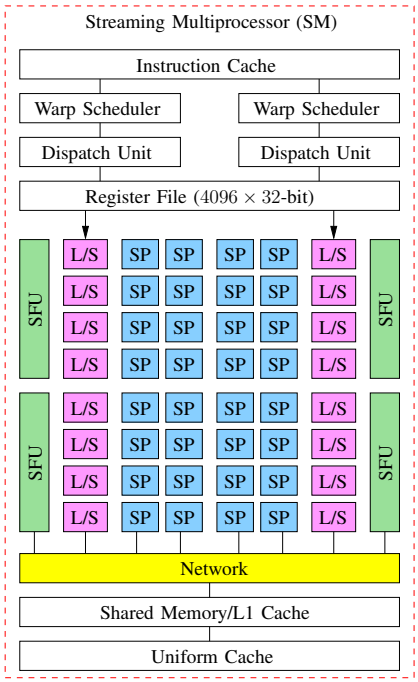


Fig. 1: Diagram of a streaming multiprocessor in NVIDIA Tesla C2070. (SP is short for streaming processor, L/S for load/store unit, and SFU for Special Function Unit.)

each of which has 32 streaming processors (SPs, or CUDA cores called by NVIDIA), 4 special function units (SFU), and its own shared memory/L1 cache. The structure of a streaming multiprocessor is shown in Fig. 1.

As the programming model of GPU, CUDA extends C into CUDA C and supports such tasks as threads calling and memory allocation, which makes programmers able to explore most of the capabilities of GPU parallelism. In CUDA programming model, illustrated in Fig. 2, threads are organized into blocks; blocks of threads are organized as grids. CUDA also assumes that both the host (CPU) and the device (GPU) maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. For every block of threads, a shared memory is accessible to all threads in that same block. And the global memory is accessible to all threads in all blocks. Developers can write programs running millions of threads with thousands of blocks in parallel. This massive parallelism forms the reason that programs with GPU acceleration can be much faster than their CPU counterparts. CUDA C provides its extended keywords and built-in variables, such as `blockIdx.{x,y,z}` and `threadIdx.{x,y.z}`, to assign unique ID to all blocks and threads in the whole grid partition. Therefore, programmers can easily map the data partition to the parallel threads, and instruct the specific thread to compute its own responsible data elements. Fig. 2 shows an example of 2-dim blocks and 2-dim threads in a grid, the block ID and thread ID are indicated by their row and column positions.

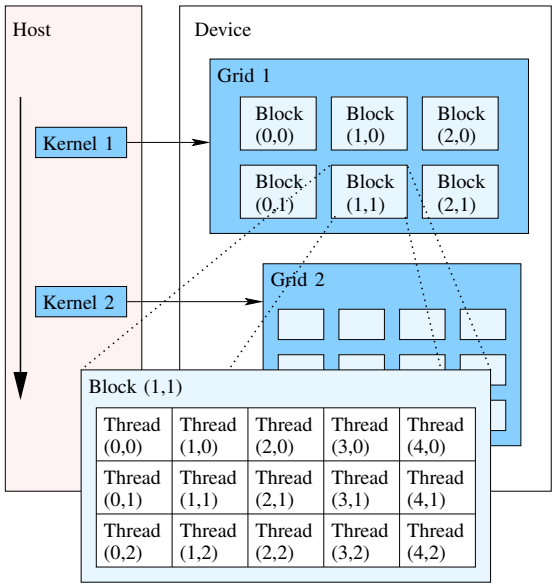


Fig. 2: The programming model of CUDA.

III. GMRES WITH PERIODIC STRUCTURED KRYLOV SUBSPACE

Based on the observation that the Krylov subspace of a periodic system usually contains a periodic-block-matrix structure (or periodic structure), we first introduce a new shooting-Newton method to utilize the periodic Arnoldi method. We show that if one can identify the periodic structure of the Krylov subspace, we can have more efficient parallelization of the resulting algorithm.

A. Periodic structured Krylov subspace

Note that the RF/MM-IC system with periodic inputs is in fact a periodic system. Hence the associated Krylov subspace would also exhibit a periodic structure. As such, it inspires us to exploit the structure-preserved Krylov subspace method during the GMRES iteration.

Let's first identify the structure of the underlying Krylov subspace. For time steps $j = 1, \ldots, M$, defining $\mathbf{A}_j = [\mathbf{G}_j + \frac{1}{h_j}\mathbf{C}_j]^{-1}\frac{1}{h_j}\mathbf{C}_{j-1}$, the shooting-Jacobian $\mathbf{J}_{\phi_T}$ in Eq. (9) becomes

$$\mathbf{J}_{\phi_T} = \mathbf{A}_M \mathbf{A}_{M-1} \cdots \mathbf{A}_1. \tag{11}$$

We emphasize again that the definition of $\mathbf{A}_j$ is only for mathematical explanation. In practice, $\mathbf{A}_j$ is never formed explicitly, because it is inefficient and unnecessary.

In the following, we show that the multiplied product $\mathbf{J}_{\phi_T}$ has an identical invariant subspace as the following periodic-block-structured matrix

$$\mathcal{J} = \begin{bmatrix} 0 & & & \mathbf{A}_1 \\ \mathbf{A}_2 & \ddots & & \\ & \ddots & & \\ & & \mathbf{A}_M & 0 \end{bmatrix}, \tag{12}$$

which has a periodic structured Krylov subspace.

Definition 4: The periodic structured m-order Krylov subspace $\mathcal{V}^m$ of $\mathcal{J}$ is defined through the following periodic Arnoldi decomposition

$$\mathcal{J}\mathcal{V}^m = \mathcal{V}^{m+1}\mathcal{H}^m, \tag{13}$$

where

$$\mathcal{V}^m = \mathbf{V}_1^m \oplus \mathbf{V}_2^m \cdots \oplus \mathbf{V}_M^m, \tag{14}$$

and

$$\mathcal{H}^m = \begin{bmatrix} 0 & & & \mathbf{H}_1^m \\ \mathbf{H}_2^m & \ddots & & \\ & \ddots & & \\ & & \mathbf{H}_M^m & 0 \end{bmatrix}. \tag{15}$$

Note that $\oplus$ denotes the direct sum operation of matrices: given $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{B} \in \mathbb{R}^{p\times q}$, $\mathbf{A}\oplus\mathbf{B}$ is a $(m+p)\times(n+q)$ matrix with $\mathbf{A}$ and $\mathbf{B}$ in the diagonal,

$$\mathbf{A} \oplus \mathbf{B} = \begin{bmatrix} \mathbf{A} & \\ & \mathbf{B} \end{bmatrix}.$$

In Eq. (13), $\mathbf{V}_j^m \in \mathbb{R}^{n\times m}$ matrices are subspace bases at the associated time steps, and $\mathbf{H}_j^m \in \mathbb{R}^{(m+1)\times m}$ matrices are the Hessenberg matrices which relate $\mathbf{A}_j$ and $\mathbf{V}_j^m$ by the Arnoldi decomposition in a cyclic fashion, i.e.,

$$\mathbf{A}_j \mathbf{V}_{j\ominus 1}^m = \mathbf{V}_j^{m+1}\mathbf{H}_j^m, \tag{16}$$

where the symbol $\ominus$ reflects the scheme of cyclic subspace construction: The operation $j \ominus 1$ returns the same integer value as $j-1$ does, if $j > 1$. If $j = 1$, $j \ominus 1$ gives the maximum integer in the range of the cycle, and in the current case for time step indices $0, 1, \ldots, M$, it gives M.

Clearly, the periodic Krylov subspace of $\mathcal{J}$ characterized by $\mathcal{V}^m$ and $\mathcal{H}^m$ is composed of block matrices $\mathbf{V}_j^m$ and $\mathbf{H}_j^m$, $j = 1, \ldots, M$. It was proved in [28] that the periodic system with a periodic block structured matrix $\mathcal{J}$, characterized by $\mathcal{V}^m$ and $\mathcal{H}^m$, has an identical Krylov subspace with the periodic system with a multiplied-product matrix $\mathbf{J}_{\phi_T}$, characterized by $\mathbf{V}_j^m$ and $\mathbf{H}_j^m$, $j = 1, \ldots, M$.

In the following subsection, we present a periodic Arnoldi Algorithm, which can provide all $\mathbf{V}_j^m$ and $\mathbf{H}_j^m$, and then, using the Arnoldi decomposition, a least squares problem is formulated and solved to construct the approximate solution in the Krylov subspaces $\mathbf{V}_j^m$. This approximate solution will play as an initial state for the next shooting cycle.

B. Periodic Arnoldi shooting algorithm

In practice, the simulation takes a lot of time steps to obtain accurate result in one signal cycle. This will generate a total number of M Krylov subspaces, which cost a high memory overhead. To reduce this M-cyclic system, [26] proposed block Gaussian elimination to derive a p-cyclic system, where $p \ll M$. The time point indices $(1, \cdots, M)$ in one cycle are partitioned as

$$(\underbrace{1, 2, \cdots, q_1}_{\text{Part 1}}, \underbrace{q_1+1, \cdots, q_2}_{\text{Part 2}}, \cdots, \underbrace{q_{p-1}+1, \cdots, q_p}_{\text{Part } p}),$$

Algorithm 3 GPU accelerated periodic Arnoldi method.

Input: Sparse LU factors of MNA matrices $\mathbf{G}_j + \frac{1}{h_j}\mathbf{C}_j$, capacitance matrices $\mathbf{C}_j$,
RHS vectors $\mathbf{b}_j$ extracted when linearization is converged, and step length h_j, $j = 1, \ldots, p$.
Output: Updated shooting result, $\mathbf{x} = [\mathbf{x}_1, \cdots, \mathbf{x}_p]$, at all time steps

1: **for** $j = 1$ to p **do** *// parallel in GPU*
2: $\hat{\mathbf{v}}_j = [\mathbf{G}_{q_{j-1}+1} + \frac{1}{h_{q_{j-1}+1}}\mathbf{C}_{q_{j-1}+1}]^{-1}\mathbf{b}_{q_{j-1}+1}$
3: **for** $k = q_{j-1} + 2$ to q_j **do**
4: $\hat{\mathbf{v}}_j = [\mathbf{G}_k + \frac{1}{h_k}\mathbf{C}_k]^{-1}\frac{1}{h_k}\mathbf{C}_{k-1}\hat{\mathbf{v}}_j + \mathbf{b}_k$
5: **end for**
6: $[\mathbf{v}_j^1, \text{dummy}] = \text{orth}(\hat{\mathbf{v}}_j, [\,])$
7: $\mathbf{V}_j^1 = [\mathbf{v}_j^1]$ *// First basis vector.*
8: $\delta_j = \langle \mathbf{v}_j^1, \mathbf{b}_j \rangle$
9: **end for**
10: **for** $i = 1$ to m **do** *// Arnoldi iteration: serial loop*
11: **for** $j = 1$ to p **do** *// parallel in GPU*
12: $\hat{\mathbf{v}}_j = \mathbf{v}_{j\ominus 1}^i$ *// $\ominus$ is defined in Eq. (17).*
13: **for** $k = q_{j-1} + 2$ to q_j **do**
14: $\hat{\mathbf{v}}_j = [\mathbf{G}_k + \frac{1}{h_k}\mathbf{C}_k]^{-1}\frac{1}{h_k}\mathbf{C}_{k-1}\hat{\mathbf{v}}_j$
15: **end for**
16: $[\mathbf{v}_j^{i+1}, \mathbf{H}_j(1{:}i+1, i)] = \text{orth}(\hat{\mathbf{v}}_j, \mathbf{V}_j^i)$
17: $\mathbf{V}_j^{i+1} = \begin{bmatrix}\mathbf{V}_j^i & \mathbf{v}_j^{i+1}\end{bmatrix}$ *// New basis vector.*
18: **end for**
19: **end for**
20: Solve least squares problem in Eq. (18)
21: Calculate the shooting result $\mathbf{x}$

where $q_p = M$, and the selection of q_j, $j = 1, \ldots, p$, usually forms a uniform partition of the whole cycle, which makes the number of time steps in each segment the same, i.e., close to M/p. This partition will separate the total M time steps into p segments, where p Krylov subspaces are to be computed by Arnoldi iteration. The parameter p is in fact number of independent Krylov subspaces and number of blocks in the block Gaussian elimination defined in [26]. We follow the reduction scheme as described in [26], with only changes to employ matrix-free basis vector computation inspired by [8]. Our algorithm is summarized in Algorithm 3. The Krylov subspaces constructed inside this algorithm still satisfy the relationship stated in Eq. (16), except that the number of blocks is reduced from M to p, and hence the definition of $j \ominus 1$ is changed to

$$j \ominus 1 = \begin{cases} j-1, & \text{if } j = 2, \ldots, p, \\ p, & \text{if } j = 1. \end{cases} \tag{17}$$

We call the procedure in Algorithm 3 as periodic Arnoldi shooting (PAS) based GMRES. In this algorithm, the subscript j denotes the index of the periodic blocks or time steps, $j = 1, \ldots, p$, and the superscript i denotes the index of the order of the Krylov subspace, $i = 1, \ldots, m$. The key procedure of PAS GMRES is the periodic Arnoldi iteration, between Line 10 and Line 19, which generates the periodic Krylov subspace, i.e., the block matrices $\mathbf{V}_j^m$ and $\mathbf{H}_j^m$, $j = 1, \ldots, p$, as shown in Eq. (14) and Eq. (15).

By employing the Arnoldi decomposition in Eq. (13), the generated subspace bases $\mathbf{V}_j^m$ and Hessenberg matrices $\mathbf{H}_j^m$ from Algorithm 3 are used to determine the approximate shooting solution $\mathbf{x}$. Similar to the traditional GMRES, the coefficient $\mathbf{y}$ of the linear combination of subspace basis vectors is solved in a least squares problem, where the left-hand side matrix is constructed from Hessenberg matrices $\mathbf{H}_j^m$, i.e.,

$$\begin{bmatrix} \tilde{\mathbf{I}} & & & -\mathbf{H}_1^m \\ -\mathbf{H}_2^m & \tilde{\mathbf{I}} & & \\ & \ddots & \ddots & \\ & & -\mathbf{H}_p^m & \tilde{\mathbf{I}} \end{bmatrix} \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_p \end{bmatrix} \simeq \begin{bmatrix} \delta_1 \mathbf{e}_1 \\ \delta_2 \mathbf{e}_1 \\ \vdots \\ \delta_p \mathbf{e}_1 \end{bmatrix}, \tag{18}$$

with $\tilde{\mathbf{I}} = \begin{bmatrix} \mathbf{I}_{m\times m} \\ \mathbf{0} \end{bmatrix}$. Then, after the solving completed, each $\mathbf{y}_j \in \mathbb{R}^m$ is used to calculate the shooting result at the j-th time step, $\mathbf{x}_j = \mathbf{V}_j^m \mathbf{y}_j$.

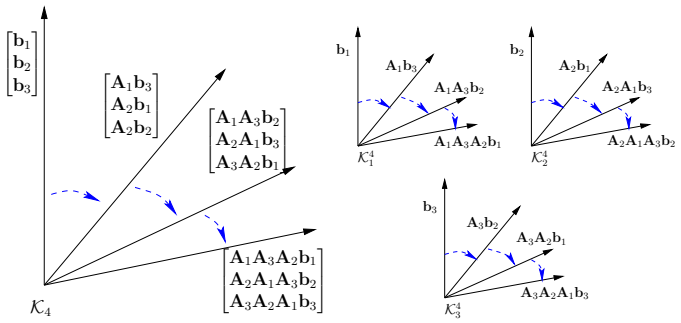
Similar to the concept of matrix-free in Algorithm 2, the product of $\mathbf{A}_j\hat{\mathbf{v}}_j$ is directly calculated by using sparse matrix $\mathbf{C}_j$ in the MVP and using pre-factored sparse LUs of $\mathbf{G}_j + \mathbf{C}_j/h_j$ in triangular solves, shown in Line 3 to Line 5 and Line 13 to Line 15 in Algorithm 3. Then, these newly computed vectors go through a series of orthogonalization with respect to their predecessors in the corresponding subspaces. Here, Householder orthogonalization is applied for its better accuracy and quality of orthogonality.

However, different from standard GMRES and matrix-free GMRES, the periodic structure of the generated Krylov subspace is preserved in Algorithm 3, because each orthonormalized base $\mathbf{v}_j^i$ is constructed separately for each $\mathbf{A}_j$. In contrast, the bases of the Krylov subspace in Algorithm 1 and Algorithm 2 are generated for the overall $\mathbf{J}_{\phi_T} = \mathbf{A}_p\mathbf{A}_{p-1}\cdots\mathbf{A}_1$. As a result, the periodic structure of the underlying Krylov subspace is destroyed.

C. More independent tasks for better parallelization

In this subsection, we show that the p-cyclic GMRES algorithm is more suitable for parallelization than the traditional GMRES method in the context of the shooting method.

To map an algorithm onto a parallel one, we first identify the parallelizable parts or independent computing parts, especially the most expensive elements, in the sequential algorithm. At one Arnoldi iteration in Algorithm 3, the p Krylov subspaces are independently calculated. For each of these subspaces, to generate a new basis vector, there would be approximately M/p matrix-vector multiplications, sparse LU triangular solves, where M is the total number of time points in the signal cycle. The orthogonalization step also needs BLAS level-1 operations such as dot product of vectors, norm of vector, and SAXPY ($\mathbf{y} \leftarrow \alpha\mathbf{x}+\mathbf{y}$). Even though matrix-free formulation is applied, the new basis vector generation still requires M/p forward eliminations and backward substitutions (two triangular matrix solving) with the saved sparse LU factors, which is still the most computation intensive task. Furthermore, to complete the m-th order Krylov subspaces at all time steps, this Arnoldi iteration is repeated for m times. Therefore, the most time consuming step in the whole



(a) Krylov subspace in traditional GMRES. (b) Krylov subspace in p-cyclic GMRES.

Fig. 3: The comparison of standard Krylov subspace and p-periodic-block-structured Krylov subspace. Order $m = 4$, and $p = 3$ time steps.

algorithm is the Arnoldi iteration. Fortunately, in the new algorithm, the calculation of subspaces at different time steps become independent and the p-cyclic Arnoldi iterations can be mapped to many independent threads in GPU, which is clearly better than the traditional non-structured Arnoldi method where these subspaces have to be computed sequentially.

The complexity and convergence of a periodic Arnoldi method is similar to the standard Arnoldi method. The least square problem in Line 20 has a reduced size which is much smaller than the scale of the original circuit problem. Hence, its solving is comparatively inexpensive and a common LAPACK routine, `gels`, can accomplish the job efficiently on CPU.

D. Krylov subspaces underlying PAS method

The primary observation in Section III-A is to illustrate that the underlying Krylov subspace of shooting-Newton implicitly possesses the periodic structure as shown in (12). Consequently, using the periodic Arnoldi shooting (PAS) solver, the subspace at each time step is calculated from each $\mathbf{A}_j$ and right-hand side $\mathbf{b}_j$, $j = 1, \ldots, p$, which preserves the periodic structure in (12).

Take $m = 4$ and $p = 3$ for example [26], in the traditional GMRES, the solution is

$$\mathbf{x}^4 \in \text{span}\left(\begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{bmatrix}, \begin{bmatrix} \mathbf{A}_1\mathbf{b}_3 \\ \mathbf{A}_2\mathbf{b}_1 \\ \mathbf{A}_3\mathbf{b}_2 \end{bmatrix}, \begin{bmatrix} \mathbf{A}_1\mathbf{A}_3\mathbf{b}_2 \\ \mathbf{A}_2\mathbf{A}_1\mathbf{b}_3 \\ \mathbf{A}_3\mathbf{A}_2\mathbf{b}_1 \end{bmatrix}, \begin{bmatrix} \mathbf{A}_1\mathbf{A}_3\mathbf{A}_2\mathbf{b}_1 \\ \mathbf{A}_2\mathbf{A}_1\mathbf{A}_3\mathbf{b}_2 \\ \mathbf{A}_3\mathbf{A}_2\mathbf{A}_1\mathbf{b}_3 \end{bmatrix} \right),$$

while the p-periodic-block-structured method has the solution

$$\mathbf{x}^4 = \begin{bmatrix} \mathbf{x}_1^4 \\ \mathbf{x}_2^4 \\ \mathbf{x}_3^4 \end{bmatrix} \Bigg| \begin{array}{l} \mathbf{x}_1^4 \in \mathcal{K}_1^4 = \text{span}(\mathbf{b}_1, \mathbf{A}_1\mathbf{b}_3, \mathbf{A}_1\mathbf{A}_3\mathbf{b}_2, \mathbf{A}_1\mathbf{A}_3\mathbf{A}_2\mathbf{b}_1) \\ \mathbf{x}_2^4 \in \mathcal{K}_2^4 = \text{span}(\mathbf{b}_2, \mathbf{A}_2\mathbf{b}_1, \mathbf{A}_2\mathbf{A}_1\mathbf{b}_3, \mathbf{A}_2\mathbf{A}_1\mathbf{A}_3\mathbf{b}_2) \\ \mathbf{x}_3^4 \in \mathcal{K}_3^4 = \text{span}(\mathbf{b}_3, \mathbf{A}_3\mathbf{b}_2, \mathbf{A}_3\mathbf{A}_2\mathbf{b}_1, \mathbf{A}_3\mathbf{A}_2\mathbf{A}_1\mathbf{b}_3) \end{array}$$

Their differences can be clearly illustrated by Fig. 3. The independency of Krylov subspaces constructed on different time step segments is obviously revealed. In the following part, we further show that such a structured subspace can also lead to a number of structured operations, which can be easily mapped on the parallel GPU platform.

IV. GPU-accelerated periodic Arnoldi shooting method — GAPAS

In this section, we will present the implementation details of GPU accelerated PAS method, GAPAS, such as thread organization, memory allocation and access, and latency hiding.

A brief illustration of shooting-Newton process is drawn in Fig. 4, where the shooting iteration is on top of one period simulation of traditional transient analyis, and the shooting update calculation on GPU part is shown at the end of each period's simulation.

The good virtue of p-cyclic GMRES is its parallel subspace construction, which is run on GPU in this work. A total number of p parallel tasks can be dispatched to parallel GPU computing platforms simultaneously. In addition, the change of p does not affect GPU scheduling, since NVIDIA CUDA GPU parallel architecture allows flexible resource allocation, such as mapping an arbitrary number of GPU cores to a parallel thread block, which works on an independent part of data in the p-cyclic GMRES. The launch of parallel GPU threads and blocks can be configured during running time, and different thread blocks are scheduled independently on the GPU streaming multi-processors. This is called transparent scalability, where hardware is free to schedule thread blocks on any processor [29].

In order to fully unleash the massive parallel power of CUDA GPU architecture, several key issues have to be considered in order to make the shooting process cater the appetite of GPU: (1) block mapping of threads with the goal to balance resource per block and thus keep all streaming multiprocessors busy; (2) optimizing thread organization to enable coalesced memory access; and (3) concurrent launching of kernel execution and memory copy, in order to hide data transfer latency. We will explore those key issues in our GPU implementation of the new algorithm as shown below.

A. Parallelism strategy on GPUs for p-cyclic GMRES method

We begin our discussion from the parallelism strategy on GPU platforms for the major computing elements in the Arnoldi iteration shown in Algorithm 3. First, the sparse matrix-vector product $\mathbf{C}_j\hat{\mathbf{v}}_j$ can be parallelized both on block level and thread level, since the multiplication of the matrix and the old basis vector can be carried out independently in different thread blocks for different `blockIdx`. Furthermore, inside each block, different rows will be computed in parallel by multiple threads, which is similar to [30]. Next, the resulting product vectors go through the sparse triangular solving steps with the pre-stored LU factors. Though the forward elimination and back substitution are inherently sequential at the first glance, there is still many parallelism suitable for GPU, especially for the sparse circuit matrices, where the dependencies on previously obtained elements of solution are not as dominant as they are in dense matrix cases [31]. Lastly, the newly generated vectors must be orthogonalized with respect to their predecessors in corresponding subspaces. This orthonormalization process can be highly parallelized, because only vectors are involved here and all operations belong to the BLAS level-1 category. Due to their data-parallel

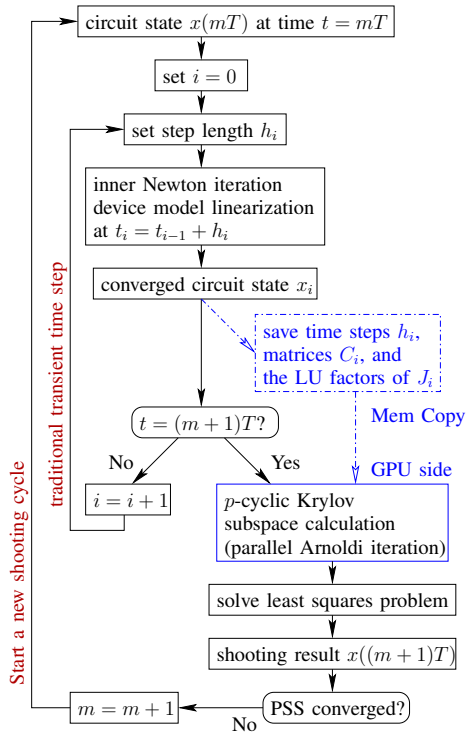


Fig. 4: The flow of shooting-Newton update with p-cyclic GMRES solver on GPU side.

nature and fairly regular memory access pattern, GPU usually can almost attain its peak performance.

B. Task and data assignments between CPU and GPU

GPU computation can accelerate both the traditional GMRES and our PAS GMRES. For traditional GMRES, one observation is that tremendous amount of matrix vector multiplications (MVMs), which is very amenable for GPU parallelism, are employed in GMRES. To implement such a common MVM operation on GPU, we notice that memory bandwidth is a critical part of performance bottleneck and must be considered when programming applications for GPU computing. Given sufficient memory, it allows us to keep our values of matrix $\mathbf{A}$ as well as basis vectors $\mathbf{v}^i$ in GPU memory without constantly swapping them in and out of memory for each iteration. One of the key aspects to making the MVM in GMRES performe well is to limit the number of memory transactions. Improving the efficiency of memory transactions within one GPU, the computing application could observe the performance difference between 10% and 50% of the theoretical computing limit of one GPU.

Specifically, once the initial transfer of the left-hand side matrix $\mathbf{A}$, right-hand side vector $\mathbf{b}$, and initial guess $\mathbf{x}_0$ are finished, the only time a memory transfer is made in the main loop of the GMRES implementation is when building the Hessenberg matrix during the orthonormalization process. During this step, the $(i+1)$-th column of $\mathbf{H}$ is transferred back to the CPU side, from which a least square minimization is performed to see if the desired tolerance of our residual has been met. Therefore, all matrix-vector, vector-vector, and scalar-vector operations are performed without any unnecessary data transfers. All of these operations are implemented with the well optimized CUDA implementation of Basic Linear Algebra Subprograms (CUBLAS).

For PAS GMRES, GPU parallel computation can also be applied to the overall Arnoldi iteration for p-cyclic Krylov-subspace construction, since the generation of new basis vectors in their corresponding Krylov-subspaces are highly parallel. This can be easily observed by inspecting the for-loop in Algorithm 3. In a serial CPU computation, it iterates through all the p time steps, in order to carry out the matrix-vector multiplication, vector orthogonalization, and save basis vectors and Hessenberg matrices. However, the computation of basis vector inside each subspace for different time steps $j = 1, \ldots, p$ are independent. Therefore the for-loop between Line 11 and Line 18 are implemented in parallel on GPU in our parallel version, as illustrated in Fig. 6. Although the for-loop at the beginning of the algorithm is a one time job and its consumes only a small fraction of the whole computation time, it is also implemented in parallel since the kernel functions in this loop, e.g., triangular solve and Householder orthogonalization, can be reused by the aforementioned parallel part in the Arnoldi iteration.

Before the GPU kernel functions (functions run in GPU multiprocessors) work on the parallel Arnoldi process, device memory allocation are required for subspace bases $\mathbf{V}_j^m$ with size of N-by-m per time step, Hessenberg matrices $\mathbf{H}_j^m$ with size of $(m+1)$-by-m per time step, and other necessary workspace for intermediate results, for all time step indices $j = 1, \ldots, p$. This is shown in Fig. 7, where memory blocks for different time steps are marked by different colors, and the assigned GPU kernels for each time step will read from and write into their corresponding memory blocks. After the overall Arnoldi process is complete, the Hessenberg matrices are used to solve a least squares problem on CPU to finally assemble the approximate solution in the span of the Krylov subspaces.

C. Coalesced memory access in subspace construction

Now let us go through the details of the GPU kernel functions for the parallel Krylov subspace construction. The first kernel of our interest is the one which carries out the matrix-vector multiplication, since p-cyclic GMRES, like all its siblings in the family of Krylov subspace methods, relies on matrix-vector multiplication to generate the new basis vector. Note that in our matrix-free technique, the matrix-vector multiplication actually comprises several sequential steps: 1) sparse matrix-vector product, i.e., $\mathbf{w} = \frac{1}{h_i}\mathbf{C}_j\mathbf{v}$, and then, 2) sparse triangular solve of $\mathbf{w} = [\mathbf{G}_j + \frac{1}{h_j}\mathbf{C}_j]^{-1}\mathbf{w}$, where the circuit Jacobian is expressed by its sparse LU factors such as $\mathbf{L}_j$, $\mathbf{U}_j$, and some pivoting information.

Since the new basis vectors generated on different time steps are calculated independently, the steps such as sparse matrix vector multiplication, triangular solve, and orthogonalization of the basis on different time steps are mapped onto separate GPU blocks in the kernel invoking.

The cyclic Krylov method is reflected in the vector copy operation, $\hat{\mathbf{v}}_j^i = \mathbf{v}_{j\ominus 1}^i$, which is a prerequisite step in the

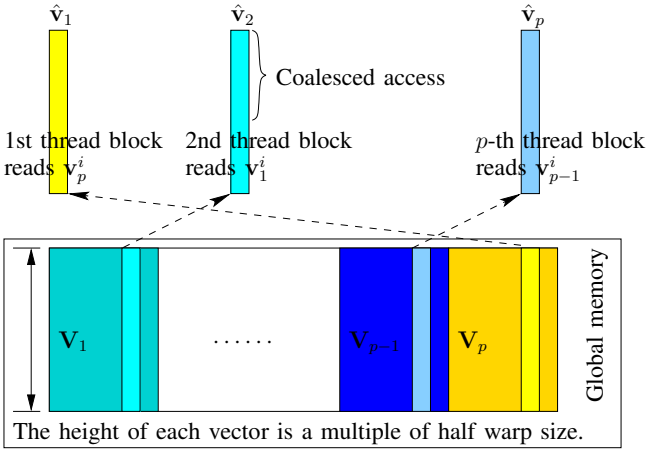


Fig. 5: Parallel p-cyclic basis vector copy using GPU thread blocks. (Line 12 in Algorithm 3)

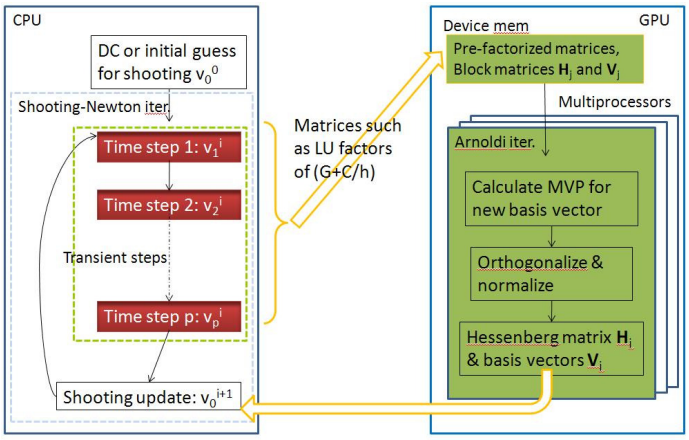


Fig. 6: CPU-GPU collaboration inside a shooting cycle.

beginning of each loop. The symbol $\ominus$ is defined in Eq. (17). In the i-th iteration of the Arnoldi process, the new basis vector $\mathbf{v}_j^{i+1}$ of the j-th block is generated using $\mathbf{v}_{j\ominus 1}^i$, i.e., the most recent basis vector of the $(j \ominus 1)$-th block. In CUDA GPU devices, when the base address of global memory access inquired by the threads of a half warp is aligned to memory segment boundary and the threads access memory in a consecutive way (neighbor thread accesses neighbor data), the number of memory transactions is minimized and thus performance is optimized. Accordingly, our allocation and storage of all the Krylov basis vectors at all time steps (blocks) have taken care of this issue, hence the vector copy can be carried out in a coalesced fashion. To be more specific, we enumerate p thread blocks in the GPU kernel, and the j-th thread block will accomplish the job of copying $\mathbf{v}_{j\ominus 1}^i$ to $\hat{\mathbf{v}}_j^i$. Inside each block, coalesced memory read and write are achieved: the k-th thread accesses the k-th element in both of the vectors.

The next step is calculation of $\mathbf{A}_j\mathbf{C}_j\hat{\mathbf{v}}_j/h_j$ using the saved sparse matrices of $\mathbf{C}_j$ and LU factors of $\mathbf{A}_j$. We further remark the application of MVM to the calculation of new basis vector. Same as the matrix-free GMRES method, we also use the pre-factorized LU matrices from the linearization process of device models, since these matrices are required by SPICE on each time step. Therefore, once they are available, we save them into compressed row major form (CSR) sparse matrix. Hence, it accelerates periodic structured GMRES when using GPU-MVM to generate the new vector. To enable the optimized global memory coalesced accesses to the CSR matrices of $\mathbf{C}_j$ and LU factors, zeros are padded to make the number of entries in each row a multiple of half warp size.

It has been shown in [26] that the Krylov subspace construction, i.e., the for-loop from $i = 1$ to m, is the most expensive part of Algorithm 3. However, it does not take much effort to find out that its loop body, i.e., the inner for-loop which calculates subspaces at all the p time steps and their corresponding blocks, can be mapped to p parallel GPU thread blocks, so that each thread block computes new basis vector and orthogonalization at one time step and different thread blocks work independently. It is noteworthy that although the operation in Line 12 requires the result from previous iteration and the j-th block needs the result of the $j\ominus 1$-th block, it can be efficiently implemented in parallel with coalesced global memory access. Synchronization are necessary to ensure all the p blocks have written the data to global memory before a memory read starts in the new iteration. This operation is illustrated in Fig. 5.

D. Overlapping memory transfer with computation

In GAPAS, solving the least squares problem in Eq. (18) is done in CPU as we have to check the results from p parallel CPU blocks to determine the convergence of the algorithm. As a result, memory transfer from device to host is necessary. A simple way to perform this memory transfer would be copying the whole m-th order Hessenberg matrices after all the Arnoldi iterations completed. However, there is a better way to do this. We propose a scheme to take the advantage of CUDA GPU's capability of copying the memory concurrently with kernel execution. This means the most recently generated columns $\mathbf{H}_j(1:i+1,i)$ are scheduled to be sent to host memory asynchronously, while the GPU kernel is calculating the new basis vector in the $(i+1)$-th Arnoldi iteration. Therefore, the memory copy time is well overlapped by the running time of kernel execution. The only prerequisite special to concurrent memory copy is the allocation of page-locked host memory. We notice that this kind of memory is a scarce resource and should not be overused. However, the memory size required to store all the Hessenberg matrices is typically not big, and the computers severs can accommodate them on page-locked memory.

Once the kernels for Krylov subspace construction are finished on all m-order subspaces and all matrices of $\mathbf{H}$ are copied to host memory, the least square problem is formulated and solved by LAPACK routine `gels` on CPU.

We remark that this work mainly focus on the parallel computing on a single GPU. As multi-GPUs at a single node and different nodes become more and more popular, it is

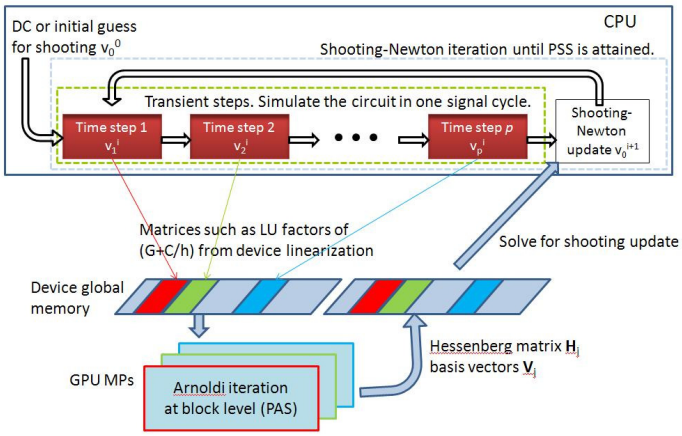


Fig. 7: Parallel computation of Krylov-subspaces on GPU.

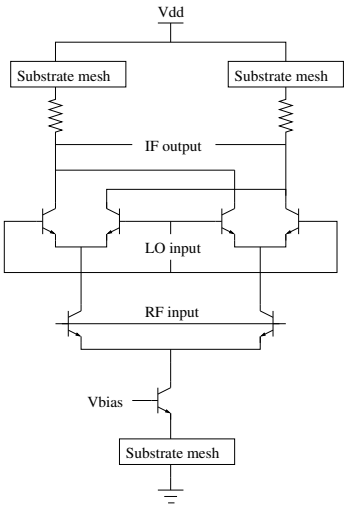


Fig. 8: A double-balanced BJT mixer.

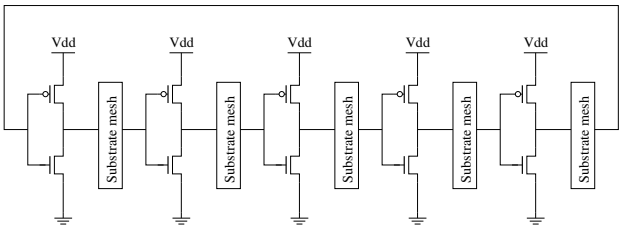


Fig. 9: A CMOS ring oscillator.

necessary to leverage multi-GPUs to further improve the simulation capability and efficiency. But multi-GPU computing is more like distributed computing versus single GPU computing on shared memory architectures and it also requires more complicated task and data partitioning at different granularity levels for both intra-GPU and inter-GPU parallelism.

V. Numerical Experiments

The proposed periodic Arnoldi shooting method, PAS-GMRES, is implemented in C++ within SPICE, and its GPU parallel version, GAPAS, is also programmed into the same simulator with CUDA C. We do admit that preconditioners are beneficial to iterative solvers such as GMRES. However, there is no preconditioner used here in any one of these iterative solvers. This is because the choice and setup of a preconditioner usually requires the knowledge of the matrix, whose explicit form is not available in the matrix-free GMRES. We list some of the parameters used in the GMRES solvers as follows. (1) The maximum number of iterations is set to 6000. In practice, any large number could be used. However, if the residual under observation does not improve very much, or the solver stalls and fails to converge, the solving process can be manually terminated. (2) The tolerance of convergence tol is set to 10^{-6}, which means the relative residual of the result, i.e., $\|\mathbf{b} - \mathbf{A}\mathbf{x}_k\|/\|\mathbf{b}\|$, is required to be less than tol. (3) The restart number is set to 32. This number is usually chosen with a small value, so that it restricts the Krylov subspace's size, and thus keeps storage cost under a limit.

The GPU card we use for all the experiments is NVIDIA's Tesla C2070 (ECC on), which is based on the Fermi architecture for true double precision support. It contains 14 multiprocessors (32 cores per multiprocessor, and totally 448 cores) and works in a 1.15 GHz clock rate with 6 GB on-chip memory and 144 GB/sec memory bandwidth. For the CPU part, the server is AMAX Tesla server, with Quad-Core Intel E5504 CPU, 24 GB memory.

The explicit GMRES with the non-structured Krylov-subspace, and the matrix-free GMRES with the non-structured Krylov-subspace, are implemented for the comparison. The non-structured Krylov-subspace is directly adapted from the template in [32], with a few modifications to comply with SPICE data structures. The matrix-free GMRES is implemented exactly following the procedure described in [8], [10], and is called *MF-GMRES* in the following. The GMRES iteration tolerance is set as 10^{-6} for voltage nodes. We compare the accuracy and running time with a scalability study using six industrial analog/RF examples, including a BJT mixer, a CMOS low noise amplifier (LNA), a CMOS frequency multiplier, a CMOS ring oscillator, a CMOS switch cap, and a CMOS DC converter. The circuit diagrams of the mixer and oscillator are given in Fig. 8 and Fig. 9. We increase the complexity by adding extracted parasitics, i.e., the substrate mesh.

Table I summarizes the simulation results for all examples. The different circuit sizes, i.e. number of equations in MNA form, are shown next to the circuit names. The table also records the running time of direct-LU method, explicit GMRES method, matrix-free GMRES method, and periodic structured Arnoldi based GMRES, in the four categories labeled 1 to 4 on the top of the table. Specifically, there are two time measurements in category 4, since the proposed PAS methods are implemented both in CPU program and GPU parallel form, to show the GPU speed up over its serial counterpart.

We first show that the GAPAS method on NVIDIA GPU platform with CUDA. We can observe from Table I that, due to the optimized implementation of MVM and the use of the structured PAS, the performance of shooting-GMRES can be

TABLE I: Comparison of shooting update time using different methods. The number of time steps in one signal cycle is $M = 400$, and the partition number p is chosen as 100.

circuit	eqn #	1	2	3		4			
		Direct-LU	Explicit-GMRES	MF-GMRES		PAS-GMRES		GAPAS (GPU)	
		time (s)	time (s)	iter #	time (s)	iter #	time (s)	time (s)	speedup
CMOS LNA	800	103	99	16	7.36	12	6.48	1.22	5.3×
CMOS freq multiplier	1024	155	148	18	10.5	15	9.06	1.53	5.9×
BJT mixer	1024	164	157	18	11.2	15	9.44	1.39	6.8×
CMOS ring osc	1152	192	185	21	15.4	16	9.15	1.10	8.3×
CMOS switch-cap	1256	199	186	20	16.7	16	13.0	2.34	5.5×
CMOS DC-converter	1617	452	424	22	32.6	16	20.3	4.21	4.8×

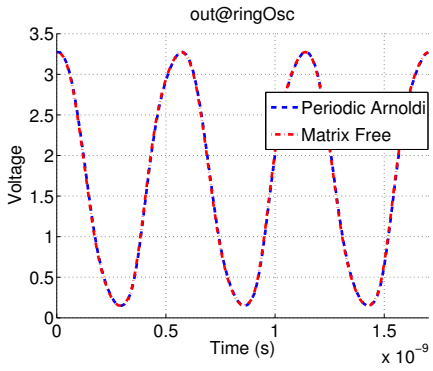


Fig. 10: The PSS waveform accuracy comparison at the output node of a CMOS ring oscillator. The red-line is the non-structured MF-GMRES, and the blue-line is the structured PAS-GMRES.

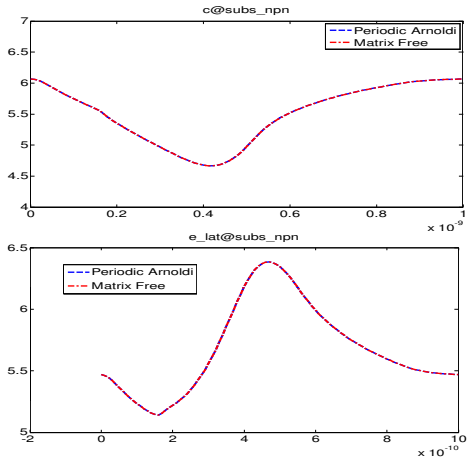


Fig. 11: The PSS waveform accuracy comparison at two nodes of a BJT mixer. The red-line is the non-structured MF-GMRES, and the blue-line is the structured PAS-GMRES.

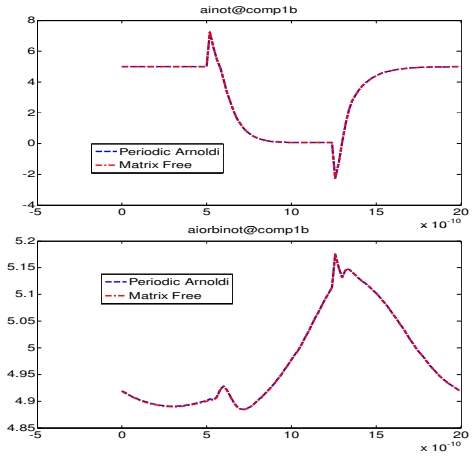


Fig. 12: The PSS waveform accuracy comparison at two nodes of a CMOS switch cap. The red-line is the non-structured MF-GMRES, and the blue-line is the structured PAS-GMRES.

TABLE II: Selection of the partition number p can affect the performance of the parallel solver. The measurement in this table is taken from the DC-converter example, with p changed to show the difference. There are totally $M = 400$ time steps in one signal cycle, which are separated into p partitions.

partition number p	GPU parallel part Arnoldi iteration time (seconds)	sequential part least squares time (seconds)	total time (seconds)
25	4.23	0.01	4.24
50	2.24	0.04	2.28
66	1.71	0.07	1.78
80	1.46	0.11	1.57
100	1.23	0.19	1.42
200	1.22	1.09	2.31

further improved in parallel implementation. Due to the limited number of testing benches, we generally observe over ten times speedup gain is obtained for GAPAS in comparison with its CPU version, the PAS-GMRES method. The performance improvement is improved for larger sized circuits. For example, 8.3× speedup is observed for a post-layout ring oscillator circuit with 1152 states. We expect that the new GAPAS solver to yield even larger speedups with much bigger sized circuits.

Second, we show that PAS-GMRES is better than the traditional MF-GMRES in terms of CPU times given the same accuracy. For PAS-GMRES, we can observe and conclude from Table I as follows. The explicit formulation of matrix in GMRES leads to the similar running time to direct-LU, since the cost of forming the matrix $\mathbf{A}$ explicitly will dominate in the whole computation in both cases. On the other hand, the direct calculation of matrix-vector product with reusing the pre-factorized sparse LUs leads to a significant speedup in MF-GMRES and PAS-GMRES.

As demonstrated in Algorithm 3 earlier, the two computationally intensive parts in this solver are the Arnoldi iteration and the least squares problem. With parallel implementation, the Arnoldi iterations on p independent Krylov subspaces are run simultaneously on GPU. However, the least squares

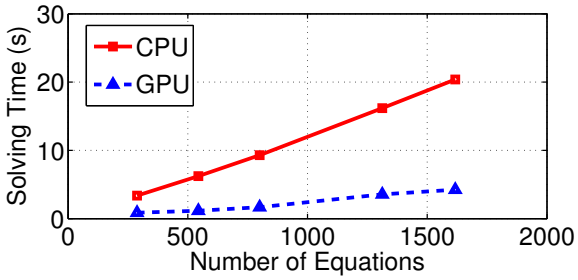


Fig. 13: The running time scalability comparison for a CMOS DC converter with increased sizes of parasitic. The red line is the scale of running time of the CPU serial version of PAS GMRES, and the blue line is the scale of running time of our GPU parallel GAPAS.

problem, whose size grows as p increases, is still a sequential one in its nature. Therefore, the partition number p must be chosen in a way that the GPU resources are utilized to their maximum capacity, and at the same time the resulting least squares problem is not too big to solve. Table II makes a comparison using different partition number p. It is observable that as more parallel Arnoldi iterations are deployed on GPU, the time spending on this part of the algorithm is reduced since the GPU cores are fully populated and maximum throughput is attained. However, simply tuning the program to accelerate this part does not result in an optimal performance. This is because the solver also needs to solve the least squares problem after the Arnoldi iteration. And the bigger the partition number p is, the bigger the size of the least squares problem, which ends up with a longer solving time. Table II suggests that $p = 100$ is the best choice which reconcile well the two contradicting procedures. Note that the time measurements here only take into account the pure CPU or GPU execution time. All the other expenses are not included, since we want to emphasize the importance of GPU task balance. The time in Table I included all time spent on the solving of a shooting update, such as CPU/GPU memory allocation and copy and other flow control overheads. The time of host/device memory copy is one concern in GPU program development, but the saved computation time from the GPU parallel kernel execution has already paid the cost of memory copy, as judged by our experiments.

We show details of three examples with a further waveform accuracy comparison. The first example is a CMOS ring oscillator, which contains 1152 states and its operation frequency is 1.6 GHz. Fig. 10 demonstrates the waveforms at the oscillator's output node, and both the MF-GMRES and the proposed PAS-GMRES attain to the same accuracy. The second example is a BJT mixer with 1024 states and a carrier input frequency of 1 GHz. Fig. 11 shows two periodic steady states at two nodes, generated by the MF-GMRES and PAS-GMRES at the same tolerance, respectively. Clearly, the two converged waveforms are identical to each other.

In Fig. 13, we further study the running time scalability by increasing the size of extracted parasitics. The example used is a CMOS DC converter with switching frequency of 1 MHz. The circuit complexity is increased from 200 to 1600 states.

TABLE III: Running time comparison between Tesla C1060 and C2070, where the same GAPAS program is run. (C2070 data is from Table I.)

Circuits	C1060 (time: s)	C2070 (time: s)
CMOS LNA	2.59	1.22
CMOS freq mult.	3.10	1.53
BJT mixer	3.07	1.39
CMOS ring osc	2.23	1.10
CMOS switch-cap	4.61	2.34
CMOS DC-convert	7.52	4.21

The running time is measured for both of serial PAS-GMRES on CPU and parallel GAPAS on GPU under the same error tolerance. For a medium sized circuit containing about one thousand states, GAPAS has a smaller running time, less than a quarter of the CPU counterpart.

Finally, we note that the performance of GAPAS on the given circuit examples is also expected to improve on newer generation GPU cards with new GPU hardware architectures and number of cores. To illustrate this, we also report the running time of GAPAS on Tesla C1060, which is the first generation general-purpose Tesla GPU from Nvidia, in Table III. From the table, we can see as expected that GAPAS runs faster on C2070 than C1060.

VI. Conclusion

In this paper, we have developed a structured shooting algorithm that can fully exploit parallelism in periodic steady state (PSS) analysis. The new algorithm, called *GAPAS*, first explores a periodic structure of the state matrix by using a periodic Arnoldi algorithm for computing the resulting structured Krylov subspace in the generalized minimal residual (GMRES) solver. We showed that the resulting periodic Arnoldi shooting method is friendly to be adapted to parallel computing like GPU. Secondly, we parallelized the periodic Arnoldi based GMRES solver in the shooting-Newton method on the recent NVIDIA Tesla GPU platform. We further explored the coalesced memory access and overlapping memory transfer with computing to boost the efficiency of the GAPAS method. Experiment results on a number of RF and millimeter wave circuits have showed that GAPAS can lead up to 8× speedup over its sequential CPU version.

References

[1] C. H. Doan *et al.*, "Design considerations for 60 GHz CMOS radios," *IEEE Commun. Mag.*, pp. 132–140, 2004.
[2] T. C. Chen, "Where CMOS is going: trendy hype vs. real technology," in *International Solid-State Circuits Conference*, pp. 22–27, Feb. 2006.
[3] A. Hajimiri, "Holistic design in mm-wave silicon ICs," *IEICE Trans. on Electronics*, pp. 817–828, 2008.
[4] B. Razavi, "Design of millimeter-wave CMOS radios: A tutorial," *IEEE Trans. Circuits Syst. I*, pp. 4–16, 2009.
[5] T. J. Aprille and T. N. Trick, "Steady-state analysis of nonlinear circuits with periodic inputs," *IEEE Proc.*, pp. 108–114, 1972.
[6] S. Skelboe, "Computation of the periodic steady-state response of nonlinear networks by extrapolation methods," *IEEE Trans. on Circuits and Systems*, pp. 161–175, 1980.
[7] K. S. Kundert, J. K. White, and A. Sangiovanni-Vincentelli, *Steady-State Methods for Simulating Analog and Microwave Circuits*. Kluwer Academic Publishers, 1990.

[8] R. Telichevesky, K. Kundert, and J. White, "Efficient steady-state analysis based on matrix-free Krylov-subspace methods," in *Proc. Design Automation Conf. (DAC)*, 1995.

[9] K. Mayaram *et al.*, "Computer-aided circuit analysis tools for RFIC simulation: algorithms, features, and limitations," *IEEE Trans. on Circuits and Systems-II*, pp. 274–286, 2000.

[10] O. Nastov, R. Telichevesky, K. Kundert, and J. White, "Fundamentals of fast simulation algorithms for RF circuits," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 600–621, 2007.

[11] H. Chang and K. Kundert, "Verification of complex analog and RF IC designs," *IEEE Proc.*, pp. 622–639, 2007.

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

[13] G. H. Golub and C. V. Loan, *Matrix Computations*. The Johns Hopkins University Press, 3rd ed., 1996.

[14] Intel Corporation, "Intel multi-core processors, making the move to quad-core and beyond (White Paper)," 2006. http://www.intel.com/multi-core.

[15] AMD Inc., "Multi-core processors—the next evolution in computing (White Paper)," 2006. http://multicore.amd.com.

[16] D. B. Kirk and W.-M. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2010.

[17] NVIDIA Corporation, "CUDA (Compute Unified Device Architecture)," 2011. http://www.nvidia.com/object/cuda_home.html.

[18] AMD Inc., "AMD Steam SDK." http://developer.amd.com/gpu/ATIStreamSDK, 2011.

[19] Khronos Group, "Open Computing Language (OpenCL)." http://www.khronos.org/opencl, 2011.

[20] K. Gulati and S. P. Khatri, *Hardware Acceleration of EDA Algorithms*. Springer, 2010.

[21] J. Shi, Y. Cai, W. Hou, L. Ma, S. X.-D. Tan, P.-H. Ho, and X. Wang, "GPU friendly fast Poisson solver for structured power grid network analysis," in *Proc. Design Automation Conf. (DAC)*, pp. 178–183, July 2009.

[22] Z. Feng and Z. Zeng, "Parallel multigrid preconditioning on graphics processing units (GPUs) for robust power grid analysis," in *Proc. Design Automation Conf. (DAC)*, pp. 661–666, 2010.

[23] Z. Feng, Z. Zeng, and P. Li, "Parallel On-Chip Power Distribution Network Analysis on Multi-Core-Multi-GPU Platforms," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 10, pp. 1823–1836, 2011.

[24] B. Wang, Y. Zhu, and Y. Deng, "Distributed time, conservative parallel logic simulation on GPUs," in *Proc. Design Automation Conf. (DAC)*, pp. 761–766, 2010.

[25] K. Gulati, J. F. Croix, S. P. Khatr, and R. Shastry, "Fast circuit simulation on graphics processing units," in *Proc. Asia South Pacific Design Automation Conf. (ASPDAC)*, pp. 403–408, 2009.

[26] W. Bomhof, *Iterative and parallel methods for linear systems, with applications in circuit simulation*. PhD thesis, Mathematical Institute, Utrecht University, 2001.

[27] X.-X. Liu, H. Yu, and S. X.-D. Tan, "A robust periodic Arnoldi shooting algorithm for efficient large-scale rf/mmic simulation," in *Proc. Design Automation Conf. (DAC)*, pp. 573–578, Jun. 2010.

[28] D. Kressner, "A periodic Krylov-Schur algorithm for large matrix products," *Numer. Math.*, vol. 103, no. 3, pp. 461–483, 2006.

[29] NVIDIA, "CUDA C programming guide." docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, October 2012.

[30] M. M. Baskaran and R. Bordawekar, "Optimizing sparse matrix-vector multiplication on GPUs," IBM Research Report RC24704, IBM Research Division, Apr. 2009.

[31] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU," NVIDIA Technical Report NVR-2011-001, NVIDIA Corp., June 2011.

[32] R. Barrett *et al.*, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA: SIAM, 1994.