UNIVERSITY OF CALIFORNIA
RIVERSIDE

Parallel and Statistical Analysis and Modeling of Nanometer VLSI Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

by

Xue-Xin Liu

March 2013

Dissertation Committee:

Professor Sheldon X.-D. Tan, Chairperson
Professor Jian-Lin Liu
Professor Qi Zhu

The Dissertation of Xue-Xin Liu is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

My thanks and appreciation go to my thesis committee members, Prof. Jian-Lin Liu and Prof. Qi Zhu for their direction, dedication, and invaluable advice. I would like to express my deepest gratitude to my adviser, Prof. Sheldon X.-D. Tan, for his help, trust and guidance. There exist wonders as well as frustrations in academic research. His kindness, insight and suggestions always lead me to the right way.

A special word of thanks for all the members in our lab should be dedicated here. I thank especially Ning, Bo-Yuan, Duo, Rui-Jing, Hai, Zao, Sahana, Rui, Santiago, Adair, Kuang-Ya, Jacob, Nema, Thom, Ryan, Eric, Jaqueline, and Omar for the collaborative research works and help, which lead to the presented works in this thesis. I appreciate the friendship of my fellow students in UCR. There are too many of them and this page does not allow me to list, but they are all in my heart.

Last but not least, I would like to thank my parents and all my friends for their constant support during the years of my study. It is the love and the light warmly shared by them that guides me to grow up.

ABSTRACT OF THE DISSERTATION

Parallel and Statistical Analysis and Modeling of Nanometer VLSI Systems

by

Xue-Xin Liu

Doctor of Philosophy, Graduate Program in Electrical Engineering
University of California, Riverside, March 2013
Professor Sheldon X.-D. Tan, Chairperson

Electronic design automation (EDA) is an important part of the integrated circuit (IC) industry, and has been evolving together with design and fabrication technologies. This evolution is reflected in both the advent of new algorithms and the rapid upgrade of software implementation. Innovative algorithms deliver accurate and reliable results in shorter computation time, and thus saves human resource and R&D cost. Smartly designed software can utilize hardware resources efficiently and improve computing performance.

However, EDA is now facing many complicated cases in current VLSI technology. As integration scales to the sub-90 nm regime, the performance of ICs is becoming less predictable. Different sources of variations are caused from manufacturing process and these variations will finally end up with parametric yield loss. To deal with yield loss, efficient algorithms are required to accurately predict the performance of a circuit at the design stage. Unfortunately, given the high complexity of VLSI systems, software tools with traditional algorithm and implementation suffer from the "curse of dimensionality" problem. For instance, Monte Carlo (MC) method, which is the most trustworthy way

to capture statistical information of a design, becomes inefficient as a large number of samplings are needed for an accurate analysis of the variations of the circuit response. Also, many verification tasks require transient simulation or frequency domain simulation of the full-chip design in order to guarantee an optimized product. A typical power grid circuit has a tremendous size of over billion nodes, and takes several days for traditional transient simulation to calculate transient response. With chips getting smaller and more complicated, the modeling and analysis will certainly become more difficult challenging.

In this thesis, we study the tough issues in statistical analysis such as Monte Carlo method and performance bound analysis, as well as modeling and simulating large scale circuit systems, such as power grid circuit and thermal circuit. Our algorithm and software solutions are proposed to reduce the computation cost without hurting the accuracy. We also take the benefits of modern multi-core and many-core computer architectures, such as multi-core CPU and general purpose GPU (GPGPU), to gain speedup in our simulation. Computational independency in MC simulation is exploited and hence we have developed a GPU parallel Monte Carlo analysis based on symbolic technique. Our parallel MC of circuit transfer functions has been verified using statistics extracted from classical MC, and the proposed method is proved to be effective. We also study optimization based method to derive the performance bounds as a non-Monte Carlo solution. The bounds from this method are accurate without over-conservativeness.

To accelerate linear algebra operations in equation solving tasks, which are common in our power grid analysis, 3D IC thermal analysis, and RF and power converter simulation, we apply GPU on fine-grained tasks in GMRES solver and attain impressive speedup

over traditional CPU method. The management of different levels of GPU resources, such as thread organization and memory assignment, are discussed so that the data intensive feature of GPU can be fully rendered and its weakness like long memory access latency be well hidden by its superb data throughput. All algorithms and implementations are demonstrated with representative numerical experiments and thorough comparisons among different methods and platforms.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Circuit simulation in nanometer regime of VLSI

Current VLSI design and fabrication technologies have marched relentlessly into nanometer scale. With the Moore's Law still dominant in this industry, although challenged more than ever, the term "deep sub-micron" used one decade ago are now fairly outdated. The 45 nm integration has already become the mainstream technology, and 22 nm transistors are not new any longer. To continue the trend of Moore's Law, more integration capability is exploited from 3-dimensional structure in the chip. In a word, the IC industry is still leading, and is expected to do so in the near future, the technology endeavor towards making cheaper, smaller, more efficient, and more powerful electronics devices for us.

This success is made possible in part by the electronic design automation (EDA) community, from both academia and industry parts. The SPICE circuit simulator is a perfect model for the software solution in this area. Originated from a course project in UC Berkeley in the early 1970s, it first evolved into a research tool in the university. Then it was

adopted and commercialized by companies and became the forerunner of the whole EDA area. Through the intensive research and development in the past several decades, SPICE and its variants are now playing crucial parts in almost every EDA software vendor's product solutions. With the facilitation of EDA tools, the IC industry gains much more efficiency and productivity than before. However, the state-of-the-art IC design also raises more tough challenges to EDA algorithms and tools. The cutting edge IC technologies do not only bring sophisticated specification and verification requirements to EDA methodologies, but also introduce more difficult issues on the algorithm and computation level. For example, the modeling and simulation of chip interconnection at the nanometer integration scale need to consider the parasitic effects and process variations, which tend to increase the computation complexity such as huge matrix size in the numerical analysis. Without novel modeling and simulation algorithms, the analysis of a full chip design could take many days to finish. In case several design modification cycles are involved, the total computation cost is too prohibitive to meet the time to market demand. Therefore, new EDA algorithm and implementation are always essential driving forces to make IC product competitive in this industry's evolution.

Among all the issues in EDA applications in nano-scale integration, variability in IC design and fabrication is a critical challenge to the continued scaling and effective utilization of CMOS technologies. While people at foundry part are doing their best to minimize transistor variation through technology optimization, the designer and EDA tools should also ensure robust product functionality and performance during their design phase.

Transistor mismatch is the primary obstacle to reach a high-yield rate for analog

designs in sub-90 nm technologies. For example, due to an inverse-square-root-law dependence with the transistor area, the mismatch of CMOS devices nearly doubles for every process generation less than 90 nm, and the 3-σ variation of drain currents reaches beyond 30% [1, 2]. Since the traditional worst-case or corner-case based analyses are so pessimistic that they sacrifice the speed, power, and area, the statistical approach [2–7] thereby becomes a trend to estimate the analog mismatch. Same as the process variation, there are two types of mismatch. One is systematical or global spatial variation, and the other is stochastic or local random variation. Analog circuit designers usually perform Monte Carlo (MC) analysis to analyze the stochastic mismatch and predict the statistical functionality of their designs. As MC analysis requires a large number of repeated circuit simulations, its computational cost is expensive. Moreover, the pseudo-random generator in MC introduces numerical noises that may lead to errors.

This thesis is motivated by the aforementioned two aspects: fast and scalable computation and accurate and reliable results. We want to develop efficient and accurate algorithm for circuit simulation and statistical analysis, while we obtain additional speedup by digging out the parallelizable part underlying the algorithm, or by rearranging the algorithm so that parallelizable part becomes outstanding. In this way, GPU platforms will perform their marvelous data parallel power in our applications.

In the next section, we will review briefly the CUDA GPU computing platform, where a majority of the researches in this thesis are done.

1.2 CUDA GPU architecture and programming

We stress at the beginning that most of the features of GPU, both advantages and limitations, comes from the fundamental design philosophy of GPU chips — GPU dedicates much more chip area to processing cores while its control logic and memory models are simpler if compared to multi-core CPU. Therefore, to get the best performance, one should use GPU on those numerically intensive tasks with simple control logic requirements. This execution style is also called single-program, multiple-data (SPMD).

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 built around a scalable array of highly threaded streaming multiprocessors (SM) and comes with the GPU card's own DRAMs, referred to as global memory. Current CUDA GPU cards have from 768 to 12,288 concurrently executing threads. Transparent scaling across this wide range of available parallelism is a key design goal of both the GPU architecture and the CUDA programming model [8]. Take the Tesla C2070 GPU for example. Fig. 1.1 shows the C2070 GPU with 14 SMs interconnected with several external DRAM partitions. Each of these 14 SMs has 32 streaming processors (SPs, or CUDA cores called by NVIDIA), 4 special function units (SFU) for transcendental functions, one multi-threaded instruction unit, and its own on-chip shared memory/L1 cache. The SM in C2070 can creates, manages, and executes up to 1,536 concurrent threads in hardware with zero scheduling overhead. The structure of a streaming multiprocessor is shown in Fig. 1.2.

As the programming model of GPU, CUDA extends C into CUDA C and supports tasks such as threads calling and memory allocation, which make programmers able

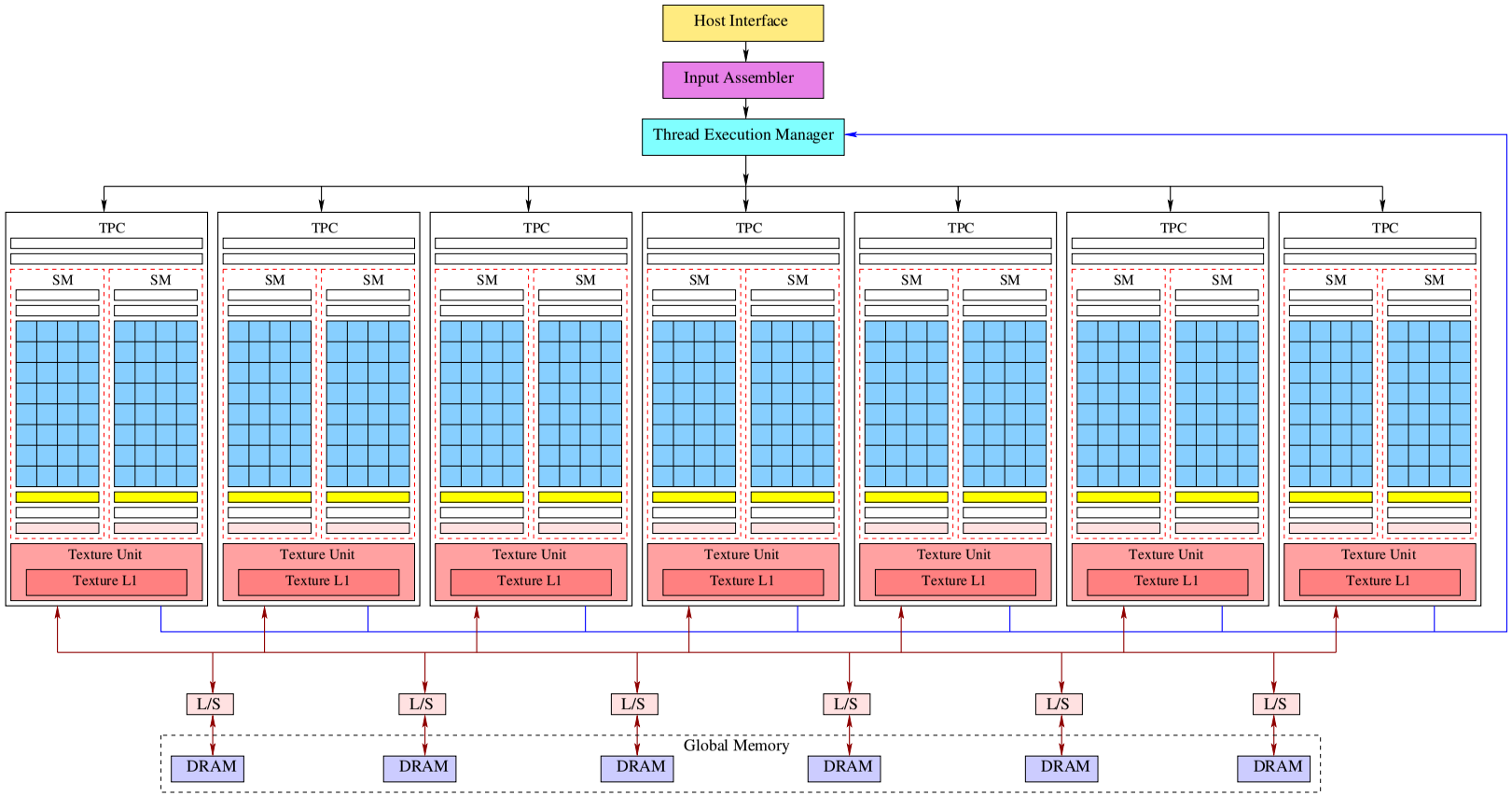


Figure 1.1: NVIDIA Tesla GPU unified graphics and computing architecture C2070. Each texture/processor cluster, or TPC, is composed of two streaming multi-processors. The off-chip DRAMs are accessible to all threads and are referred to as global memory. L/S is short for load/store.

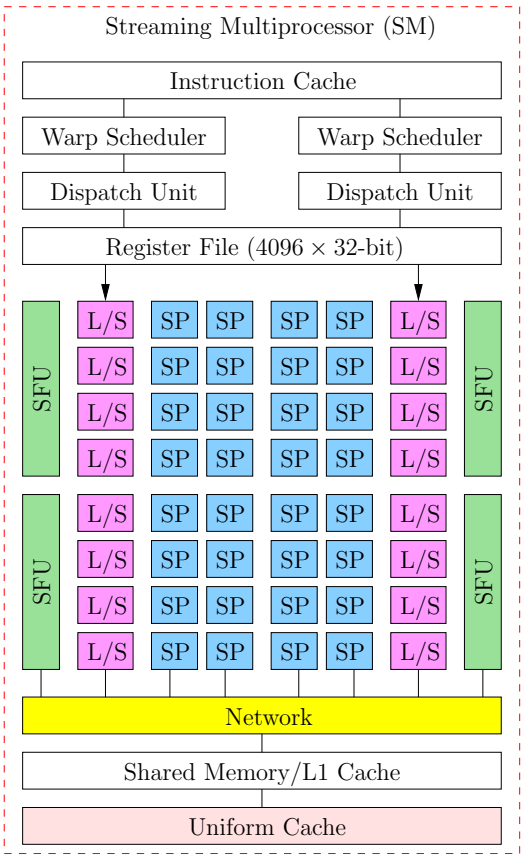


Figure 1.2: Diagram of a streaming multiprocessor in NVIDIA Tesla C2070. (SP is short for streaming processor, L/S for load/store unit, and SFU for Special Function Unit.)

to explore most of the capabilities of GPU parallelism. In CUDA programming model, illustrated in Fig. 1.3, threads are organized into thread blocks; thread blocks are further organized into grids. When a CUDA program on the host CPU invokes a kernel grid, the compute work distribution (CWD) unit enumerates the blocks of the grid and begins distributing them to SMs with available execution capacity. The threads of a thread block execute concurrently on one SM. As thread blocks terminate, the CWD unit launches new blocks on the vacated multiprocessors.

Note that it is not correct, or at least not quite precise, to view a GPU core or CUDA core as a freely controlled processing core equivalent to the ones in a multi-core CPU. Although threads within a block will actually execute on multiple CUDA cores within the SM, the number of thread blocks shall be understood as the analogy to "processor count" in parallel algorithm analysis. It is at the thread block level or SM level where internal communication is cheap and external communication becomes expensive. Therefore, focusing on the decomposition of work into a number of thread blocks with appropriate size is the most important factor in tuning the performance. 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 programming language creates its extended keywords and built-in variables. For example, in order to distinguish the parallel threads in a kernel launch, two index variable structures, `blockIdx.{x,y,z}` and `threadIdx.{x,y,z}`, are provided. They assign unique coordinates, or say IDs, to all blocks and threads in the whole grid partition.

Therefore, programmers can easily map the data partition to parallel threads, and instruct the specific thread to compute its own responsible data elements. Fig. 1.3 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. This figure shows the two-level parallelism in CUDA computing: Parallel SMs compute result blocks, and parallel threads in each block compute result elements.

CUDA also assumes that both the host (CPU) and the device (GPU) maintain their own separate memory spaces, referred to as host memory and device memory, respectively. For every block of threads, the on-chip shared memory is accessible to all threads in that same block, thus providing efficient cooperation and communication amongst threads in a block. And the global memory, i.e., DRAMs on GPU card, is accessible to all threads in all blocks. However, due the lack of large cache and the required time to refresh charges on the DRAM, the latency of reading and writing data on global memory is long. To hide this latency, it is particularly advantageous when a thread block can load a block of global data into on-chip shared memory, process it with the parallel threads, and then write the final result back out to global memory. Coalesced memory access is favored by GPU. When threads of a warp access aligned and consecutive words in memory, the hardware is able to group these accesses into aggregate transactions with the memory system, resulting in substantially higher memory throughput. For instance, a warp of 32 threads gathering from widely separated addresses will issue 32 requests to memory, while a warp reading 32 consecutive words will only issue 2 requests.

To achieve best efficiency, kernels should avoid execution divergence, where threads

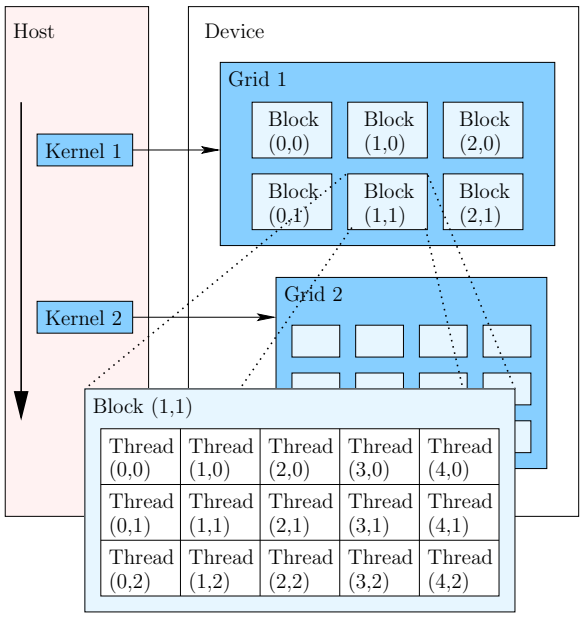


Figure 1.3: The programming model of CUDA. When Kernel 1 is launched on GPU, a 3-by-2 grid of 6 thread blocks is created, and each block in Kernel 1 is organized into a 5-by-3 array of threads.

within a warp follow different execution paths. Divergence between warps, however, introduces no performance penalty. This means the programmer should minimize the conditional switches in statements such as if-then-else.

One thing to mention is that for current GPUs, computations of double-precision float number is still slower than single-precision ones due to the limit of ALUs. Thus, for applications whose execution time is dominated by floating point computations, switching from single-precision to double-precision will decrease performance remarkably. However, this situation is being improved. Newer models of NVIDIA GPUs can provide much better double-precision performance than before.

1.3 Objectives and results of this thesis

The main objective of this thesis is to explore and improve circuit simulation algorithms, such as Monte Carlo and transient analysis, and propose new implementations on modern computing platforms. Efficient and reliable statistical analysis and parallel methodology of very large scale integrated circuits will be presented through the researches. The key contribution of these researches is the introduction and the exploration of several novel techniques for statistical analysis and circuit simulation on both time domain and frequency domain. The major achievements accomplished in this dissertation are as follows.

- **Symbolic based statistical analysis with GPU.** This work is based on a powerful graph based symbolic technique. With the proposed new algorithms, we extend the application of this technique to run parallel Monte Carlo simulation on GPU. Experiments show that our GPU MC analysis brings 2–3× speedup over HSPICE MC simulation. In another independent work, this same symbolic technique is used as a fundamental stage where a novel performance bound analysis is built upon to obtain statistical information.

- **Fast parallel power grid simulation on heterogeneous CPU-GPU platforms.** To cope with the ever increasing size of power grid circuit, model order reduction technique is applied to generate a smaller system with tolerable approximation error. Then GPU parallel computing is introduced in transient simulation of this reduced model for speedup. The fine-grained thread level parallelization of GPU on the transient simulation makes significant speedup on industrial benchmark circuits.

- **GPU based iterative GMRES solver in a variety of applications.** Krylov subspace based iterative GMRES solver is used in thermal simulation of 3D ICs. For the thermal modeling and simulation of 3D IC with liquid cooling system, we implement GPU GMRES solver suited to the characteristics of 3D IC's equivalent thermal circuit models. The computation intensive procedures in GMRES such as subspace constructions are made parallel. With well chosen preconditioners, we gain further speedup in the equation solving phase.

  Variants of GPU GMRES are also applied to PSS shooting method for RF simulation and envelope-following method for power converter simulation, in which a matrix-free technique for subspace basis vector generation is used to avoid the cost incurred from explicitly building the Jacobian matrix. We demonstrate the efficiency of our methods by comparing both accuracy and acceleration in practical design examples.

1.4 Organization of the thesis

To carry out statistical analysis on circuits, Chapter 2 and Chapter 3 both start from determinant decision diagram (DDD) based symbolic method, which generates the expressions of circuit transfer functions in frequency domain. Chapter 2 describes a novel data structure and computation algorithm for parallel DDD evaluation on GPU platforms, and hence results in faster parallel Monte Carlo simulation. In contrary to this Monte Carlo analysis, Chapter 3 deploys an optimization approach on symbolic expressions of transfer functions, and calculates the lower and upper performance bounds of magnitude and phase of the transfer function, which is called non Monte Carlo method. With this frequency

domain performance bounds and a given time domain input stimulus signal, Chapter 3 further applies uncertainty region method and signal processing techniques to determine time domain performance bounds of the output signal.

Chapter 4 to Chapter 7 focus on fast transient simulation of power grid circuits, 3D thermal circuit models, radio frequency/monolithic microwave circuits (RF/MM IC), and power converter circuits. Various GPU features are used in these works. Chapter 4 first reduces the order of power grid circuit, which is usually large and costs high computation time and memory. Then the reduced system is transferred to GPU, where large number of current sources in the grid can be evaluated independently in parallel and the linear algebra operations such as triangular solves, matrix-vector multiplications, and vector additions can also be accelerated by GPU fine-grained parallelism.

Chapter 5 first describes the background of state-of-the-art modeling of 3D IC with liquid cooling system, and then derives the thermal circuit model for this thermal system. To accelerate the solving of large equations in the transient simulation, GPU based GMRES iterative solver is applied. We also investigate good preconditioners for GMRES in order to accelerate the iterative solver. Various parts in the solver are made parallel on GPU and the programming details are talked about in this chapter.

Chapter 6 uses a modified GMRES solver in shooting problem for steady state analysis of RF/MM circuits. A core problem related to shooting analysis is the solving of a Newton update equation. The benefit of using this modified GMRES as solver of Newton update is the capability to calculate Krylov subspace independently. Hence, parallel GPU thread blocks work on these subspaces simultaneously. The mathematics and formu-

lation of the shooting problem is provided, and detailed GPU algorithm implementation is also discussed. With a similar Newton update problem in the envelope-following method, Chapter 7 develops a tool for the simulation of power converter circuits by using GPU GMRES solver and Gear-2 integration scheme, both of whom help deliver speedup and ensure accuracy.

Finally, Chapter 8 concludes the thesis with brief summaries of the works, and gives proposal for some future works.

Chapter 2

DDD based Monte Carlo analysis on GPU

This chapter shows a new parallel statistical analysis method for large analog circuits using determinant decision diagram (DDD) based graph technique based on GPU platforms. DDD-based symbolic analysis technique enables exact symbolic analysis of vary large analog circuits. But we show that DDD-based graph analysis is very amenable for massively threaded based parallel computing based on GPU platforms. We design novel data structures to represent the DDD graphs in the GPUs to enable fast memory access of massive parallel threads for computing the numerical values of DDD graphs. The new method is inspired by inherent data parallelism and simple data independence in the DDD-based numerical evaluation process. Experimental results show that the new evaluation algorithm can achieve about one to two order of magnitudes speedup over the serial CPU based evaluations and 2–3 times speedup over numerical SPICE-based simulation method

on some large analog circuits.

2.1 Introduction

It is well known that analog and mixed-signal circuits are very sensitive to the process variations as transistor matching and regularities are required. This situation becomes worse as technology continues to scale to 90 nm and below owing to the increasing process-induced variability [9,10]. For example, due to an inverse-square-root-law dependence with the transistor area, the mismatch of CMOS devices escalates by two times for each process generation less than 90 nm [1, 2]. To consider the impacts of process variations on circuit performance, Monte Carlo based statistical approaches are the most reliable solutions to this problem. But the prohibitive computational cost of Monte Carlo method prevents it from solving large analog circuits.

Modern computer architecture has shifted towards designs that employ multiple processor cores on a chip, so called multi-core processor [11,12]. GPUs are one of the most powerful many-core computing systems in mass-market use. For instance, NVIDIA Tesla T10 chips have a peak performance of over 1 TFLOPS versus about 80–100 GFLOPS of Intel i5 series Quad-core CPUs [13]. In addition to the primary use of GPUs in accelerating graphics rendering operations, there has been considerable interest in exploiting GPUs for general purpose computation (GPGPU) [14]. Accordingly, the introduction of new parallel programming interfaces for general purpose computations, such as Computer Unified Device Architecture (CUDA) [15], Stream SDK [16] and OpenCL [17], has made GPUs powerful and attractive for developing high-performance tools of solving practical engineering prob-

lems. Parallelization on GPU platforms is an emerging strategy to improve the efficiency of Monte Carlo based statistical analysis method. But traditional numerical simulators such as SPICE are based on LU decomposition, and are difficult to be parallelized on GPUs due to irregular memory access and huge memory-intensive operations.

Graph-based symbolic technique is a viable tool for calculating the behavior or characteristic of analog circuits [18]. The introduction of determinant decision diagrams based symbolic analysis technique (DDD) allows exact symbolic analysis of much larger analog circuits than all the other existing approaches [19,20]. Furthermore, with hierarchical symbolic representations [21,22], exact symbolic analysis via DDD graphs essentially allows the analysis of arbitrary large analog circuits. Once the small-signal characteristics of circuits are presented by DDDs, evaluation of DDDs, whose CPU time is proportional to the size of DDDs, will give exact numerical values. One important observation is that the DDD-based simulation is very amenable for parallel computing as the main computation is distributed to each DDD node (via graph traversals) and the data dependency is very simple due to the simple binary graph structure.

In this chapter, we develop efficient parallel graph-based simulation technique based on GPU computing platforms for Monte Carlo based statistical analysis of analog circuits. We design novel data structures to represent the DDD graphs in the GPUs to enable fast memory access of massive parallel threads for computing the numerical values of DDD graphs. The new method is inspired by inherent data parallelism and simple data independence in the DDD-based numerical evaluation process. Experimental results show that the new evaluation algorithm can achieve about one to two orders of magnitudes

speedup over the serial CPU based evaluations of analog circuits and 2–3 times speedup over numerical SPICE-based simulation method on some large analog circuits. Besides, the proposed parallel techniques can be used for the parallelization of many decision diagrams based applications, such as logic synthesis, optimization and formal verifications, which are based on binary decision diagrams (BDDs) and its variants [23, 24].

Section 2.2 outlines DDD-based symbolic analysis techniques. Then, we introduce the flow of the proposed GPU Monte Carlo simulation. Section 2.4 describes the proposed GPU parallel algorithm, followed by several numerical examples in section 2.5. Last, Section 2.6 makes a summary.

2.2 DDD symbolic analysis

In this section, we first provide a brief overview of determinant decision diagrams (DDDs) [19] and its application to symbolic analysis.

The DDD technique uses directed binary graphs to represent a determinant. The paths in the graph represent those product terms in the determinant. Since the number of paths in a graph can be much larger than the number of nodes, DDD representation enables symbolic analysis of much larger analog circuits than before [19]. The concept of DDD representation is briefly reviewed as follows. The determinant of a matrix can be expressed as the symbolic product terms from the subset of all elements in the matrix. For

example, consider the following matrix determinant.

$$\det(\mathbf{M}) = \begin{vmatrix} a & b & 0 & 0 \\ c & d & e & 0 \\ 0 & f & g & h \\ 0 & 0 & i & j \end{vmatrix} = adgj - adhi - aefj - bcgj + cbih \tag{2.1}$$

We can express each element using a node in a diagram and each product term using a path going through four nodes. For every node, it has a value of itself and a sign attached to it. The sign of each product term is decided by multiplying the sign of every node in the corresponding path. When the matrix $\mathbf{M}$ is a circuit matrix (such as modified nodal analysis (MNA) matrix), the value for each node represents the RLC value for the element in the MNA matrix. The diagram for the above matrix is shown in Fig. 2.1.

A DDD is a signed, rooted, directed acyclic graph with two terminal nodes, namely the 0-terminal node and the 1-terminal node. Each non-terminal DDD node is labeled by a symbol in the determinant denoted by a_i (a to j in Fig. 2.1), and a positive or negative sign denoted by $s(a_i)$. It originates two outgoing edges, called 1-edge and 0-edge. Each node a_i represents a symbolic expression $D(a_i)$ defined recursively as follows:

$$D(a_i) = a_i \cdot s(a_i) \cdot D_{a_i} + D_{\overline{a}_i}, \tag{2.2}$$

where D_{a_i} and $D_{\overline{a}_i}$ represent, respectively, the symbolic expressions of the nodes pointed by the 1-edge and 0-edge of a_i. The 1-terminal node represents expression 1, whereas the 0-terminal node represents expression 0. For example, node h (in Fig. 2.1) represents expression h, and node i represents expression $-ih$, and node g represents expression $gj-ih$. We also say that a DDD node g represents an expression defined the DDD sub-graph rooted

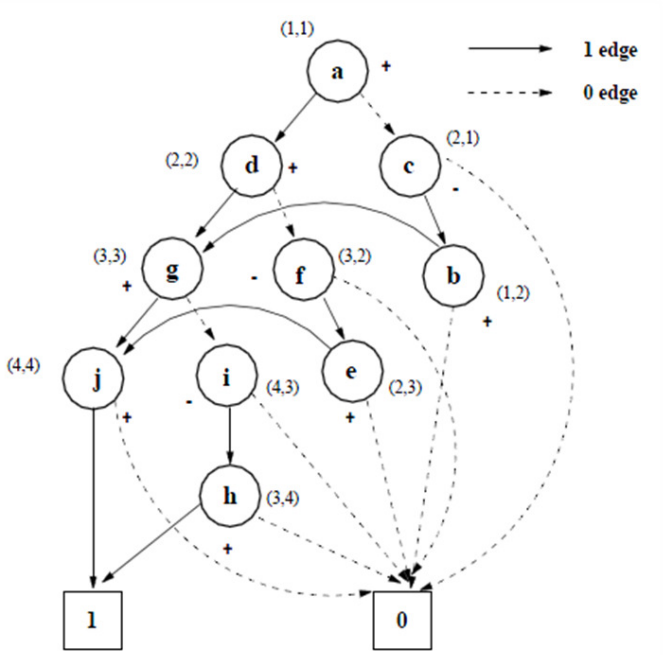


Figure 2.1: DDD representation for matrix **M**.

at g. For each node, there are two values, v_{self} and v_{tree}. In (2.2), v_{self} represents the value of the element itself, which is D_{a_i}; while the v_{tree} represents the value of the whole tree (or sub-tree), which is $D(a_i)$.

A 1-path in a DDD corresponds with a product term in the original DDD, which is defined as a path from the root node (a in our example) to the 1-terminal including all symbols and signs of the nodes that originate all the 1-edges along the 1-path. In our example, there exist five 1-paths representing five product terms: $adgj$, $adhi$, $aefj$, $bcgj$, and $cbih$. The root node represents the sum of these product terms. Size of a DDD is the number of DDD nodes, denoted by |DDD|.

Once a DDD has been constructed, its numerical values of the determinant it represents can be computed by performing the depth-first type search of the graph and

performing (2.2) at each node, whose time complexity is linear function of the size of the graphs (its number of nodes). The computing step is called the evaluation of D, where D is a DDD root. With proper node ordering and hierarchical approaches, DDD can be very efficient to compute transfer functions of large analog circuits [19, 22].

Now we use a circuit example to illustrate the idea of DDD. A simple RC filter circuit is shown in Fig. 2.2. Its MNA formulation can be written as $\mathbf{Y} \cdot \mathbf{v} = \mathbf{i}$, where $\mathbf{Y}$ is the MNA matrix, i.e.,

$$\begin{bmatrix} \frac{1}{R_1} + sC_1 + \frac{1}{R_2} & -\frac{1}{R_2} & 0 \\ -\frac{1}{R_2} & \frac{1}{R_2} + sC_2 + \frac{1}{R_3} & -\frac{1}{R_3} \\ 0 & -\frac{1}{R_3} & \frac{1}{R_3} + sC_3 \end{bmatrix},$$

$\mathbf{i} = [I(s), 0, 0]^{\mathrm{T}}$ is the right-hand side vector of current stimulus, and the vector of nodal voltages to be solved is $\mathbf{v} = [v_1(s), v_2(s), v_3(s)]^{\mathrm{T}}$.

We view each entry in the circuit matrix as a distinct symbol, and rewrite its system determinant in the left part of Fig. 2.3. Then its DDD representation is shown in the right part.

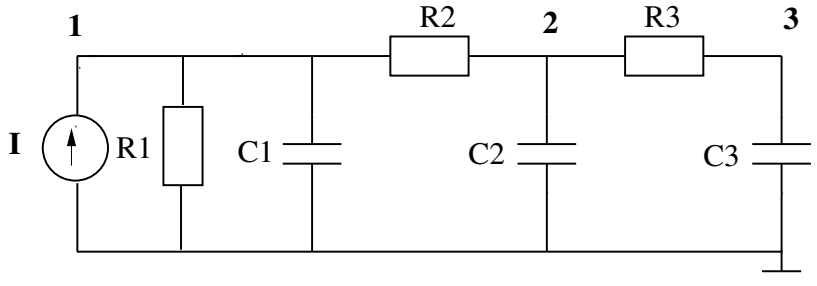


Figure 2.2: A RC filter circuit.

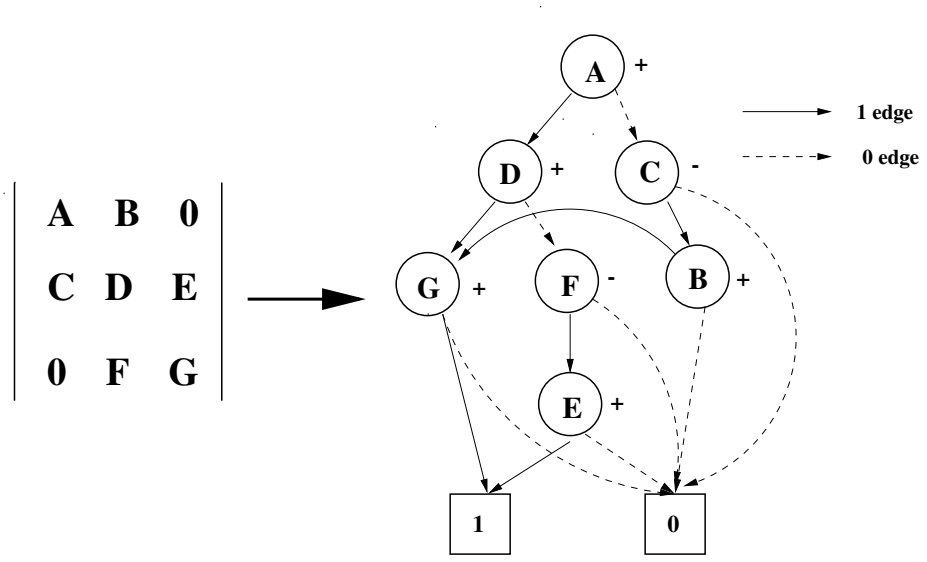


Figure 2.3: A matrix determinant and its DDD representation.

2.3 Parallel evaluation of DDD

In this section, we first provide an overview of our graph-based GPU-based parallel statistical analysis before the detailed explanation.

As mentioned before, in DDD-based analysis, computing numerical value of the determinant of the DDD essentially boils down to the depth-first traversal of the graph. The data dependency is very simple: a node can be evaluated only after its children are evaluated. Such dependency implies the parallelism where all the nodes satisfying this constraint can be evaluated at the same time. Also, in statistical frequency analysis of analog circuits, evaluation of a DDD node at different frequency points and different Monte Carlo runs can be performed in parallel. We show that all those parallelism will be explored by the new statistical analysis approach on GPU platforms.

2.3.1 The overall algorithm flow

Fig. 2.4 gives the overall flow of our statistical method. The whole algorithm has two main parts, the CPU part (host) and GPU part (device) as clearly marked in the figure. CPU part mainly reads the netlist, generate the original DDD tree structures and builds new continuous DDD vector array structure (for GPU) and outputs the final numerical results. GPU part takes care of the main parallel DDD evaluation and communicates with CPU. The new program reads input netlist containing variation information of the relevant circuit devices. Then, the analyzer builds the MNA (modified nodal analysis) matrix and DDD binary tree data structure [19] as shown in step ①.

2.3.2 Continuous and levelized DDD structure

To prepare for the GPU computing, we need to build new data structures from the original binary tree DDD structures. This will be done in the CPU as the construction only needs to be performed once and traversal of original DDD linked trees is still sequential in nature and will be difficult to handle in GPU, as labeled ② in Fig. 4.2.

For GPU computing, the main challenge is to allow fast memory access by threads or reduce memory traffic as much as possible by using shared memory (or texture memory) within blocks so that GPU cores can be busy all the time. In GPU, fast global memory access by threads can be done by coalesced memory access where a half warp (or a warp) of threads (16 or 32 threads respectively) can read their data from the global memory in one read access. Coalesced memory access requires that data are arranged continuously in memory and consecutive with respective to involved thread indexes. As a result, we need

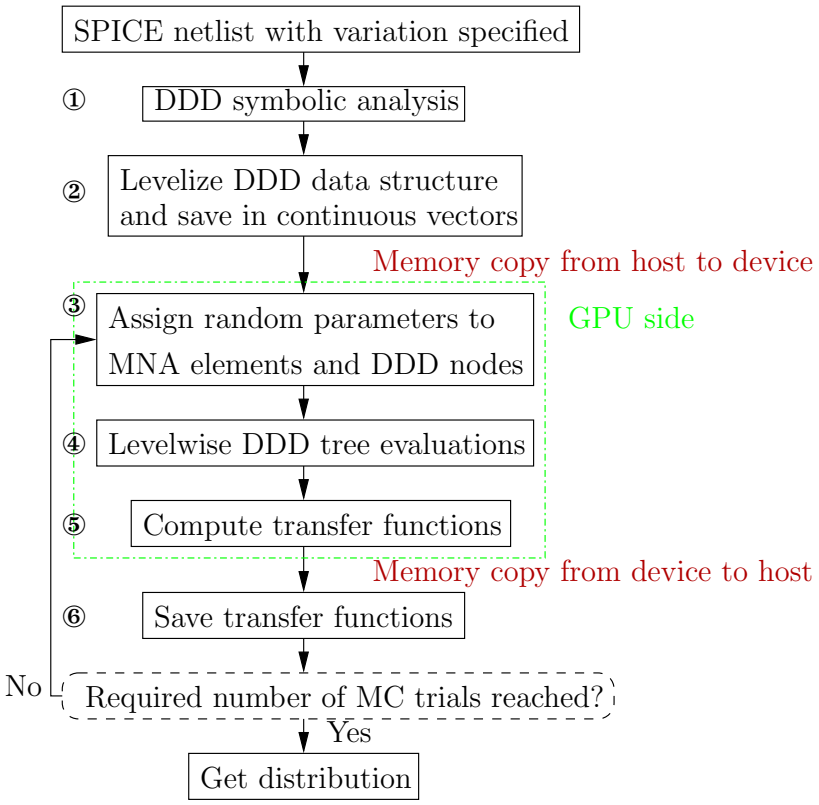


Figure 2.4: The flow of GPU-based parallel Monte Carlo analysis.

to remap the linked DDD trees into a memory-continuous data structure.

The second issue is that we do not need to perform the DDD node evaluation for all the DDD nodes. Only those nodes whose children have been evaluated should be computed by threads (one thread for one DDD node). This can be done by sorting the DDD nodes by their *level*. Two DDD nodes have same level if they have the same number of edges on their longest path to the *1-terminal*. For instance, node g and node f has the same level in Fig. 2.1. DDD nodes at the same level can be computed in parallel in GPU. As we can see, the largest level of DDD nodes will be bounded by the numbers of non-zeros in a determinant. But practically, number of level can be much less than the number of non-zeroes. For instance, we have 6 levels in the DDD shown in Fig. 2.1 versus 10 nonzero elements.

In the new DDD structure, all the DDD nodes at the same level will be put in continuous and consecutive memories (mainly the v_{self} and future v_{tree} values) and be assigned to threads (one DDD node per thread) at the same time (one kernel launch). The level assignment can be done by simple depth-first traversal of the DDD graph. After this, we can allow the continuous memory for all the DDD nodes for one level starting from the lowest level until the highest level. We use the DDD example in Fig. 2.1 again to illustrate the new data structure shown in Fig. 2.5. For each value associated with a DDD node such as its value (v_{self}), left child index, right child index, level index, sign (not shown), a linear array will be generated based on the level indexes of DDD node. For example, node b in Fig. 2.1 becomes the 7-th element in the vector, and the index of its children, g and 0-terminal, are 4 and -2 accordingly. Note that, by our definition, 1-terminal's index is

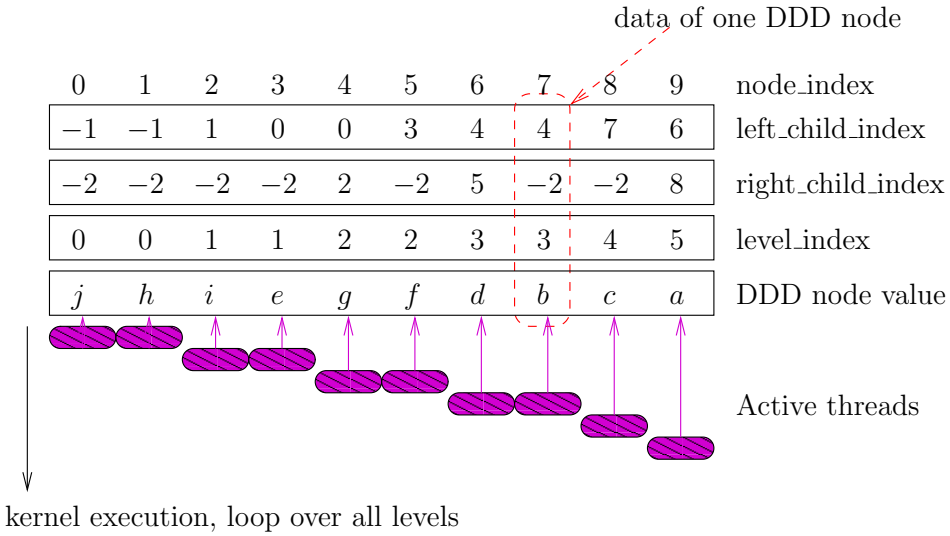


Figure 2.5: Levelized continuous storage of a DDD, and levelwise GPU evaluation of the DDD in Fig. 2.1.

-1, and 0-terminal's is -2. Those arrays then will be copied into GPU memory for future DDD evaluation.

Fig. 2.5 also shows the execution pattern of GPU threads during DDD evaluation where we start with the DDD nodes in the lowest level and continue one level at a time until we hit the highest level. Since all nodes of the same level have been reorganized into one continuous memory segment, the active GPU threads working on them can achieve coalesced read/write access and also minimize the occurrence of branch divergence. As we observed, consecutive and levelwise data format improves the performance of GPU by 2–3$\times$ for large circuits.

2.4 Parallel Monte Carlo analysis

2.4.1 Random number assignment to MNA elements and DDD nodes

For statistical analysis, we need to generate variations from devices into the elements of the determinant and then into the data in the continuous DDD data structure. Due to MNA formulation, each device may appear 4 positions in a MNA matrix. Hence we track and save the MNA stamp patterns of circuit devices, and also their locations in DDD, during DDD construction. These data are transferred to GPU texture memory as texture memory is read-only and can be accessed much faster than GPU global memory.

Next, in random number assignment, CURAND library is used to generate variations on nominal values of circuit parameters in GPU kernel function. Since one device parameter may appear at 4 positions in the MNA matrix, we need to make sure that in each MC run the variation of this parameter are reflected properly at the 4 corresponding DDD nodes. This is done in Line 2 and Line 3 of the pseudo-code in Algorithm 1. The variations introduced in our experiments are Gaussian random values, whose means and deviations can be specified by users from input netlist.

Note that since we perform the frequency domain analysis, we need to evaluate the MNA and DDD on all frequency points of interest. To enable coalesced memory access to compute DDD values for many frequencies, as Line 5 and Line 9, the DDD continuous structure will be further changed so that all frequency responses of the same element or node reside in consecutive memory addresses. We observe that this frequency related calculation is very suitable for intra-block GPU computing as all the threads in a block can share the same DDD information (except for the frequency values).

Algorithm 1 Parallel random value assignment for DDD nodes

1: **for** all Monte Carlo runs **do** *// launch threads in grids*

2: Assign random values to involved device parameters and stamp MNA elements.

3: Save each DDD node's admittance, capacitance, and inductance components as $R[k] = \{g, c, l\}$.

4: **for** all DDD nodes **do** *// launch threads in grids*

5: Load frequency values to f.

6: **for** all frequencies $f[i]$ **do** *// launch threads in a block*

7: $v_{\text{self}}[i] = R[k].g + j \cdot (R[k].c \cdot f[i] + R[k].l/f[i])$

8: **end for**

9: Save v_{self}.

10: **end for**

11: **end for**

In GPU, the threads are organized into blocks, and blocks are formed into grids. For Tesla C2070, both the configurations of blocks and grids can have three dimensions, and maximum size of each dimension of a grid is 65,535. C2070 allows each block to have as many as 1024 threads. (This upper limit varies with different GPU families from NVIDIA.) Threads in a block can communicate via shared or texture memory and can be explicitly synchronized. In our problem, the dimension of the grid is set to $N_{MC} \times |DDD|$, i.e., the number of Monte Carlo runs times the number of DDD nodes (assume that it is less than 64K) and each block of this grid contains TILE_DIM threads, where TILE_DIM is a multiple of 16 to enable coalesced access on neighboring frequency responses and is also set with consideration of available GPU resources per block. In practice, we set TILE_DIM as 256. So we can allow each block to compute 256 frequency responses for one DDD node. Notice that all the three FOR loops in Algorithm 1 will be replaced by massive thread launches in parallel. The two outer loops are parallelized at grid level, and the innermost loop is parallelized at block level. Hence, the DDD node values v_{self} are computed for all Monte Carlo runs and all frequency points in their respective blocks and threads (Line 7). If number of frequency points is larger than TILE_DIM, the innermost FOR loop will be kept inside the kernel function. But instead of loop over each frequency point, we loop over TILE_DIM frequency points every time.

The number of Monte Carlo runs in each kernel launch is determined by the GPU specification and the allocated resources, such as global memory, to each Monte Carlo calculation. For a typical μA741 circuit whose DDD contains 6,205 nodes and 2,400 evaluated frequency points, the Tesla C2070 can allow 20 Monte Carlo runs in parallel. In case more

runs are required, the steps from ③ through ⑥ in Fig. 4.2 are repeated as many times as needed.

2.4.2 Parallel frequency sweep of MC simulation

The evaluation of DDD is a process that computes the final numerical value of the determinant it represents. This procedure is labeled with ④ in Fig. 4.2. As we previously discussed in Section 2.3.2, the data structure of DDD has been remapped to GPU friendly continuous and consecutive arrays and are sorted by level to enhance efficiency of evaluation.

Algorithm 2 Parallel Monte Carlo evaluation of DDD

```
1:  for level from Level-0 to top_level do    // CPU host iteration
2:    for all Monte Carlo runs do    // launch threads in grids
3:      for all DDD nodes do    // launch threads in grids
4:        if node.level == level then
5:          Load v_self of the current node, and v_tree of its children.
6:          for all frequencies do    // launch threads in a block
7:            Evaluate v_tree for the current node by Eq. (2.2) on all frequencies.
8:          end for
9:          Save current node's v_tree.
10:       end if
11:     end for
12:   end for
13: end for
```

Similar to the GPU calculation of DDD node values mentioned in previous subsec-

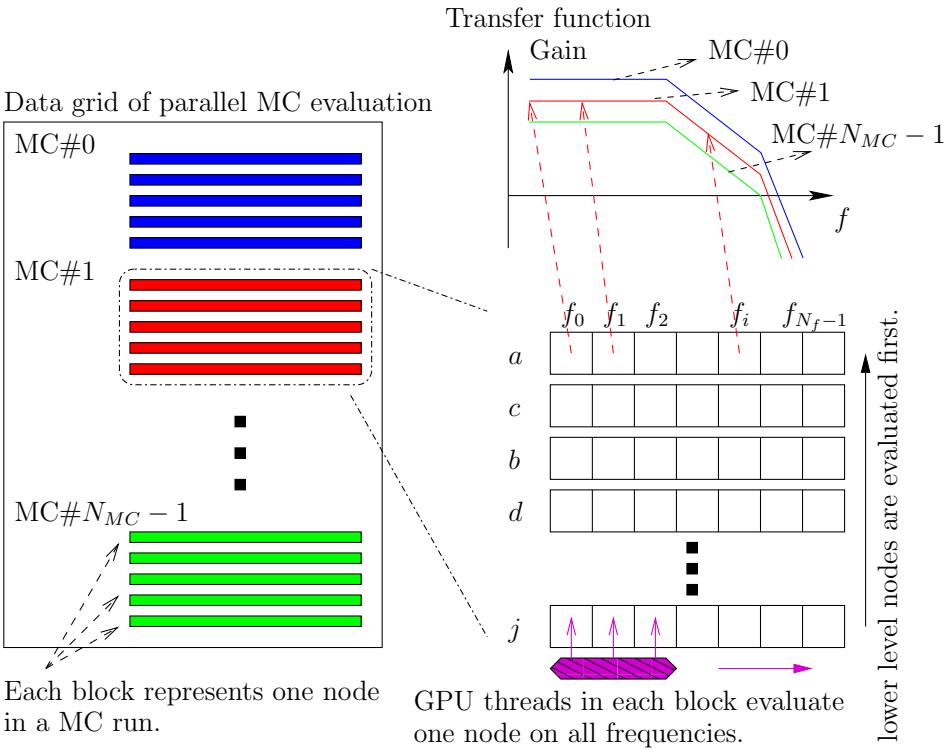


Figure 2.6: GPU parallel evaluation of the DDD in Fig. 2.1.

tion, we also launch independent blocks for different Monte Carlo runs and different DDD nodes, and use each thread block to calculate values of each node's v_{tree} for all frequency points, which is depicted in Fig. 2.6.

Algorithm 2 lists the main flow of this algorithm. To ensure that the nodes are evaluated from bottom to top, the first FOR loop iterates the level index from 0 to the maximum level in the DDD, and launches kernel function on the DDD nodes of the specific level, one at a time. Note that we keep this FOR loop in CPU control, instead of moving it inside the GPU kernel, in order to accomplish inter-block synchronization. This is necessary because we deploy the evaluation of different nodes in different thread blocks, and, if there is no synchronization, it is possible that a node of higher level gets evaluated before its children. Moreover, CUDA only provides synchronization among threads in a block. The kernel has to be finished if all blocks in the kernel grid are required to be synchronized. Therefore, in our implementation, the index of current level is passed into the kernel function as an argument, and the kernel will evaluate those thread blocks with the same level indicated by the argument index.

The coalesced memory access to the node's v_{self} and its children's v_{tree} values are also ensured in the load and save operations in Line 5 and Line 9. During the evaluation of the current node on all frequencies, the k-th thread will work on the k-th frequency, and all threads in a warp execute the same code path. Consequently, such a kernel launching exhibits a highly data intensive pattern, and reduces global memory traffic at the same time.

2.5 Experimental results

To show the performance of the proposed GPU parallel Monte Carlo simulation, we test the program on several industrial benchmark circuit netlists. For running time comparisons, we also measure the time cost by the CPU version of DDD evaluation and HSPICE.

All of our programs are implemented in C++, with NVIDIA CUDA for the GPU computation part. All running time are sampled from a Linux server with a 2.4 GHz Intel Xeon Quad-Core CPU, and 36 GBytes memory. The GPU card installed on this server is Tesla C2070, which contains 448 cores running at 1.15 GHz and up to 5 GBytes global memory.

Now let us investigate one typical example in detail. Fig. 2.7 shows the schematic of a μA741 circuit. This bipolar opamp contains 26 transistors and 11 resistors. DC analysis is first performed by SPICE to obtain the operation point, and then small-signal model, shown in Fig. 2.8, is used for DDD symbolic analysis and numerical evaluation. The AC analysis is performed with the variation of several circuit components for Monte Carlo simulation. Several Monte Carlo samples of the magnitude response are plotted in Fig. 2.9. The 3-dB bandwidth of all the statistics is calculated and shown in the histogram in Fig. 2.10. In this example, the nominal 3-dB frequency is 1.2 kHz. As we can observe from Fig. 2.10, the histogram of the bandwidth frequency is similar to the Gaussian distribution.

Next, we study the speedup and scalability of the GPU and CPU DDD based Monte Carlo simulations. The measurements of time taken by both programs running on the same RC tree circuit are shown in Table 2.1, where different snumber of Monte Carlo runs

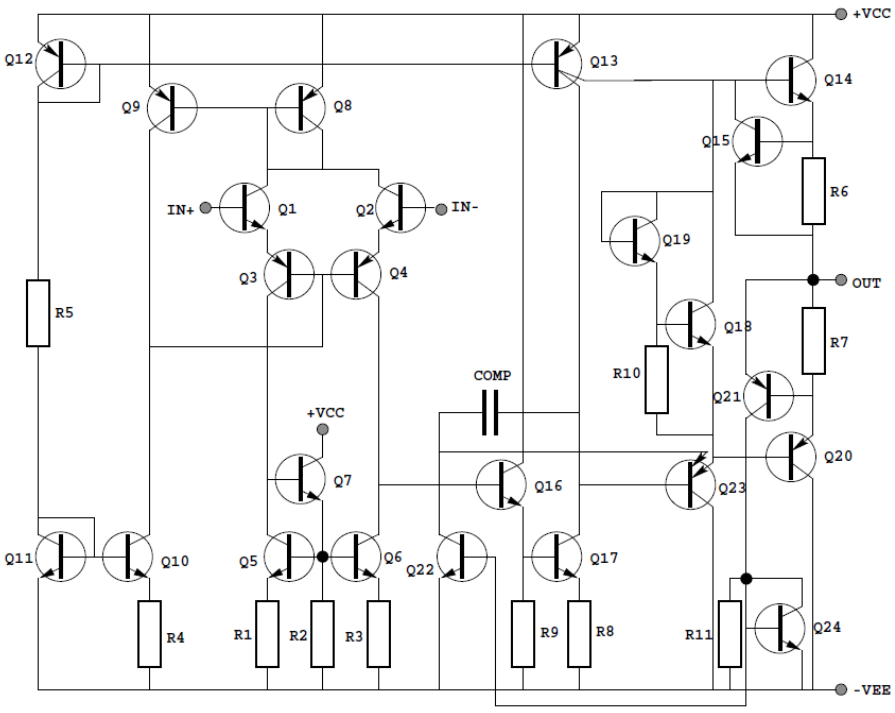


Figure 2.7: The circuit schematic of μA741

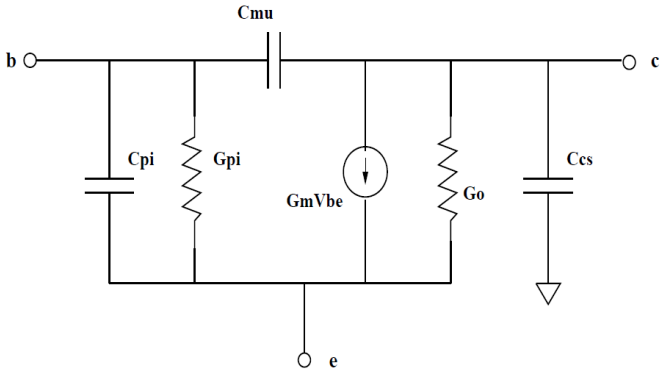


Figure 2.8: The small signal model for bipolar transistor

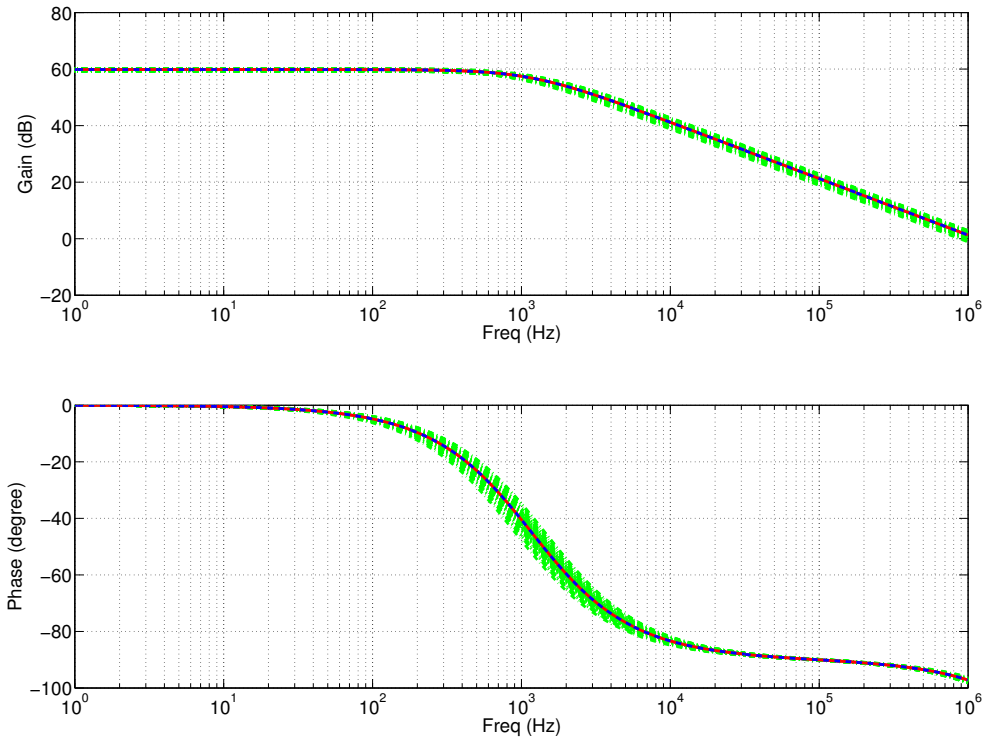


Figure 2.9: The cluster of frequency responses of the tested μA741 circuit. Red curve is the CPU golden result, blue curve is the GPU result, and all green curves are GPU MC samples from the parallel computation.

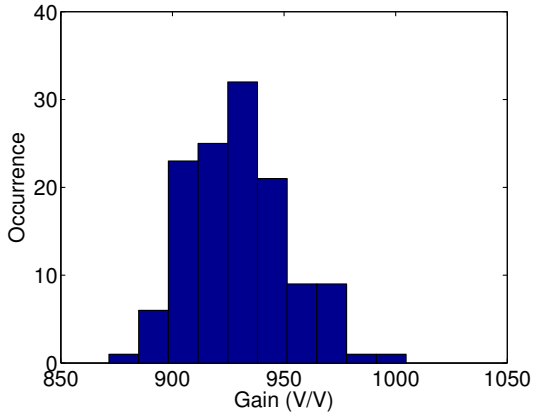


Figure 2.10: Histogram diagram of the gain samples of μA741 at frequency 314 Hz.

Table 2.1: Performance comparison of CPU serial and GPU parallel DDD evaluation for RC tree circuit with different MC runs.

# MC Runs	GPU time (s)	CPU time (s)	Speedup
1	1.98	23.0	11
2	2.08	46.2	22
4	2.21	90.5	41
8	2.50	183.8	73
16	3.03	364.1	120
32	4.76	725.3	152
64	8.68	1442	166
128	17.42	2910	167

are tested. It is obvious that the speedup of GPU method over the CPU one is significant. Also, when the number of Monte Carlo runs increases, GPU running time does not multiply as fast as the CPU version does, provided that the GPU resources can accommodate parallel execution of these Monte Carlo evaluations in one kernel launch. Hence, in this way, all the GPU streaming multiprocessors are kept busy and the throughput is maximized, which results in a striking speedup over the CPU serial version.

Last, we list the results of all benchmark tests in Table 2.2. The information of the circuits and their DDD representation is also included in the same table. The 2nd column and the 3rd column record number of interconnect nodes and number of devices in circuits.

Table 2.2: Performance comparison of Monte Carlo simulations using the proposed GPU method, its CPU counterpart, and HSPICE.

circuit name	# cir. nodes	# cir. devices	|DDD|	# DDD prod. terms	GPU time (s)	CPU time (s)	HSPICE time (s)
bigtst	32	112	642	2.68×10^7	19.7	3143	38.4
ccstest	9	35	109	260	0.80	108	2.5
rlctest	9	39	119	572	1.05	145	2.6
vcstst	12	46	121	536	0.73	104	3.8
ladder21	22	64	64	28657	2.10	365	5.1
ladder100	101	301	301	9.27×10^{20}	30.6	3965	42.5
rctree1	40	119	211	1.15×10^8	5.55	928	11.3
rctree2	53	158	302	4.89×10^{10}	17.42	2910	46.1
μA741	23	89	6205	363914	59.1	6243	73.6

In the 4th column, |DDD| stands for number of DDD nodes in the generated DDD graph, which is equal to number of non-zero elements in the MNA matrix. The column "# DDD prod. terms" lists number of product terms in the determinant. The last three columns summarize the run-time of GPU parallel algorithm, serial algorithm and the HSPICE. The number of Monte Carlo runs for all tests is set to 128. It is clear from this table that the GPU-accelerated version outperforms its CPU counterpart, and also achieves 2–3 times speedup over the commercial HSPICE on a variety of test circuits.

2.6 Summary

A new parallel statistical analysis method for large analog circuits using determinant decision diagram (DDD) based graph technique is proposed. To make it amenable for massively threaded based parallel computing GPU platforms, we designed novel data structures to represent the DDD graphs in the GPUs to enable fast memory access of massive parallel threads for computing the numerical values of DDD graphs. The new method is inspired by inherent data parallelism and simple data independence in the DDD-based numerical evaluation process. Experimental results show that the new evaluation algorithm can achieve about one to two order of magnitudes speedup over the serial CPU based evaluations and 2–3× speedup over numerical SPICE-based simulation method on some large analog circuits.

Chapter 3

Performance bound analysis of analog circuits in frequency and time domain considering process variations

3.1 Introduction

Analog circuit designers usually perform a Monte Carlo (MC) analysis to analyze the stochastic mismatch and predict the variational responses of their designs under faults. As MC analysis requires a large number of repeated circuit simulations, its computational cost is very expensive. Although some fast MC methods have been applied for statistical and yield analysis recently [25, 26], we remark that MC and its variants still remain the popular

approaches for statistical analysis and optimization for analog/mixed-signal methods at current stage. But more efficient statistical analysis techniques, especially non-Monte Carlo methods, are still highly desirable. The work described in this chapter takes some initiative efforts toward this direction. We do not aim to solve all the existing problems. Instead, we try to look at the basic problem for statistical analysis — finding the performance bounds of linear or linearized analog circuits for given variation parameters — which will lay the foundation for future bound performance analysis for general nonlinear analog/mixed-signal circuits.

Bound analysis or worse case analysis of analog circuits under parameter variations has been studied in the past for fault-driven testing and tolerance analysis of analog circuits [27–29]. Among them, sensitivity analysis [30], sampling method [31], and interval arithmetic based approaches [27–29, 32] have their advantages in well suited scenarios. However, sensitivity based methods can't give the worse case in general, the sampling based method is limited to a few variables, and interval arithmetic methods have the notoriety of overly pessimism. Recently, worst-case analysis of linearized analog circuits in frequency domain has been proposed [32], where Kharitonov's functions [33] were applied to obtain the performance bounds in frequency domain, but no systematic method was proposed to obtain variational transfer functions. Recently, authors in [34] applied an optimization based method to compute the bounds. But still, no systematic method was proposed to obtain variational performance objective functions from the circuit netlist.

This work proposes a new performance bound analysis of analog circuits considering the process variations. The new method employs several techniques to compute

the response bounds of analog circuits in both frequency domain and time domain. The overall algorithm consists of several steps. First, the new method models the variations of component values as intervals measured from tested chips and manufacture processes. Then, determinant decision diagram (DDD) based symbolic analysis is applied to derive the exact symbolic transfer functions from linearized analog circuits. After this, we formulate the bound problem into nonlinear constrained optimization problem, where the objective functions are the magnitude or phase of the transfer function, subject to the linear constraints, which are the ranges of process variation parameters. The nonlinear constrained optimization problems are then solved by the active-set algorithm, which is a general nonlinear optimization method. The optimization is solved on each frequency point of interest. The maximum and minimum value returned by the optimization will compose the lower and upper bounds of the frequency domain response. The important feature of the proposed method is that the bounds computed in this way are very accurate without the over-conservativeness issues, which are suffered by some existing approaches such as internal or affine interval based methods. To compute the time domain bound, we propose generalized time domain bound analysis technique, or TIDBA, in which time domain bounds for general signal inputs can be computed based on the given frequency domain responses. This represents a major improvement over the existing method [35]. Experimental results from several analog benchmark circuits show that the proposed method gives the correct bounds verified by the Monte Carlo analysis while it delivers one order of magnitude speedup over the Monte Carlo method in both frequency and time domain bound analysis. As an application of the bound analysis, we also show analog circuit yield analysis results.

The rest of this chapter is organized as follows: Section 3.2 gives a review on determinant decision diagram based symbolic generation of transfer functions. We present our proposed frequency domain performance bound analysis method using nonlinear constrained optimization in Section 3.3. Then Section 3.4 introduces the proposed time domain bound analysis technique. Section 3.5 shows the experimental results. Finally, Section 3.6 gives the summary.

3.2 Variational transfer functions due to process variations

This section shows the concept of variational transfer functions based on the symbolic expressions generated by DDD, which is reviewed in Section 2.2 in Chapter 2.

In order to compute the symbolic coefficients of the transfer function in different powers of s, the original DDD can be expanded to the s-expanded DDD [20]. Specifically, to obtain the transfer function $H(s)$, we can build the s-expanded DDD [20] as follows:

$$H(s, p_1, \ldots, p_m) = \frac{\sum_{i=0}^{m} a_i(p_1, \ldots, p_m)s^i}{\sum_{j=0}^{n} b_j(p_1, \ldots, p_m)s^j} \tag{3.1}$$

where coefficients $a_i(p_1, \ldots, p_m)$ and $b_j(p_1, \ldots, p_m)$ are presented by each root in s-expanded DDD graphs and $p_1, \ldots, p_m$ are m circuit variables. Notice that $H(s, p_1, \ldots, p_m)$ is a nonlinear function of p_i, $i = 1, \ldots, m$.

Here, we assume that each circuit parameter p_i is a random variable with a range. Let $s = j\omega$, instead of getting a nominal transfer function $H(j\omega) = H^0(\omega)e^{j\theta(\omega)}$, we will

obtain a variational transfer function with bounded magnitude and phase regions, i.e.,

$$H_{\mathrm{l}}^0(\omega) \leq H^0(\omega) \leq H_{\mathrm{u}}^0(\omega), \tag{3.2}$$

$$\theta_{\mathrm{l}}(\omega) \leq \theta(\omega) \leq \theta_{\mathrm{u}}(\omega). \tag{3.3}$$

where $H_l^0(\omega)$ and $H_u^0(\omega)$ are the lower and upper bounds of the magnitude, and $\theta_l(\omega)$ and $\theta_u(\omega)$ are the lower and upper bounds of phase.

3.3 Computation of frequency domain bounds

In this section, we first describe the performance bounds in the frequency domain for a circuit under process variation, and then we propose the optimization based method to compute the bounds, which is very general and accurate.

The evaluation of the transfer function gives a complex valued result, $H(j\omega) = H^0(\omega)e^{j\theta(\omega)}$, where the magnitude $H^0(\omega) = |H(j\omega)|$ and the phase angle $\theta(\omega) = \angle H(j\omega)$ are real values. The goal of our frequency response bound analysis is to derive the bounds of magnitude and phase for $H(j\omega)$, shown in the inequalities (3.2) and (3.3). Hence, in the presence of process variation, the signal is also perturbed from its nominal behavior, and is usually bounded between its minimum and maximum limits.

We start with a specific example to look at this problem. Fig. 3.1 shows the circuit diagram of an RLC ladder circuit, where the three components are all in series with the voltage source. In frequency domain, capacitors and inductors are analyzed as complex impedances, and assuming the voltage on the capacitor is our observation, then the transfer

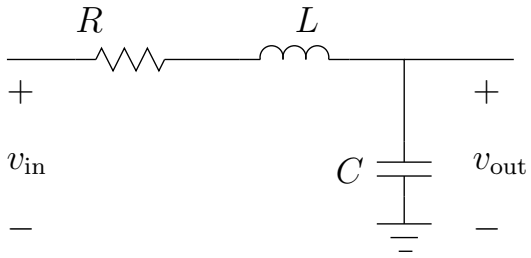


Figure 3.1: An RLC ladder circuit.

function of this circuit is

$$H(j\omega, R, C, L) = \frac{v_{\text{out}}(j\omega)}{v_{\text{in}}(j\omega)} = \frac{\frac{1}{j\omega C}}{R + j\omega L + \frac{1}{j\omega C}}. \tag{3.4}$$

We assume in this example only capacitor and inductor are under process variation. For their nominal parameters $C = 1$ μF and $L = 1$ μH, the magnitude and phase of its Bode plot are drawn in as solid curves. Assume that the variation imposed on the resistor and the capacitor is 20%, i.e., $C \in [0.8, 1.2]$ μF and $L \in [0.8, 1.2]$ μH. Then the resulted frequency response performance bounds are deviated from the nominal counterparts, and are plotted as dash curves in Fig. 3.2. In the same figure, we show three snapshots of the transfer function at different frequency points. In each snapshot, both the value variations of capacitor and inductor affect the magnitude $H^0(\omega)$.

The second example is the simplified CMOS device model as shown in the left part of Fig. 3.3, where the singular network elements like nullator and norator are used to model the ideal voltage controlled current sources. Suppose we apply a Norton current source, i.e, an ideal current source i_{s} with a parallel resistor g_{cur}, onto the gate node G of the MOS model, and observe the circuit output, shown in the right part of Fig. 3.3. The exact symbolic transfer function with current source as input and observed voltage on drain

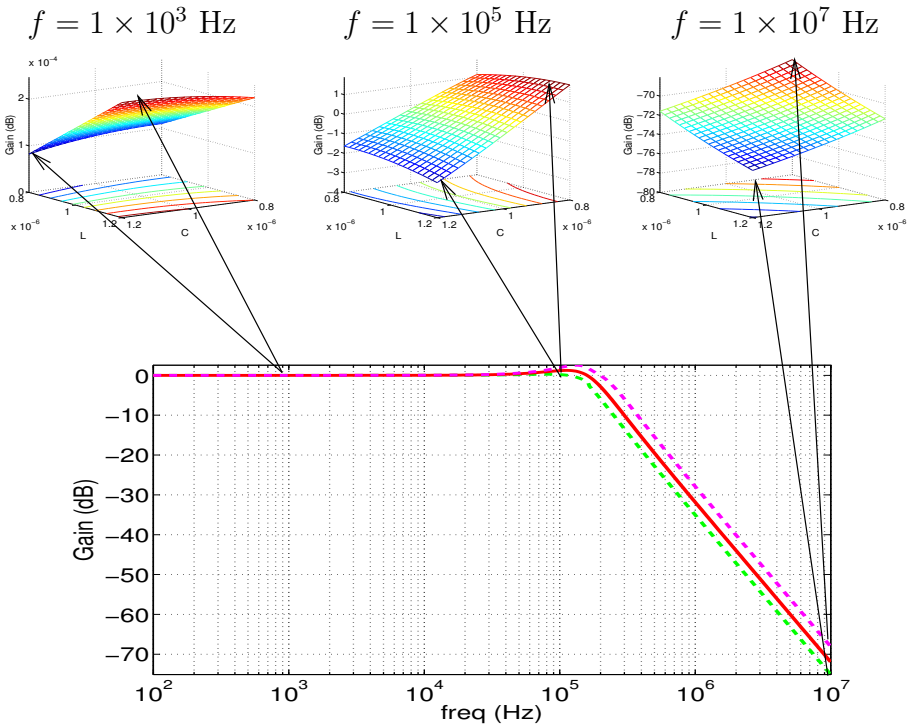


Figure 3.2: Frequency response of the RLC circuit. Solid curve is the magnitude response with nominal parameters, while the two dashed curves are lower and upper bounds due to process variation. The three surfaces at top, with L and C as x-axis and y-axis accordingly, and magnitude as z-axis, illustrate the variations of magnitude at three sampling frequencies.

node D as output is

$$H(j\omega) = \frac{v_{\rm D}(j\omega)}{i_{\rm s}(j\omega)} = \frac{g_{\rm m} - j\omega C_{\rm gd}}{(j\omega)^2 C_{\rm gs} C_{\rm gd} + j\omega(C_{\rm gs} g_{\rm ds} + C_{\rm gd}(g_{\rm ds} + g_{\rm m} + g_{\rm cur})) + g_{\rm ds} g_{\rm cur}}. \quad (3.5)$$

Once we know the exact transfer function and variations of the parameters such as $g_{\rm m}$, $g_{\rm ds}$, $C_{\rm gd}$, $C_{\rm gs}$, one can find the bounds of $H(s)$ easily as in the previous example. The transfer function and its variation bounds are plotted in Fig. 3.4. In this example, we have two variation parameters $g_{\rm ds}$ and $g_{\rm m}$. The variation spaces for the two variables at three different frequencies are also shown on the top of the same figure, which show the searching spaces at those frequency points for the two variables.

As we can see, to obtain the performance bounds of the analog circuit performances, the first step is to obtain the exact symbolic transfer functions like Eq. (3.4) in terms of all the variational circuit parameters. This will be done by the DDD-based exact symbolic analysis method as mentioned in Section 3.2. We remark that one can also use circuit simulator like SPICE to evaluate the performances for a given set of parameter values and frequency points. But the DDD method is relevant here because it can give closed form expressions for a given circuit performances, which can lead to much faster evaluations compared to numerical methods [19].

Secondly, once we obtain the exact symbolic transfer functions, we need to find a systematic way to obtain the performance bounds given the bounds of variation parameters. In this work, we formulate the bound computing problem into a nonlinear constrained optimization problem. To obtain the two performance bounds for magnitude or phase at one frequency point, two evaluation processes, or optimization runs, of the transfer function are needed: one for $H^0(\omega)$, and the other for $\theta(\omega)$. The range of frequency sweep and number

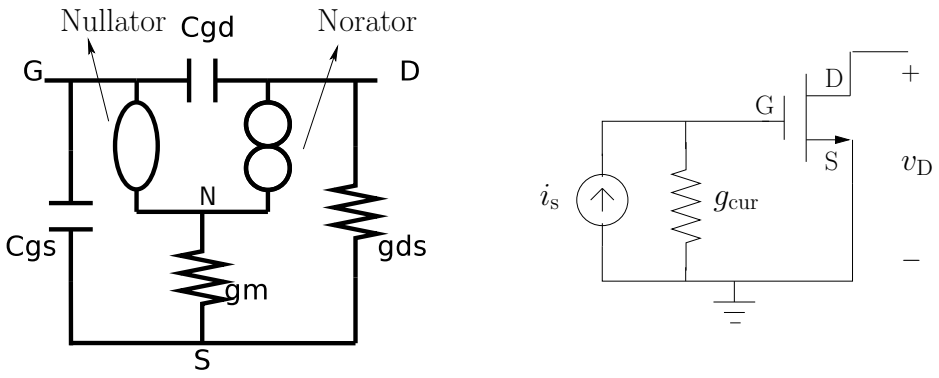


Figure 3.3: The small-signal model for MOS transistors.

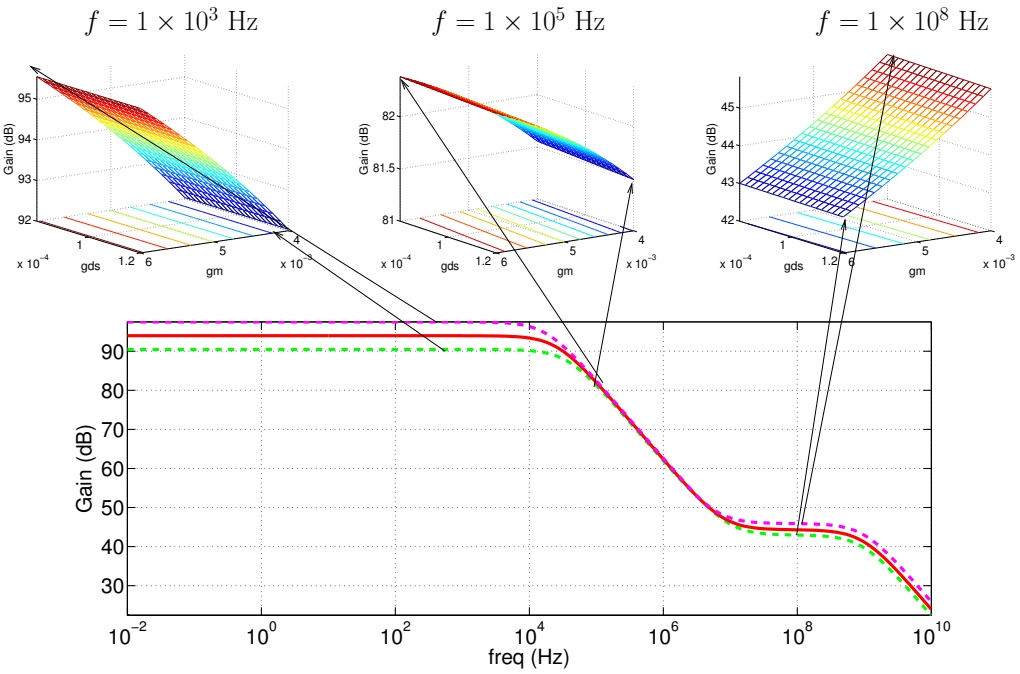


Figure 3.4: Frequency response of the simplified MOS model driven by Norton current source. Solid curve is the magnitude response with nominal parameters, while the two dashed curves are lower and upper bounds due to process variation. The three surfaces at top, with $g_{\rm ds}$ and $g_{\rm m}$ as x-axis and y-axis accordingly, and magnitude as z-axis, illustrate the variations of magnitude at three sampling frequencies.

of frequency points are determined freely by the designer. We use the lower bound of the magnitude response $|H(j\omega)|$ at frequency ω for an example. The magnitude of the transfer function, which can be evaluated from the available symbolic transfer function, is used as the nonlinear objective function to be minimized:

$$\begin{aligned} &\text{minimize} && |(H(j\omega, \mathbf{x})| \\ &\text{subject to} && \mathbf{x}_{\text{lower}} \leq \mathbf{x} \leq \mathbf{x}_{\text{upper}}, \end{aligned} \tag{3.6}$$

where $\mathbf{x} = [p_1, \ldots, p_m]$ represents the circuit parameter variable vector, which is subjected to the optimization constraints $[\mathbf{x}_{\text{lower}}, \mathbf{x}_{\text{upper}}]$. In circuit design, these constraints are supplied by foundries and cell library vendors. Hence, after (3.6) is solved by an optimization engine, the lower bound of the magnitude response at ω, i.e., $|H_{\text{l}}(j\omega)|$, is returned and a parameter set at which the minimum is attained will also be saved as a by-product. The iterative search procedure in the constrained minimization is illustrated in Fig. 3.5.

The nonlinear optimization problem with simple upper and lower bounds given in (3.6) can be efficiently solved by several methods such as active-set, interior point, and trust region algorithms [36–38]. All those methods are iterative approaches starting with an initial feasible solution. In this work, we use the active-set method [38], as it turns to be the most robust nonlinear optimization method for our application. Active-set methods are two-phase iterative methods that provide an estimate of the active set (active set is the set of constraints that are satisfied with equality) at the solution. In the first phase, the objective is ignored while a feasible point is found for the constraints. In the second phase, the objective is minimized while feasibility is maintained. In the second phase, starting from a feasible initial point $\mathbf{x}_0$, the method computes a sequence of feasible iterates $\{\mathbf{x}_k\}$ such that $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$ and $H(\mathbf{x}_{k+1}) \leq H(\mathbf{x}_k)$ via methods like quadratic programming,

where $\mathbf{p}_k$ is a nonzero search direction and α_k is a non-negative step length.

We remark that the active-set method is still a local optimization method, which finds the local optimal solutions. It will be desirable to find the global optimal solution, which can give true bounds of performance. But this goal may come with more or much higher computing costs by performing many tries. The effort boils down to a trade-off between accuracy and costs in this problem. In our approach, we still perform one optimization. Our experimental results show that by using reasonable initial guesses as mentioned before, the proposed method gives very close bounds compared with Monte Carlo based methods for the examples used.

Algorithm 3 Calculation of frequency response bounds via symbolic analysis and constrained optimization.

1: Read circuit netlist.
2: Set bounds on process variation affected parameters.
3: Generate symbolic expression of transfer functions.
4: **for** each ω_i **do**
5: Run nonlinear constrained optimization (3.6) which uses transfer function as objective to find magnitude and phase bounds on ω_i.
6: **end for**
7: Save bound information for future statistical and yield analysis.

To further speedup the optimization, the initial point selection can be further improved. Since the responses at two neighboring frequency points are usually close to each other, the starting point $\mathbf{x}$ for frequency point ω_{i+1} can be set using the solution at the previous frequency point ω_i. Therefore, the initial guess points do not always have to be

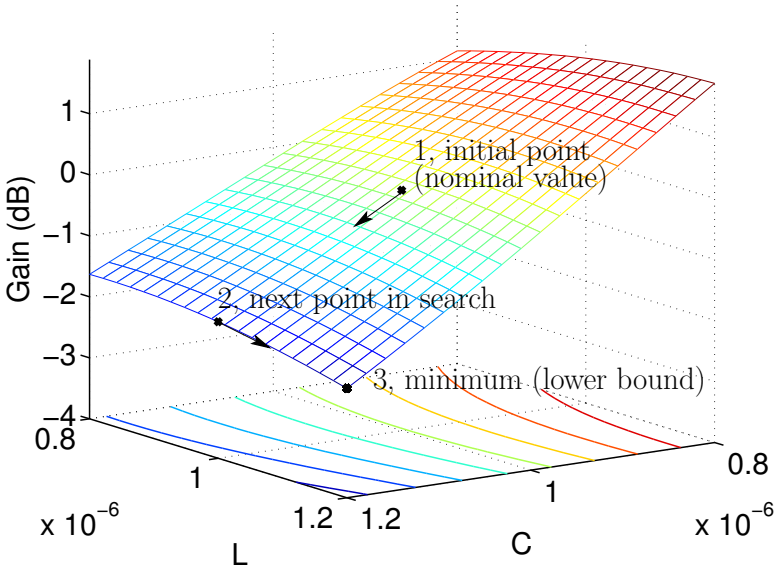


Figure 3.5: Optimization space searching for the RLC circuit in Fig. 3.1, where both C and L have 20% variation in this illustration. This surface shows the magnitude variations at frequency $f = 10^6$ Hz.

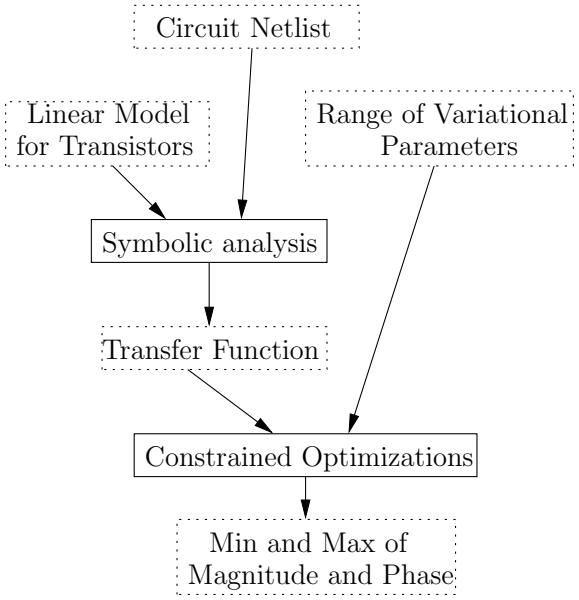


Figure 3.6: The flowchart of frequency domain performance bound calculation.

Point 1, i.e., the nominal value, in Fig. 3.5, and the previous frequency's optimal point is heuristically the best shortcut of initial guess at current frequency. This strategy tends to reduce the time required by the optimization to search its minimal or maximal point in the whole variable space, and thus speedup the calculation time of the bound analysis. Fig. 3.6 summarizes the flow of the performance bound calculation.

3.4 Time domain bound analysis

In the previous section, we have shown our frequency performance bound method using symbolic analysis and constrained optimization. Based on the calculated frequency performance bounds, we next develop our time domain bound analysis, or TIDBA, which converts the frequency domain bounds to time-domain bounds for general input signals. Our approach was inspired by [35], which determines time-domain performance bounds of an uncertain system for impulse or step input signals. However, this method does not give transient performance bounds in response to general input signals, which are required by our analog circuit analysis. In the sequel, we first briefly review the work in [35] before introducing our new approach. Note that the bounds of magnitude and phase of the transfer function required by TIDBA can be generated by any existing frequency bound analysis methods and not limited to the one we proposed in the previous section.

We first present the whole TIDBA algorithm flow, which is shown in Algorithm 4. Then we explain each step in detail in the following sections. As can be seen from the flow, the time domain response analysis requires the results such as transfer function bounds from the procedures we studied in previous sections. After the first two steps, the bounds

of magnitude and phase (angle) shown in the inequalities (3.2) and (3.3) are available. Then TIDBA converts frequency-domain performance bounds into the time-domain performance bounds by impulse signal based time-domain bound analysis and FFT/IFFT, which will be the focus of this section.

Algorithm 4 The algorithm flow of the new time-domain performance bound analysis method — TIDBA.

Input: circuit netlist with variable parameters; stimulus signal of the circuit.

Output: lower and upper bounds of the output signal in time-domain.

1: Compute the variational transfer function by graph-based symbolic method and affine arithmetic method.

2: Compute the performance bounds of the variational transfer function by constrained-nonlinear optimization method.

3: Compute the time-domain performance bounds by a new general-signal transient bound analysis method.

3.4.1 Review of transient bound analysis driven by impulse signals

For the completeness of our presentation, we first present the transient bound analysis with impulse signals. For a purely real signal $x(t)$ in time domain, its Fourier transform $X(j\omega) = X^0(\omega) \cdot e^{j\phi(\omega)}$ in frequency domain holds the property of conjugate symmetry, i.e.,

$$X(-j\omega) = X(j\omega)^*. \tag{3.7}$$

It can be equivalently expressed by the even property of magnitude and the odd property of phase: $X(-\omega) = X^0(\omega)$, and $\phi(-\omega) = -\phi(\omega)$. It is not difficult to show that the transfer

function of a physically realizable system also holds the conjugate symmetry property [39].

Since the spectrum of an impulse signal $\delta(t)$ is $X(j\omega) = 1$ everywhere, the spectrum of the system's output signal is $Y(j\omega) = X(j\omega)H(j\omega) = H(j\omega)$, and hence the impulse response of the system in time domain is simply the inverse Fourier transform of $H(j\omega)$,

$$\begin{aligned} y(t) &= \frac{1}{2\pi}\int_{-\infty}^{\infty} H(j\omega)e^{j\omega t}\,\mathrm{d}\omega \\ &= \frac{1}{2\pi}\int_{-\infty}^{\infty} H^0(\omega)e^{j(\omega t+\theta(\omega))}\,\mathrm{d}\omega, \quad t>0. \end{aligned} \tag{3.8}$$

Employing the even and odd properties of $H(j\omega)$, (3.8) can be equivalently integrated from $\omega = 0$ to ∞,

$$\begin{aligned} y(t) &= \frac{1}{\pi}\int_{0}^{\infty} H^0(\omega)\mathrm{Re}(e^{j(\omega t+\theta(\omega))})\,\mathrm{d}\omega \\ &= \frac{1}{\pi}\int_{0}^{\infty} H^0(\omega)\cos(\omega t+\theta(\omega))\,\mathrm{d}\omega, \quad t>0. \end{aligned} \tag{3.9}$$

A modification of this integral to discrete sum on sampled frequency points allows one to calculate the approximate result of $y(t)$ at each time point as

$$y(t) = \frac{1}{\pi}\sum_{n=0}^{N-1} \underbrace{H^0(\omega_n)\cos(\omega_n t+\theta(\omega_n))}_{I(\omega_n)}\Delta\omega_n, \quad t>0. \tag{3.10}$$

In the presence of process variation, the transfer function will be given in the bounded form in (3.2) and (3.3). Therefore, to compute the lower and upper transient bounds $y_{\mathrm{l}}(t)$ and $y_{\mathrm{u}}(t)$ for each time point t, the integrand body $I(\omega_n)$ in (3.10) is calculated using the following rules.

First, find the minimum and maximum values of $\cos(\omega_n t+\theta(\omega_n))$, where the phase angle $\theta(\omega_n)$ can vary in the interval $[\theta_{\mathrm{l}}(\omega_n), \theta_{\mathrm{u}}(\omega_n)]$. Let $C_{\min}(\omega_n)$ and $C_{\max}(\omega_n)$ denote the

two extreme values of the cosine function. Then, for $y_{\rm l}(t)$, all $I(\omega_n)$ shall be calculated as

$$I(\omega_n) = \begin{cases} H_{\rm u}^0(\omega_n) C_{\min}(\omega_n), & C_{\min}(\omega_n) \leq 0 \\ H_{\rm l}^0(\omega_n) C_{\min}(\omega_n), & C_{\min}(\omega_n) > 0, \end{cases} \tag{3.11}$$

and, for $y_{\rm u}(t)$, the situation is simply reversed,

$$I(\omega_n) = \begin{cases} H_{\rm l}^0(\omega_n) C_{\max}(\omega_n), & C_{\max}(\omega_n) \leq 0 \\ H_{\rm u}^0(\omega_n) C_{\max}(\omega_n), & C_{\max}(\omega_n) > 0. \end{cases} \tag{3.12}$$

3.4.2 Proposed transient bound analysis with general input signal

For a general time domain signal $x(t)$ in circuit analysis application, its frequency-domain transform $X(j\omega)$ can be calculated by fast Fourier transform, FFT. This requires sampling points of the signal on a set of discretized time points. For example, with a uniform sampling period $T_s = 1/F_s$, $x(t)$ is sampled and stored as $x(0)$, $x(T_s)$, $x(2T_s)$,..., $x(NT_s)$. For the sake of simplicity, we will omit the term T_s and denote the time point indices by subscripts in the following descriptions. Thus the notation x_n will stand for the sampled value of signal $x(t)$ at time $t = nT_s$.

To achieve accurate results from FFT and IFFT, Nyquist sampling theorem requires the sampling frequency $F_s = 1/T_s$ to be at least twice of the bandwidth of signal [40]. Meanwhile, the total sampling duration $T_0 = T_s N$ determines the resolution of the FFT spectrum, i.e., the sampling interval of frequency domain is $F_0 = 1/T_0$. The longer T_0 is, the higher resolution we can get, and thus the more sampling points are needed.

Given N sampling points, the FFT transform pair is

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N}nk}, \quad k = 0, 1, \ldots, N-1, \text{ and} \tag{3.13}$$

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{j\frac{2\pi}{N}nk}, \quad n = 0, 1, \ldots, N-1. \tag{3.14}$$

In circuit analysis applications, because the input data x_n are purely real, the symmetry property mentioned previously still holds, though in a different form, $X_{N-k} = X_k^*$. This means that the right half in X_k is a conjugate swap of its left half without X_0, which is the zero-frequency component of the spectrum. The points in the left half, i.e., X_k for $k = 0, \ldots, N/2$, are the spectral points of frequencies $f = kF_0$. Fig. 3.7 illustrates the FFT series and its conjugate symmetry.

Based on this property of a real signal's spectrum, the inverse discrete Fourier transform can be calculated with the spectrum's left half. Consequently, the equivalent form of (3.14) becomes

$$x_n = \frac{1}{N}\left[X_0 + 2\sum_{k=1}^{N/2} \mathrm{Re}(X_k e^{j\frac{2\pi}{N}nk})\right], \quad n = 0, 1, \ldots, N-1. \tag{3.15}$$

A remark is made that throughout this work the expression using only left half is mainly for the convenience of mathematical analysis. In practice, full spectrum is constructed before the final converting to time domain, and thus IFFT is ready to be applied for its efficiency.

Now it is the time to derive the time response bounds from the FFT series of signal $x(t)$ given the frequency bounds of the system $H(j\omega)$. First we consider the system without variation. After the FFT of x_n as represented in (3.13), its spectrum $X_k = |X_k|e^{j\phi_k}$ is multiplied with $H_k = H(j\omega_k)$, $\omega_k = 2\pi k F_0$, to obtain the spectrum of output signal. Then,

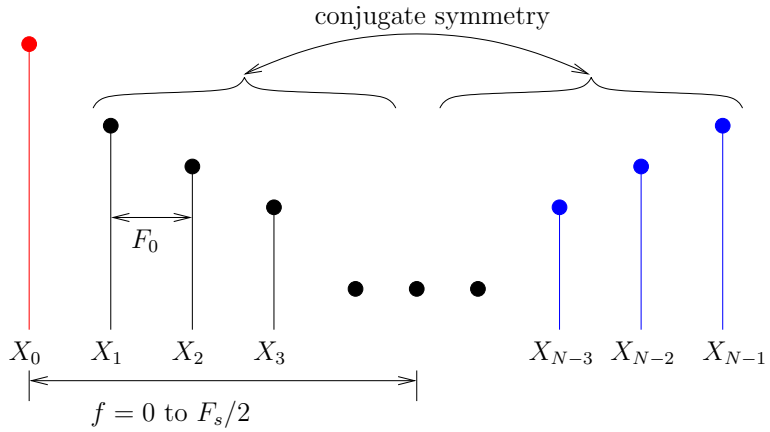


Figure 3.7: Conjugate symmetry between left half and right half of the FFT series X_k, $k = 0, \ldots, N-1$.

we make a domain translation from frequency domain to time domain, which is similar to (3.15). In this way, the output signal y_n is obtained for the nominal designed system.

$$\begin{aligned} y_n &= \frac{1}{N}\left[Y_0 + 2\sum_{k=1}^{N/2} \mathrm{Re}\left[Y_k e^{j\frac{2\pi}{N}nk}\right]\right] \\ &= \frac{1}{N}\left[X_0 H^0(0) + 2\sum_{k=1}^{N/2} \mathrm{Re}\left[H^0(\omega_k) e^{j\theta(\omega_k)} X_k e^{j\frac{2\pi}{N}nk}\right]\right] \\ &= \frac{1}{N}\left[X_0 H^0(0) + 2\sum_{k=1}^{N/2} |X_k| H^0(\omega_k) \mathrm{Re}\left[e^{j(\phi_k+\theta(\omega_k)+\frac{2\pi}{N}nk)}\right]\right] \end{aligned} \tag{3.16}$$

Now we consider the process variations. In this case, the minimum and maximum values, similar to (3.11) and (3.12) for the impulse signals, have to be derived from Eq. (3.16) in the bounded region of the system transfer function at every frequency point. Specifically, the selection and combinations of $H^0(\omega)$ and $\theta(\omega)$ will depend on the sign of the real part of the output spectrum, i.e., $\mathrm{Re}\{e^{j(\phi_k+\theta(\omega_k)+\frac{2\pi}{N}nk)}\}$. Detailed analysis shows that there are many combinations of extreme values of $H^0(\omega)$ and $\theta(\omega)$ depending on the locations of $\phi_k + \theta(\omega_k) + \frac{2\pi}{N}nk$ in the complex plane, which are summarized in Table 3.1. Let's walk through one example illustrated in Fig. 3.8, where all possible values of $\theta(\omega_k)$ make the

phase $\phi_k + \theta(\omega_k) + \frac{2\pi}{N}nk$ fall in the first quadrant, and thus their real parts are all positive. Therefore, the selection of $H_{\mathrm{l}}^0(\omega_k)$ and $\theta_{\mathrm{u}}(\omega_k)$ will lead to the minimum of output value, while $H_{\mathrm{u}}^0(\omega_k)$ and $\theta_{\mathrm{l}}(\omega_k)$ lead to the maximum one. In Fig. 3.8, these two combinations are marked by black dots.

We remark that the range of allowed phase values $[\theta_{\mathrm{l}}(\omega_k), \theta_{\mathrm{u}}(\omega_k)]$ affects the rules for bound determination, as shown in Table 3.1. In this work, the maximum phase range is restricted to be less than 90 degrees, i.e., $\theta_{\mathrm{u}}(\omega_k) - \theta_{\mathrm{l}}(\omega_k) < \pi/2$ rads. There are two reasons for this restriction: 1) The restriction of 90 degrees accommodates most circuit transfer function's variation very well. 2) If much larger phase variation is detected at the frequency domain, the variation is very likely to cause faults in the circuit. We stress that there is no difficulty to generate new bound determination rules to handle phase range larger than 90 degrees.

With this assumption, the rules for time domain bound determination are summarized in Table 3.1. For brevity, let $\Theta_{\mathrm{l}}(\omega_k) = \phi_k + \theta_{\mathrm{l}}(\omega_k) + \frac{2\pi}{N}nk$, and $\Theta_{\mathrm{u}}(\omega_k) = \phi_k + \theta_{\mathrm{u}}(\omega_k) + \frac{2\pi}{N}nk$. If the range of Θ is not covered by the enumerated regions, a phase shift of 2π can be applied to relocate its value within the listed ranges. In addition, the "either $\Theta_{\mathrm{l}}(\omega_k)$ or $\Theta_{\mathrm{u}}(\omega_k)$" in the first row and the fifth row in the table means one of them will be selected: in the first row, the lower bound will happen at one of them which makes $\cos(\Theta)$ smaller; and in the fifth row, the upper bound will take place at the phase angle making $\cos(\Theta)$ larger.

Fig. 3.9 shows the implementation flow of the proposed general-signal transient bound determination method. It starts from a time domain sampling of input signal $x(t)$ and

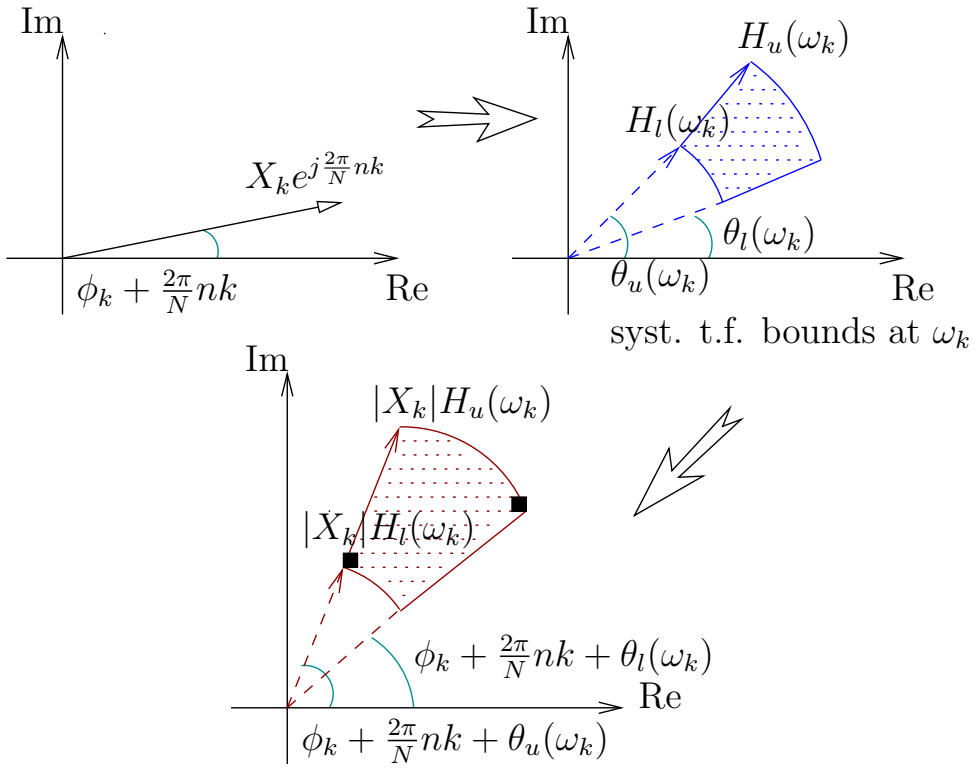


Figure 3.8: The magnification and rotation of input spectrum by the transfer function bounds.

Table 3.1: Rules for time domain bound determination.

range of phase		quad-	sign of	magnitude and phase combinations for			
$\Theta_{\rm l}(\omega_k)$	$\Theta_{\rm u}(\omega_k)$	rants	$\mathrm{Re}[e^{j\Theta}]$	lower bound		upper bound	
$(-\pi/2, 0)$	$(0, \pi/2)$	IV, I	+	$H^0_{\rm l}(\omega_k)$	either $\Theta_{\rm l}(\omega_k)$ or $\Theta_{\rm u}(\omega_k)$	$H^0_{\rm u}(\omega_k)$	$\Theta(\omega_k) = 0$
$(0, \pi/2)$	$(0, \pi/2)$	I	+	$H^0_{\rm l}(\omega_k)$	$\Theta_{\rm u}(\omega_k)$	$H^0_{\rm u}(\omega_k)$	$\Theta_{\rm l}(\omega_k)$
$(0, \pi/2)$	$(\pi/2, \pi)$	I, II	+, −	$H^0_{\rm u}(\omega_k)$	$\Theta_{\rm u}(\omega_k)$	$H^0_{\rm u}(\omega_k)$	$\Theta_{\rm l}(\omega_k)$
$(\pi/2, \pi)$	$(\pi/2, \pi)$	II	−	$H^0_{\rm u}(\omega_k)$	$\Theta_{\rm u}(\omega_k)$	$H^0_{\rm l}(\omega_k)$	$\Theta_{\rm l}(\omega_k)$
$(\pi/2, \pi)$	$(\pi, 3\pi/2)$	II, III	−	$H^0_{\rm u}(\omega_k)$	$\Theta(\omega_k) = \pi$	$H^0_{\rm l}(\omega_k)$	either $\Theta_{\rm l}(\omega_k)$ or $\Theta_{\rm u}(\omega_k)$
$(\pi, 3\pi/2)$	$(\pi, 3\pi/2)$	III	−	$H^0_{\rm u}(\omega_k)$	$\Theta_{\rm l}(\omega_k)$	$H^0_{\rm l}(\omega_k)$	$\Theta_{\rm u}(\omega_k)$
$(\pi, 3\pi/2)$	$(3\pi/2, 2\pi)$	III, IV	+, −	$H^0_{\rm u}(\omega_k)$	$\Theta_{\rm l}(\omega_k)$	$H^0_{\rm u}(\omega_k)$	$\Theta_{\rm u}(\omega_k)$
$(3\pi/2, 2\pi)$	$(3\pi/2, 2\pi)$	IV	+	$H^0_{\rm l}(\omega_k)$	$\Theta_{\rm l}(\omega_k)$	$H^0_{\rm u}(\omega_k)$	$\Theta_{\rm u}(\omega_k)$

given system transfer function bounds in frequency domain. The FFT operation transforms the input signal to its spectrum and then the proposed rules in Table 3.1 are applied to determine the magnitude and phase combinations for lower and upper time domain bounds at every frequency point in the left half of the spectrum. This process is marked by the dashed line box, labeled "1" in Fig. 3.9. Next, frequency domain results, i.e., $Y_0, Y_1, \ldots, Y_{N/2}$, either lower ones or upper ones, are used to construct a full N-length series based on conjugate symmetry property. Last, IFFT is used to calculate the final result of time domain bounds. This procedure is also marked by dashed line box, labeled "2" in the figure.

3.5 Numerical results

In this section, we show experimental results of the proposed method on some benchmark analog circuit netlists. Both frequency domain bounds and time domain bounds are calculated by our new method. As an application, analog yield analysis is also performed for two circuits based on the bound analysis results. This section is divided into two subsections: the first one shows the frequency domain response bound results, while the second one demonstrates those results of time domain response bounds.

For running time comparisons, we also measure the time cost by the commercial HSPICE, which runs all the Monte Carlo simulations. All running times are obtained from a Linux server with a 2.4 GHz Intel Xeon Quad-Core CPU, and 36 GBytes memory.

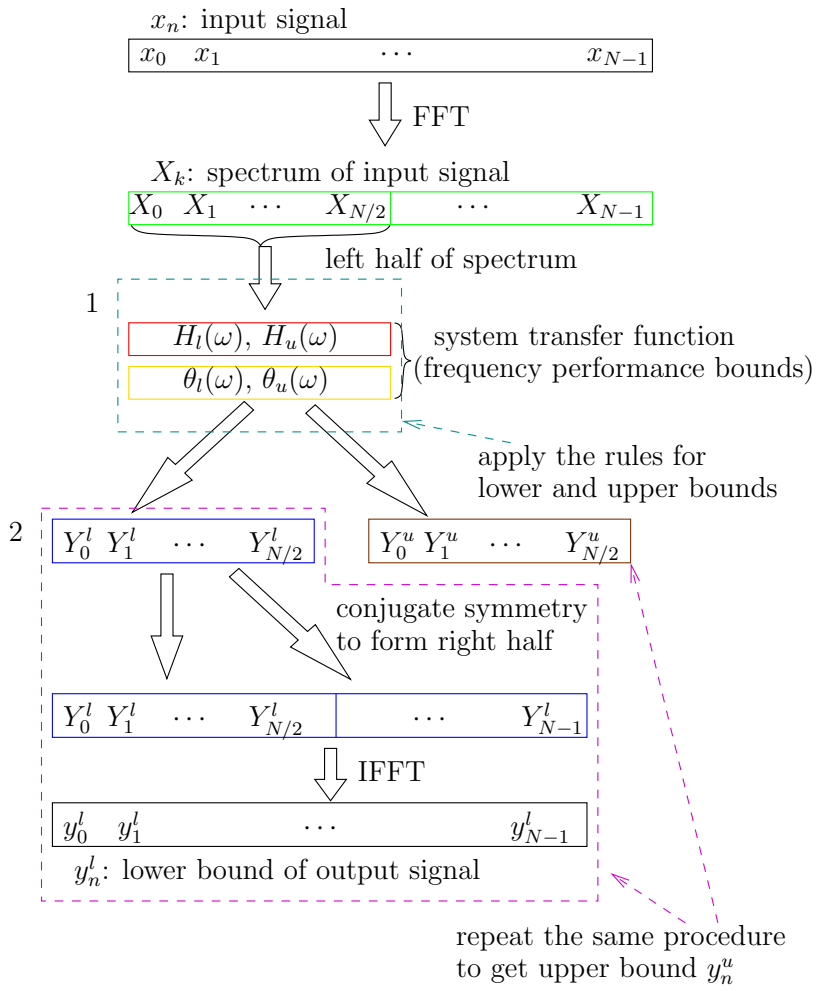


Figure 3.9: The proposed general-signal transient bound determination method

3.5.1 Frequency domain response bounds

The symbolic expressions of transfer functions are generated by the DDD symbolic analysis tool [19], and the optimization based bound calculation are done in MATLAB. The nonlinear constrained optimizations are solved by the `fmincon` function in MATLAB's Optimization Toolbox [41]. We explicitly select the active-set algorithm for `fmincon` for the optimizations (we also tried other methods and found out that the active-set method is the most robust and reliable method).

We first investigate the accuracy and efficiency of our method with typical circuit examples. Fig. 3.10 shows the schematic of a CMOS operational amplifier circuit, which contains 9 transistors. Its differential inputs are provided at the gate terminals of the differential pair (M1 and M2), while the output is observed at the output node of the source follower stage. DC analysis is first performed by HSPICE to obtain the operating point, and then small-signal models of nonlinear devices, such as MOS transistors, are used for DDD symbolic analysis and transfer function evaluation. For example, the original NMOS device is replaced by the equivalent circuit model consisting of voltage controlled current source (VCCS), gate-source capacitance (C_{GS}), gate-drain capacitance (C_{GD}), terminal resistance, and so on, as shown in Fig. 3.3. In this MOS small signal model, we use singular network elements like nullator and norator. The combination of these elements in the MOS model behaves as an ideal VCCS. However, the properties of the nullator (who does not allow current flowing through it and provides zero voltage difference between its two terminals, i.e., the voltage values on nodes G and N are same) and the norator (who allows any voltage across its two terminals and any current flowing through it) allow us to formulate

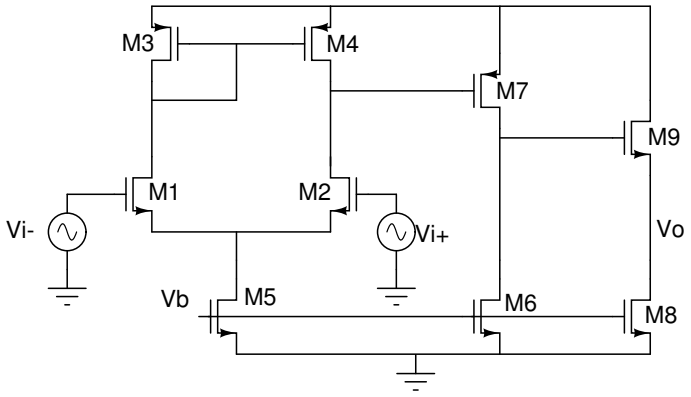


Figure 3.10: The circuit schematic of the CMOS operational amplifier.

more compact equations than MNA [42].

Table 3.2 enumerates the experiment parameters and running time comparisons. In the op-amp netlist, six variational parameters are introduced, including resistors, capacitors, and controlled sources inside the transistor model as local variations, and other four parameters are modeled as global variations. The active filter example is modeled in a similar way.

After the symbolic expressions of the op-amp's transfer function are obtained, the nominal frequency response can be evaluated straightforwardly using the specified parameter values. The lower and upper bounds of the magnitude and phase are then obtained by the aforementioned constrained optimization. Fig. 3.11 plots the nominal magnitude curve along with its lower and upper bounds. On the same figure, we also plot the Monte Carlo

Table 3.2: Process variation setup in the benchmarks.

Circuit name	# Var. param	Variation	
		local	global
CMOS op-amp	10	5%	10%
Active filter	7	5%	10%

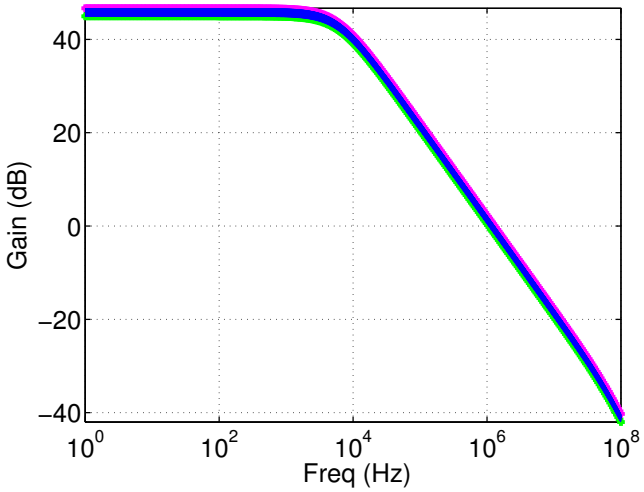
samples of the same circuits. It is obvious that our bounds include all possible variations, and do not show much over-conservativeness. The result demonstrates the effectiveness of the optimization-based method to find accurate bounds.

As an application of the proposed method, we apply the proposed method for analog yield estimation. We illustrate this using the same op-amp. The yield estimation is calculated using preset specification. One important specification of op-amp is open-loop gain at a relatively low frequency. For the CMOS op-amp in Fig. 3.10, we set a requirement that the accepted circuits should have its gain larger than 45 dB at frequency $f = 1$ kHz. HSPICE Monte Carlo analysis gives the yield as 98.8%, and the histogram of all samples is drawn in Fig. 3.12. Meanwhile, the predicted yield using the proposed method is 98.4%, which is fairly close to that of the MC analysis. The detailed statistics of the comparison are shown in Table 3.3. With the accurate calculation of performance bounds and the yield, the presented method only takes 3.8 seconds. This is a 22× speedup over the Monte Carlo method.

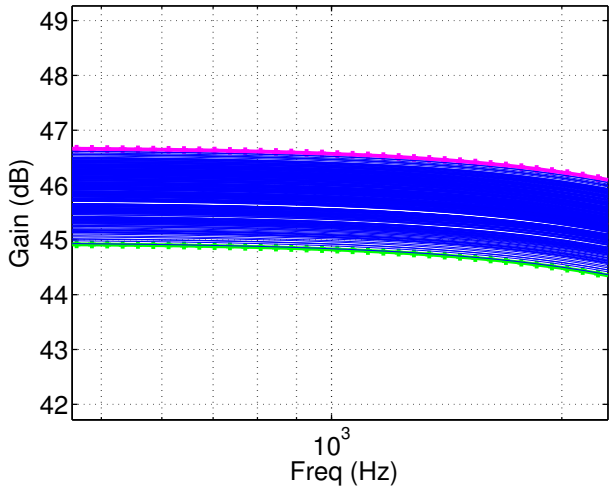
The proposed algorithm is also applied to a CMOS active filter [43] (circuit diagram not shown here). Fig. 3.13 shows the magnitude bounds together with HSPICE Monte Carlo results. The statistical data is listed in Table 3.4. A speedup of 13× is observed on this example.

3.5.2 Time domain response bounds

Using the frequency domain bounds we calculated in the previous experiments, the time domain bounds of the CMOS op-amp are obtained by the TIDBA method. Fig. 3.14 shows a spectrum of Monte Carlo pulse responses at the output node of the op-amp, and



(a) Monte Carlo simulations and magnitude bounds of op-amp.



(b) Detailed view around 10^3 Hz.

Figure 3.11: Monte Carlo simulations and magnitude bounds of op-amp circuit. The thick dashed lines are lower and upper bounds, and the thin solid lines are Monte Carlo results.

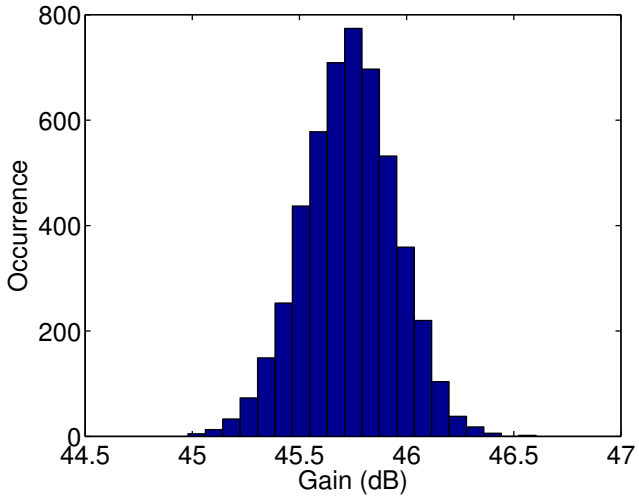


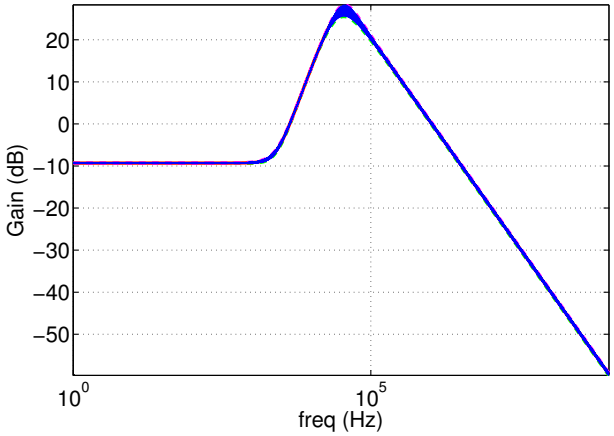
Figure 3.12: The histogram of gain distribution of the CMOS op-amp at frequency f = 1 kHz. (5000 times Monte Carlo simulation.)

Table 3.3: Statistical information of the CMOS op-amp circuit. (Comparison with 5000 times Monte Carlo.)

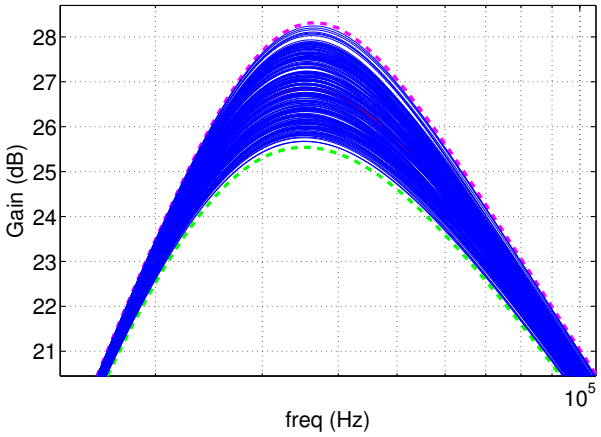
CMOS op-amp		
Runtime (seconds)	MC	85.2
	proposed	3.8
Mean value (μ)	MC	45.8
Unit: dB	proposed	45.8
Std. value (σ)	MC	0.214
Unit: dB	proposed	0.214
Yield rate	MC	98.8%
	proposed	98.4%

Table 3.4: Statistical information of the CMOS filter. (Comparison with 5000 times Monte Carlo.)

CMOS Filter		
Runtime (seconds)	MC	100.4
	proposed	8.2
Mean value (μ)	MC	26.83
Unit: dB	proposed	26.81
Std. value (σ)	MC	0.389
Unit: dB	proposed	0.384
Yield rate	MC	82.7%
	proposed	84.2%



(a) Monte Carlo simulations and magnitude bounds of active filter.



(b) Detailed view around 10^3 Hz.

Figure 3.13: Monte Carlo simulations and magnitude bounds of active filter. The thick dashed lines are lower and upper bounds, and the thin solid lines are Monte Carlo results.

the bounds generated from TIDBA are overlaid onto the same figure as thicker curves.

We simulated the active filter with a pulse waveform as input, and the resulting output waveforms are plotted as dot-dashed curves in Fig. 3.15. Due to the process variation of the filter, it can be observed that the output waveforms are deviated from its nominal benchmark. Detailed plots of the up ramp and down ramp are shown in Fig. 3.15(b) and Fig. 3.15(c). The time domain performance bounds, computed by TIDBA, are plotted as solid curves. An input signal composed of several sinusoidal waves is also used to test this filter. The circuit's possible minimum and maximum values in time domain and the TIDBA bounds are plotted in Fig. 3.16.

We notice that the bounds given by TIDBA may not be able to converge to the steady state of the response, for example, after 0.06 seconds in Fig. 3.15(a), which should be zero. This is due to the loss of dependence between magnitude and phase when we apply the frequency response bounds (3.2) and (3.3). However, for many steady states, which are known to be zero, even with variations in parameters, we can ignore the bounds given by the proposed method. Another way to mitigate this problem is to directly compute time-domain bounds using the optimization based approaches, which will be investigated in our future works.

Table 3.5 summarizes the experiment parameters and running time comparisons. In the op-amp netlist, six variation parameters are introduced, including resistors, capacitors, and controlled sources inside the transistor model as local variations, and four other parameters are modeled as global variations. The active filter example is modeled in a similar way. We make these variation parameters Gaussian random and run a 10,000 times

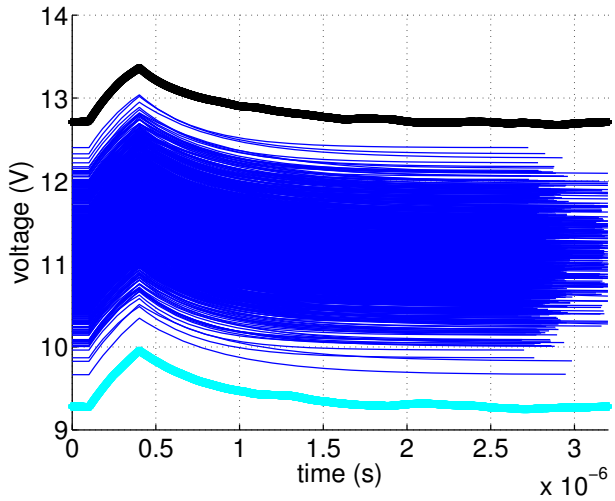


Figure 3.14: Time domain response of CMOS op-amp with pulse input. The two thicker curves are the lower and upper bounds from TIDBA, while the thinner lines are output waveforms from Monte Carlo simulation affected by variations.

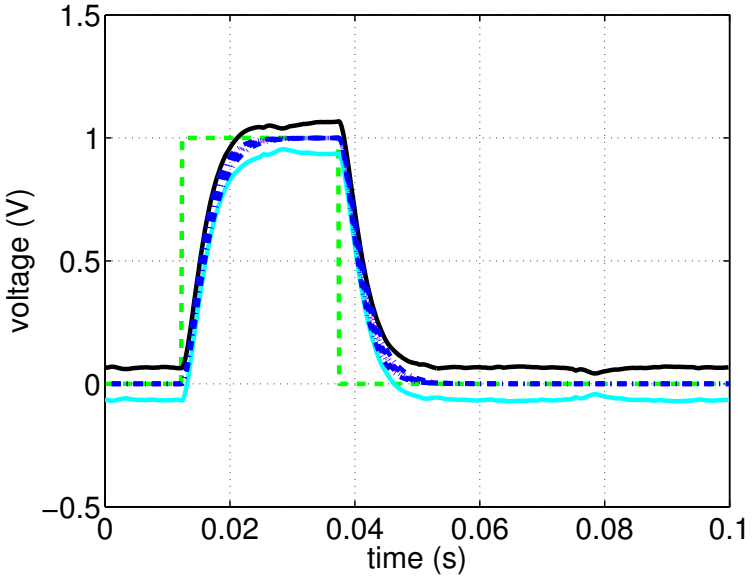
Monte Carlo simulations in HSPICE. A total number of 6,400 time domain samplings of input stimulus are fed into FFT. The running time measurements of the Monte Carlo and our method are also listed in the table. The maximum speedup of TIDBA over Monte Carlo simulation can be 38×.

3.6 Summary

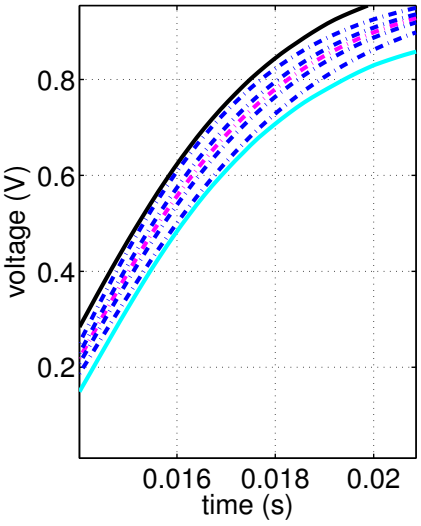
We have proposed a new performance bound analysis algorithm of analog circuits considering process variations in both time and frequency domains. The new method applies

Table 3.5: Performance comparison of TIDBA against the Monte Carlo method

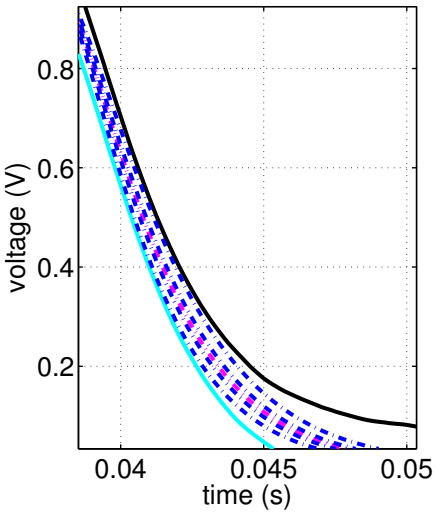
Circuit name	CPU time		speed up
	MC (10,000)	TIDBA	
Op-amp	362.9 s	11.2 s	32×
Filter	459.7 s	12.1 s	38×



(a) The whole plot



(b) Detail of up ramp



(c) Detail of down ramp

Figure 3.15: Time domain response of the active filter with pulse wave input. The two solid curves are the lower and upper bounds, while the region marked by dot-dashed lines are possible output waveforms from Monte Carlo simulation affected by variations.

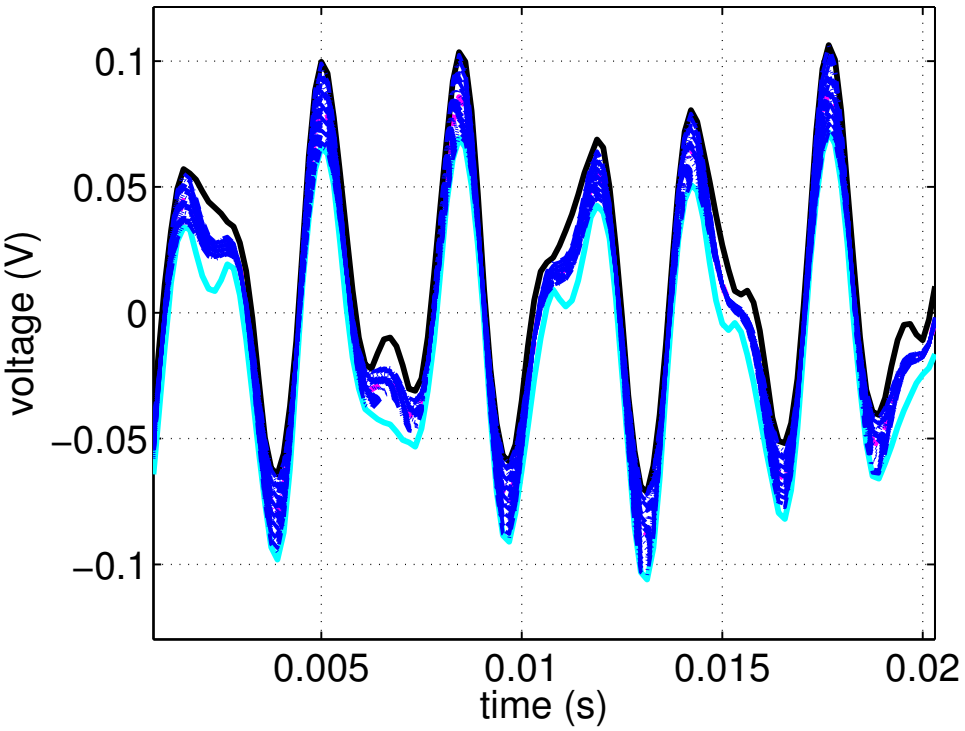


Figure 3.16: Time domain response of the active filter with sinusoidal wave input. The two solid curves are the lower and upper bounds, while the region marked by dot-dashed lines are possible output waveforms from Monte Carlo simulation affected by variations.

a graph-based analysis technique to derive the symbolic transfer functions of linearized analog circuits. Then the problem of finding frequency response bounds was formulated into a nonlinear constrained optimization problem, where the cost functions are magnitude and phase of the transfer function subject to the linear constraints, which are the upper and lower bounds of process variation parameters. The bounds calculated in this way are accurate and show no over-conservativeness suffered by the previous approaches. Based on the frequency response bounds, we further proposed an algorithm to compute time domain response bounds for any arbitrary input signals. Experimental results from several analog benchmark circuits show that the proposed method gives the correct bounds verified by Monte Carlo analysis while it delivers one order of magnitude speedup over Monte Carlo method in both frequency and time domain bound analysis. We also showed analog circuit yield analysis as an application of the bound analysis technique.

Currently the proposed method can only work on linear or linearized analog circuits. We are working on the performance bound analysis for general nonlinear circuits. For many circuits where time domain analysis are mainly performed, directly time-domain based bound analysis using optimization method will be more desirable and will be investigated in the future.

Chapter 4

MOR and GPU based power grid simulation

This chapter presents a new transient analysis method for general linear dynamic networks, such as on-chip power grid networks, using heterogeneous computing system with both GPU and multi-core CPU, which is the popular parallel computing solution nowadays. The new algorithm, called ETBR-GPU, first reduces the circuit complexity in frequency domain before the transient simulation is run on the reduced models. Such reduction based simulation technique is very amenable for parallelization on the hybrid GPU and multi-core CPU platforms, where coarse-grained task-level parallelism and fine-grained lightweight thread-level parallelism can be both exploited. ETBR-GPU first performs sampling-like reduction on the original circuit system where the frequency domain responses at different frequency samples can be calculated in parallel on multi-core CPU. After the reduction, the reduced circuit matrices, which are dense but well suitable for GPU's data parallel

computing, are simulated on GPU. The proposed method is very general, since it can analyze any linear networks with complicated structures and macro-models, and it does not assume any structure properties in order to build problem-specific preconditioners, as many iterative solvers do. Experimental results are given to show the efficiency of our tool.

4.1 Introduction

The verification of today's large linear global networks such as on-chip large power grid networks is very challenging for chip designers. Fast verification of voltage drops and other noises on power delivery networks is critical for final design closure. As the VLSI technology proceeds into the sub-65 nm scale, one challenging job of power grid network design is to predict and ensure a reliable on-chip power delivery. Since the power grid network usually comes with a huge size, its simulation and verification take a lot of time, and sometimes even make the analysis completely failed. This situation is further deteriorated as technology continues to scale down for several reasons. Firstly, the continued scale-down results in decreased interconnect width and thus higher interconnect resistance in a power supply network. Secondly, increased device density leads to increased supply current density on a chip. And thirdly, a higher clock frequency gives rise to more significant inductance effect. At the same time, supply voltage is lowered for power consideration, which results in a decreased noise margin for signal transition, and makes transistor more vulnerable to supply voltage degradation. Hence, efficient verification of power integrity becomes crucial for final design closure [44].

Intensive studies have been carried out to seek for efficient analysis of large power

grid networks in the past decade. Various algorithms have been proposed to improve scalability in computing time and to reduce memory footprints [45–50]. But most of those techniques are based on the homogeneous single-core architectures. Since the introduction of multi-core architectures, hardware designs are going through a renaissance. The leap from single-core to multi-core or many-core technologies has permanently altered the course of computing. Among them, GPU is one of the most powerful many-core computing systems arousing interests and input from both research and industry community. Several GPU-accelerated power grid analysis methods have been proposed recently [51–53]. A geometrical multi-grid algorithm based on hybrid GPU-CPU platforms was proposed in [51] to solve irregular power grid structures. In the work of [52], a 2D Poisson solver, which has closed form solutions computed by GPU-based FFT routine, was proposed for solving 2D structured power grids. But the algorithm may not be suitable for solving power grid networks with irregular structures. The work in [53] proposed to apply GPU-accelerated multi-grid method as the preconditioner for the Krylov subspace based iterative solver to handle more general power grid structures. However, the proposed iterative method is still very sensitive to the circuit structures as some assumptions about topologies are made (such as degree of a node). For power grid networks with heavy inductive effects and with other linear compact models, like high speed links and connectors for interface, the proposed method may not work well.

In our tool ETBR-GPU, we propose a new transient analysis method for general linear dynamic networks based on hybrid GPU and multi-core CPU platforms, which are the popular parallel computing system. We use on-chip power grid networks as the driving

problem to illustrate the effectiveness of the proposed method. ETBR-GPU, first reduces the circuit complexity in frequency domain before the transient simulation of the reduced circuits. Direct solvers, like sparse LU factorization, are used so that it can solve for any linear networks without dependency on problem-specific preconditioners as required by most of the iterative solvers. As a result, the parallelism and data independency in ETBR-GPU are exploited in a different way.

We show that ETBR-GPU has coarse-grained task-level and fine-grained lightweight thread-level parallelisms, each of them can be exploited by exploring the strengths of the GPU and CPU architectures. ETBR-GPU first performs sampling-like reduction on the original circuit matrices where the frequency domain responses at different frequency points have been calculated in parallel on multi-core CPU at the task level. After the reduction, the reduced circuit matrices, which are dense but well suitable for GPU platform computing, are simulated on GPU platforms at fine-grained thread level. ETBR-GPU especially favors transient simulation with long simulation periods as frequency-domain reduction related costs is almost independent of simulation time steps. Experimental results show that the new method can achieve about one or two orders of magnitude speedup when compared to the general LU-based simulation method on some recently published IBM power grid benchmark circuits [54].

This chapter is organized as follows. Section 4.2.1 and 4.2.2 describe the power grid simulation and the extended truncated balanced reduction techniques to reduce the original power grid system into a much smaller model. In Section 4.3, we introduce the flow of the proposed ETBR-GPU simulation of power grid networks with implementation

details. Experiment results are demonstrated in section 4.4. Section 4.5 summarizes the work.

4.2 Background

4.2.1 The problem of power grid simulation

An on-chip power grid network can be modeled as RC or RLC networks with known time-variant current sources, which can be obtained by gate-level logic simulations of the circuits. A typical power grid model contains several million nodes, and up to hundreds of thousands of input current sources. There are some nodes with known voltages in the grid, and are modeled as nodes connected with DC voltage sources. For C4 power grids, the known voltage nodes can be internal nodes inside the power grid. Fig. 4.1 shows an RLC power grid model.

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

$$\mathbf{G}\mathbf{x}(t) + \mathbf{C}\frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t} = \mathbf{B}\mathbf{u}(t), \tag{4.1}$$

where $\mathbf{G} \in \mathbb{R}^{n\times n}$ is the conductance matrix, $\mathbf{C} \in \mathbb{R}^{n\times n}$ is the matrix resulting from charge storage elements, $\mathbf{B} \in \mathbb{R}^{n\times m}$ is the input selector matrix, $\mathbf{x}(t) \in \mathbb{R}^n$ is the vector of time-varying node voltages and branch currents of inductors and voltage sources, and $\mathbf{u}(t) \in \mathbb{R}^m$ is the vector of independent power sources.

Suppose the backward Euler method is applied to the integration of this dynamical system, the transient behavior of this power grid can be solved step by step from a given

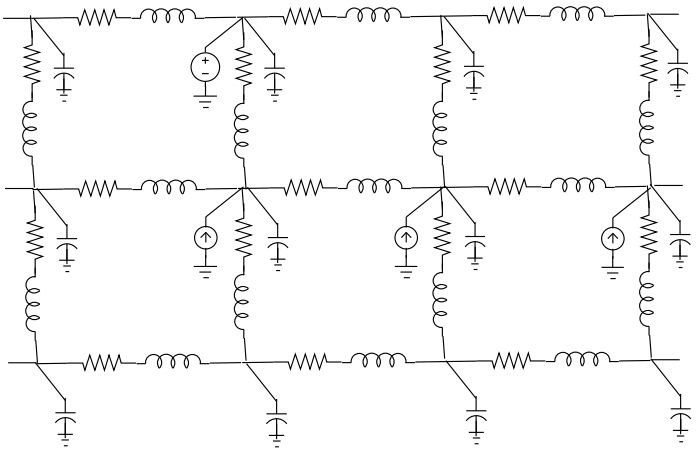


Figure 4.1: An RLC model of power grid network.

initial condition $\mathbf{x}(0)$ using

$$(\mathbf{G} + \frac{1}{h}\mathbf{C})\mathbf{x}(t+h) = \frac{1}{h}\mathbf{C}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t+h), \tag{4.2}$$

where h is the time step length. If a fixed time step h is chosen, then the left-hand side matrix, $\mathbf{G} + \frac{1}{h}\mathbf{C}$, will remain the same along all time steps. Hence, its LU factors can be factorized only once and reused for the triangular solves in the following time steps.

Though power grids are used as our examples for ETBR-GPU tool, we remark that our tool is not limited to solve power grid problems, but can be applied to any linear dynamic networks.

4.2.2 Review of reduction-based simulation methods

Model order reduction is usually used to reduce the circuit system complexity, so that simulation can be carried out on the reduced system to boost efficiency. Existing approaches mainly consist of two categories: Krylov subspace based methods, like EKS/IKES methods [46,49], and (fast) truncated balanced realization based method, like (ETBR) [50].

We in this work use ETBR method because it is more amenable for parallel computing as each frequency domain response can be computed independently, if compared to sequential computing of the Krylov subspace moments in EKS.

Algorithm 5 Extended truncated balanced realization (ETBR) method

Input: MNA matrices of the linear system, $\mathbf{G}$, $\mathbf{C}$, $\mathbf{B}$, input source $\mathbf{u}(t)$, and reduced order (number of samples), q

Output: Reduced system matrices $\hat{\mathbf{G}}$, $\hat{\mathbf{C}}$, $\hat{\mathbf{B}}$

1: Convert all the input signals $\mathbf{u}(t)$ into $\mathbf{u}(s)$ using FFT.

2: Select q frequency points $s_1, s_2, \ldots, s_q$ over the frequency range.

3: Compute $\mathbf{z}_k = (s_k\mathbf{C} + \mathbf{G})^{-1}\mathbf{B}\mathbf{u}(s_k)$. *// Parallel multithread*

4: Form the matrix $\mathbf{Z} = [\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_q]$.

5: Perform SVD on $\mathbf{Z}$, such that $\mathbf{Z} = \mathbf{V}\mathbf{S}\mathbf{U}^{\mathrm{T}}$.

6: $\hat{\mathbf{G}} = \mathbf{V}^{\mathrm{T}}\mathbf{G}\mathbf{V}$, $\hat{\mathbf{C}} = \mathbf{V}^{\mathrm{T}}\mathbf{C}\mathbf{V}$, and $\hat{\mathbf{B}} = \mathbf{V}^{\mathrm{T}}\mathbf{B}$,

The ETBR method we employed in this work is summarized in Algorithm 5. After the algorithm, the original system in Eq. (4.1) is reduced to

$$\hat{\mathbf{G}}\hat{\mathbf{x}}(t) + \hat{\mathbf{C}}\frac{\mathrm{d}}{\mathrm{d}t}\hat{\mathbf{x}}(t) = \hat{\mathbf{B}}\mathbf{u}(t), \tag{4.3}$$

and this reduced system can be simulated in time domain more efficiently. To retrieve the original waveforms, a left multiplication of the basis matrix $\mathbf{V}$ on the reduced circuit state $\hat{\mathbf{x}}(t)$ is required, i.e., $\mathbf{x}(t) \simeq \mathbf{V}\hat{\mathbf{x}}(t)$. Note that as congruence transformation is used for the reduction process with orthogonal columns in the projection matrix (using Arnoldi-like process), the reduced system is guaranteed to be stable. As far as simulation is concerned, this is good enough. If all the observable ports are also the current source nodes, i.e.,

$\mathbf{y}(t) = \mathbf{B}^{\mathrm{T}}\mathbf{x}(t)$, where $\mathbf{y}(t)$ is the voltage vector at all observable ports, then the reduced system is passive.

We observe that for the ETBR algorithm, computing $\mathbf{z}_k$, shown in Line 3, can be easily parallelized as each $\mathbf{z}_k$ can solved independently. Solving $\mathbf{z}_k$ using direct sparse solvers however is challenging to parallelization in GPU as memory access of sparse matrices are difficult to be optimized. As a result, $\mathbf{z}_k$ will be computed on multi-core CPU platforms. After the reduction, the reduced system matrices are dense, whose simulation is better performed on GPU than CPU. NVIDIA has released good linear algebra libraries, i.e., CUBLAS [55], for dense matrices, such as LU triangular solve on GPU platforms.

4.3 The proposed ETBR-GPU method

In this section, we first provide the flow of our heterogeneous ETBR-GPU method, which does MOR on multi-core CPU and calculates the transient simulation of reduced system on GPU. Then we discuss the implementation of our GPU program.

4.3.1 The overall algorithm flow

As mentioned before, to cope with the large scale linear system model, ETBR is first applied to produce the reduced system. Once the reduced system matrices are ready, they are transferred to GPU global memory for transient simulation. Meanwhile, the input stimulus information is also transferred to GPU memory, so the input source interpolation and evaluation of $\mathbf{u}(t)$ can be run in parallel. This is important as there are a huge number of input current sources and their numerical interpolation takes significant amount of time.

Fortunately, $\mathbf{u}(t)$ can be computed independently at different time t, although the transient time steps have to be calculated one by one in a sequential way. As we will show below, we can compute all vectors of $\mathbf{u}(t)$ very efficiently by harvesting the massive data parallel power of GPU. Also on GPU side, backward Euler equation's left-hand side matrix is formed and factorized in to LU factors, which are saved for repeated use in all time steps. The overall flow of our new method is shown in Fig. 4.2, which has two main parts, the CPU part (host) and GPU part (device) as clearly marked in the figure. The CPU part mainly reads and parses the netlist, run ETBR of multi-core parallelism to generate the reduced system, while the GPU part takes care of the input source evaluation and LU triangular solves of transient integration equations, which are the most time-consuming parts.

4.3.2 Parallel input source interpolation on GPU

In transient simulation, current sources attached onto the power grid need to be evaluated or interpolated repeatedly at all time steps. Typically, the time varying current sources are modeled as general piecewise linear waveforms, and hence, for each time step, the contribution of each source to the input vector $\mathbf{u}(t)$ has to be interpolated using the two neighboring wave points of time t. Though linear interpolation is a simple operation with low cost, the computing efforts for interpolation of all the sources, usually in the order of the total number of nodes in the network, is a significant port of the total simulation time.

Since the elements of $\mathbf{u}(t)$, contributed from different sources and at different time steps, are independently calculated, this task can be easily accelerated by parallel GPU computing. In our proposed simulator, a parallel GPU kernel will launch a multiple of thread blocks to evaluate the sources. Each thread block is responsible for interpolating

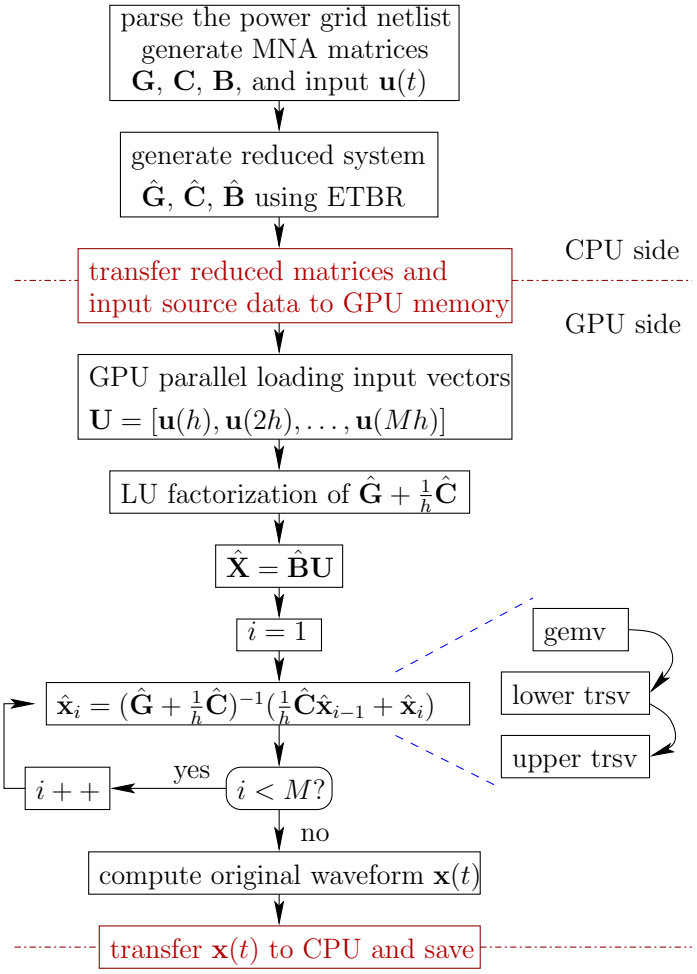


Figure 4.2: The flow of GPU accelerated ETBR based power grid analysis.

one specific source, and the threads inside the block calculate the source's values at a group of, say, 256, time steps. Fig. 4.3 illustrates the thread organization for parallel source interpolation. Note that there can be many concurrent blocks working on the same source, but at different time step groups.

Algorithm 6 GPU Parallel evaluation of input sources for transient power grid analysis

1: Configure grid and block dimensions of the GPU kernel according to number of sources and number of time steps.
2: **for** all sources **do** *// launch threads in grids*
3: Specify the block index for corresponding input source.
4: Read piecewise linear (PWL) or pulse waveform parameters into shared memory.
5: **for** all time steps **do** *// threads in each block calculate values for the same source*
6: Calculate the time t handled by each thread.
7: Use t to judge the location in a PWL or pulse.
8: Linear interpolate PWL or pulse to get source value.
9: Save value element to $\mathbf{u}(t)$.
10: **end for**
11: **end for**

For GPU computing, the main challenge is to allow fast memory access by threads or reduce memory traffic by reuse via shared memory (or texture memory) within blocks. In this way, GPU cores can be busy all the time without waiting for memory access. In GPU, fast global memory access by threads can be done by coalesced memory access, where a half warp (or a warp) of threads (16 or 32 threads respectively) can read their data from the global memory in one read access. Coalesced memory access requires the data to be

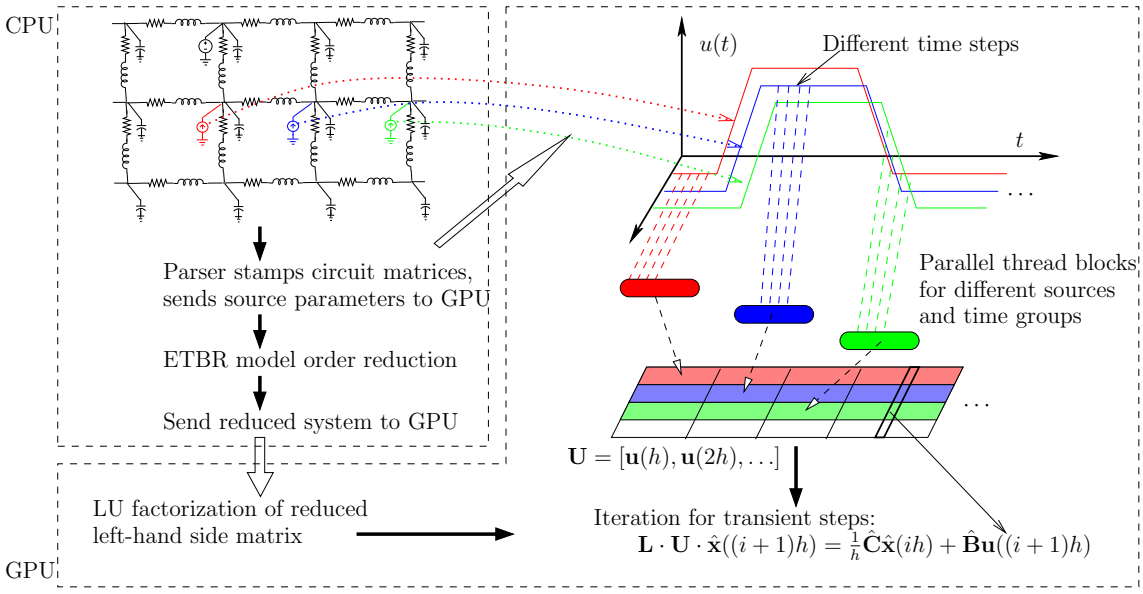


Figure 4.3: ETBR-GPU flow and GPU-accelerated current source evaluation.

arranged continuously in memory and consecutive with respective to the involved thread indices. As a result, for the parallel source evaluation, we need to align the values of one source at all time steps in one row. Hence, once all N_{cur} sources and all M time steps are evaluated, we obtain a matrix $\mathbf{U}$ of dimension N_{cur}-by-M. In this way, the i-th column of $\mathbf{U}$ is exactly the input vector $\mathbf{u}(t)$ when $t = i \cdot h$, and can be used in computing the right-hand side of transient simulation.

This process of memory allocation and parallel interpolation of sources is demonstrated in Fig. 4.3.

4.3.3 Solving transient steps on GPU

ETBR is based on model order reduction technique to reduce the circuit complexity. As a result, the reduced system matrices are dense due to the nature of the projection

based reduction. Solving dense matrices is very expensive for traditional single-core or even multi-core CPU architecture. However, dense matrix solving is very amenable for GPU computing as dense matrices based LU factorization and triangular factor solvers on GPU platforms have been well studied and implemented in CUBLAS (for triangular matrices solvers) [55] and in MAGMA library [56] due to more regular memory access patterns and simple data dependency of dense or full matrices. For instance, for LU factorization using right-looking method with look-ahead techniques, the row exchanging can be done very efficient in the row-major format as coalescent memory access can be explored.

To calculate the reduced state variables $\hat{\mathbf{x}}$ at the transient time steps, the backward Euler equation are solved by reusing the saved LU factors of $\hat{\mathbf{G}} + \frac{1}{h}\hat{\mathbf{C}}$. Since a fixed transient step length h is used, the LU factors, or the lower and upper triangular matrices will not change. Once they are ready in GPU global memory, there is no need to update them as the transient steps move on. Each time step just call CUBLAS triangular solve functions, `trsv`, to calculate the forward substitution and backward elimination, as shown in Algorithm 7.

Algorithm 7 GPU transient iteration with LU triangular solves

1: $\hat{\mathbf{X}} \leftarrow \hat{\mathbf{B}}\mathbf{U}$

2: **for** $i = 1$ to M **do**

3: $\quad \hat{\mathbf{x}}_i \leftarrow (\hat{\mathbf{C}}/h)\hat{\mathbf{x}}_{i-1} + \hat{\mathbf{x}}_i$. *// cuBLAS*

4: $\quad$ Permutation. $\hat{\mathbf{x}}_i \leftarrow \mathbf{P}\hat{\mathbf{x}}_i$ *// Customized GPU kernel function*

5: $\quad \hat{\mathbf{x}}_i \leftarrow \mathbf{L}^{-1}\hat{\mathbf{x}}_i$. *// cuBLAS*

6: $\quad \hat{\mathbf{x}}_i \leftarrow \mathbf{U}^{-1}\hat{\mathbf{x}}_i$. *// cuBLAS*

7: **end for**

4.4 Numerical results

The ETBR-GPU tool is implemented in C programming language. The GPU part of the proposed new method is incorporated into the main program with CUDA C programming interface.

The accuracy and efficiency of the program are tested on several benchmark circuits, including two industrial benchmark power grid network circuits from IBM [54]. To put our new simulator's performance into a right perspective, we compare ETBR-GPU with the multi-core CPU version of ETBR and a standard LU-based method based on UMFPACK [57], which are both implemented in C. We remark that we do not compare ETBR-GPU with any iterative solvers as most of existing iterative solvers are highly tuned to specific problems and are not general enough for commonplace linear dynamic systems. On the other hand, ETBR or ETBR-GPU is a general simulator for linear dynamic systems and it does not assume or exploit any structures of the given systems. As a result, it will be fair to compare ETBR-GPU with the general LU-based simulator.

The computer running these programs is a Linux server with two-sockets, and each socket hosts one Intel 2.4 GHz Xeon Quad-Core CPU chip. As a result, there are 16 threads from the 8 cores on the server. The host side has a total of 36 GBytes memory available for the two quad-core CPUs. Meanwhile, the GPU card installed on this server is Tesla C2070 containing 448 cores running at 1.15 GHz and up to 5 GBytes global memory.

We have 5 benchmark circuits with complexities ranging from the order of 10^4 to 10^6 nodes. All of them are RLC circuits and two of them come from IBM as mentioned before. Fig. 4.4 shows the simulation results of an IBM benchmark, imbpg1t. It is a voltage

waveform at node `n0_2679_17913`. The errors of transient waveforms using ETBR reduced models are shown in Fig. 4.5, which shows about 5% maximum relative error.

Next, we study the speedup of the ETBR-GPU over ETBR and LU-based general simulators. The statistics are shown in Table 4.1, where T_{LU} is the time for LU-based method, T_{ETBR} is the CPU time for reduction only. T_{CPU} is the time for transient simulation of reduced systems on CPU and T_{GPU} is the time for transient simulation of reduced systems on GPU. Column-12 is the speedup of the transient simulation (only) of GPU over CPU. Column-13 is the speedup of total time of ETBR-GPU over LU-based method. And Column-14 is the ETBR-GPU speedup over the CPU-only ETBR simulation.

We have several observations. First, for the first three power grid circuits, ETBR does not need high order to achieve a good accuracy as switching speeds of currents are not very high. The ETBR-GPU speedup over LU is quite impressive (can be 100×), although ETBR-GPU does not significantly faster than ETBR as the gain from the simulation of the reduce matrices are not significant due to the small sizes of the reduce matrices. Secondly, for two IBM power grid benchmarks, ibmpg1t and ibmpg2t, high order reduced models are used as the currents are switching faster. In this case, transient simulation of the reduced models becomes more expensive as the reduced matrices become larger. But ETBR-GPU can still deliver 10× speedup over the LU-based method as the GPU dense matrix solvers are much more efficient than CPU as the speedup in Column-12 shows. We also observe the with longer simulation time, we have better speedups for the two IBM cases. The reason is that in this case, GPU portion of the computing will grow more dominant in terms of total computing costs and GPU raw powers are better unleashed. The speedup of ETBR-GPU

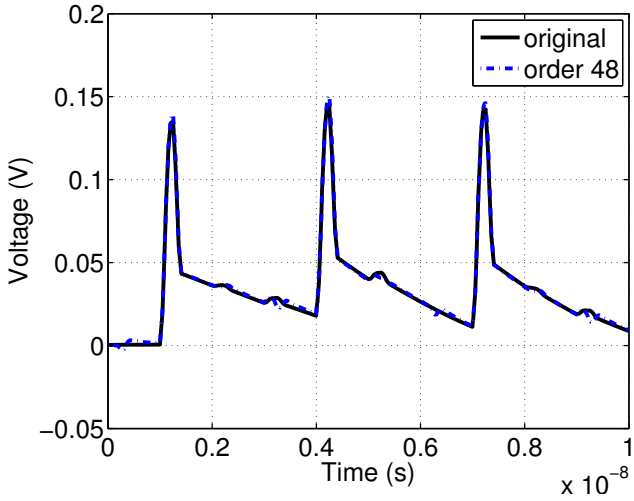


Figure 4.4: Transient waveforms of standard LU and ETBR-GPU at one port node of ibmpg1t.

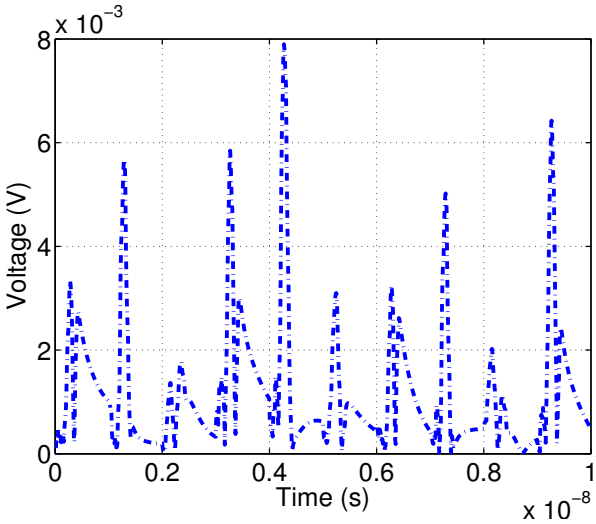


Figure 4.5: The simulation error of ETBR-GPU result of ibmpg1t using reduced model of order 48.

Table 4.1: Performance comparison of standard LU, ETBR and ETBR-GPU methods. N, and q are orders of original system and reduced system, correspondingly. N_{cur} is the number of current sources. M is the number of transient time step. The average and maximum errors are measured in volts. All time results are measured in seconds.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
ckt	N	N_{cur}	M	T_{LU}	Proposed Method—ETBR-GPU							Speedup over LU	Speedup over ETBR
					MOR by ETBR				Transient Simulation				
					q	T_{ETBR}	ave err	max err	T_{CPU}	T_{GPU}	Speedup		
pgckt1	89,200	1,000	20,000	162.1	16	2.7	4e−6	9e−5	12.2	5.8	2.1×	18×	1.8×
pgckt2	366,400	1,000	20,000	656.5	8	3.1	1.3e−5	5e−4	6.5	3.0	2.2×	107×	1.6×
pgckt3	3,064,800	4,000	2,000	714.2	8	37.2	1.4e−4	5e−4	33.5	16.6	2×	13×	1.3×
ibmpg1t	39,680	10,774	10,000	107.6	48	4.7	1e−3	8e−3	158.6	1.66	96×	17×	26×
			20,000	206.8		4.9	1e−3	8e−3	305.2	2.92	104×	25×	39×
			40,000	412.7		5.0	1e−3	8e−3	597.1	5.81	102×	37×	55×
			80,000	805.2		5.0	1e−3	8e−3	1126	10.0	112×	52×	75×
ibmpg2t	164,237	36,838	5,000	324.4	32	54.0	1e−3	6e−3	131.2	13.2	10×	4.5×	3×
			10,000	650.7		55.1	1e−3	6e−3	263.8	13.6	20×	9.5×	5×
			20,000	1127.4		55.9	1e−3	6e−3	491.5	15.2	32×	16×	8×
			40,000	2525.9		60.4	1e−3	6e−3	1046.6	18.3	57×	32×	14×

over LU will be up-bounded by the speedup numbers in Column-12 in theory in this case, which can be 100× as shown in the table.

4.5 Summary

A new transient analysis method, ETBR-GPU, has been discussed for general linear dynamic networks, such as on-chip power grid networks, using heterogeneous computing systems with both GPU and multi-core CPU. The new algorithm is a reduction based simulation technique and is very amenable for parallelization on the hybrid multi-core CPU and GPU platforms, where coarse-grained task-level and fine-grained lightweight thread-level parallelism can be both exploited. The proposed method can analyze any linear networks with complicated structures and macro-models. It especially favors transient simulation with long simulation periods as reduction costs in frequency domain is less sensitive to the simulation time. Experimental results show that the new method can achieve about one or two orders of magnitudes speedup when compared to the general LU-based simulation method on some recently published IBM power grid benchmark circuits.

Chapter 5

Parallel thermal analysis of 3D ICs on GPU-CPU platforms

Cooling related thermal-induced reliability problems are considered as the major hurdles for the emerging through silicon vias (TSV) based 3D integrated circuits (3D ICs) due to increased power density and elevated thermal resistances for the added embedded layers. Advanced cooling techniques such as integrated inter-tier liquid cooling may be required to resolve the demanding cooling problems. Fast and accurate thermal analysis techniques are crucial for designing 3D ICs and associated packages to find well traded-off cooling solutions.

In this chapter, an efficient parallel finite difference based thermal simulation algorithm for 3D-ICs using GMRES solver on CPU-GPU platforms is studied. First, unlike existing fast thermal analysis methods where macro-modeling was used, the new method starts from the basic physics heat equations to model 3D-ICs with inter-tier liquid cooling

micro-channels and directly solves the resulting partial differential equations. Second, we implement a new GMRES solver to computing the resulting thermal systems using GPU parallel acceleration. We also explore different preconditioners (implicit and explicit) and study their performances on thermal circuits and other types of matrices. Experimental results show the proposed GPU GMRES solver can deliver order of magnitudes speedup over the parallel LU solver and up to 4× speedup over CPU GMRES solver for both DC and transient thermal analyses on a number of thermal circuits and other published problems.

5.1 Introduction

Three-dimensional integrated circuit technology is viewed as a necessary driving force to maintain the trend described by Moore's Law [58, 59]. The 3D stacked integration based on TSV connection technique enables heterogeneous integration of processor cores, memories, and analog devices, and overcomes the barriers in interconnect scaling. However, it also introduces new challenges. The greatest technological challenge for 3D ICs is heat control and removal from within a 3D IC, particularly in the embedded layers. As a result, cooling and thermal problems has received much research attention.

To remove the excessive heat in 3D chips, traditional fan-based cooling techniques are not sufficient due to their limited heat removal capabilities [60]. Active cooling techniques, such as embedded micro-channel cooling, are promising alternatives. Micro-channel based liquid cooling can remove up to 200–400 W/cm^2 and has the potential to reach 1000 W/cm^2 [61, 62]. To design efficient package and 3D IC structures with advanced cooling solutions, an accurate and fast detailed transient thermal analysis is required [63, 64].

Traditional thermal analysis solves the partial thermal diffusion equation directly using numerical approaches such as finite difference method, finite element method, and computational fluid dynamics. This process is computationally intensive, especially for large-scale 3D ICs, as it requires solving a large number of linear equations arising from equivalent thermal circuit models.

To significantly improve the simulation efficiency, exploiting the parallelism of the simulation algorithms on multi-core and many-core computing platforms becomes a viable solution. The GPU family is among the most powerful many-core computing systems in mass-market use. Currently, GPUs or GPU-clusters can easily deliver tera-scale computing, which was only available on super computers in the past, for solving many scientific and engineering problems. To date, dense linear algebra support on the GPU is well developed, with its own BLAS implementation [55], but sparse linear algebra support is still limited. In [64], a CPU based sparse LU solver was used to simulate the linear systems. However, sparse LU solvers have complicated data dependency compared to dense matrix solvers, and its implementation is considered to be difficult on today's high performance GPUs (although they are some recent efforts on this direction [65]).

On the other hand, iterative solvers such as generalized minimum residual method (GMRES) [66], which depends on matrix-vector multiplications and other matrix operations, are more amicable for parallelization, especially on GPU platforms. And when combined with a parallel-friendly preconditioner for better convergence, GMRES can be significantly faster than a sparse LU solver especially for large problems. Also practically, today's high performance GPU is seen as collaborator and accelerator for multi-core CPU.

GPU works in tandem with CPU on same computing node connected by high-speed link like PCI-Express (PCIe) buses. Since some computing tasks, like graph based operations and small matrix operations with low computing density or low compute-to-load ratio, are not suitable for GPU computing, parallel programs have to leverage both multi-core CPU and GPU to achieve the best performance. How to partition the computing tasks and reduce the data traffic among CPU and GPU becomes a critical issue for GPU based heterogeneous computing and programming.

In this chapter, we propose an efficient GPU-accelerated GMRES iterative solver for finite difference thermal analysis of 3D ICs on GPU-CPU platforms. This chapter is organized as follows. Section 5.2 first reviews 3D ICs with integrated liquid cooling structure and their features from a computing perspective, and our physics based modeling for 3D ICs with integrated micro-channels is introduced. Section 5.3 describes the proposed GPU-GMRES parallel algorithm and discussions of two GPU-friendly preconditioners. We carefully partition the computing tasks among CPU and GPU in GMRES to boost the performance. We also explore different types of preconditioners and investigate their performances on our applications in section 5.4. The work is summarized by Section 5.5.

5.2 Finite difference model of 3D ICs with micro-channels

5.2.1 3D-ICs with integrated inter-tier micro-channels

Fig. 5.1 and Fig. 5.2 show a 3D system consisting of a number of stacked layers (with cores, L2 caches, crossbar, memory controllers, buffers, etc.) and micro-channels built in-between the vertically stacked layers for liquid cooling. Forced inter-layer convective

cooling with water is applied [67], and the micro-channels are distributed uniformly, and fluid flows through all channels at the same flow rate, which can be dynamically altered at runtime by water pump controller.

Laminate liquid flows in the micro-channels make the resulting heat equations more complicated as heat is removed by both heat sinks and laminate liquid flows via convection effect. To mitigate this problem, some simple models were proposed as an add-on to existing thermal models for packages and chips at the cost of the accuracy. In [64,68], liquid cooling effects are modeled by RC networks with simplified voltage-controlled current sources to model the dominant convective heat flow (in flow direction). In [69], a simple resistor model is proposed for the liquid cooling micro-channels. Instead, we consider both conductive heat flow in the solid (chips and package) and the convective heat flow in the coolant flow are considered, and directly solve the resulting partial differential equations without any approximation using finite different method.

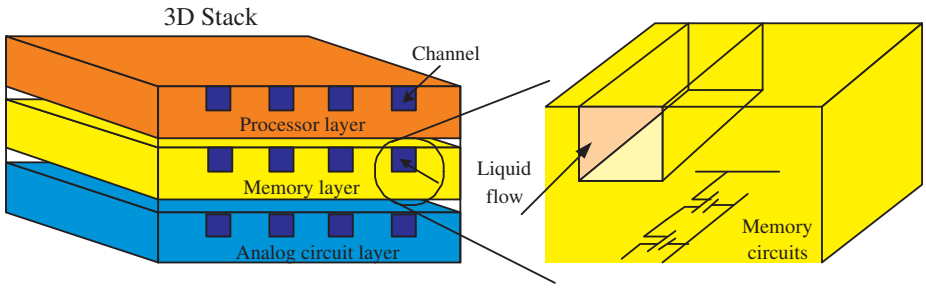


Figure 5.1: 3D stacked IC with inter-tier liquid cooling.

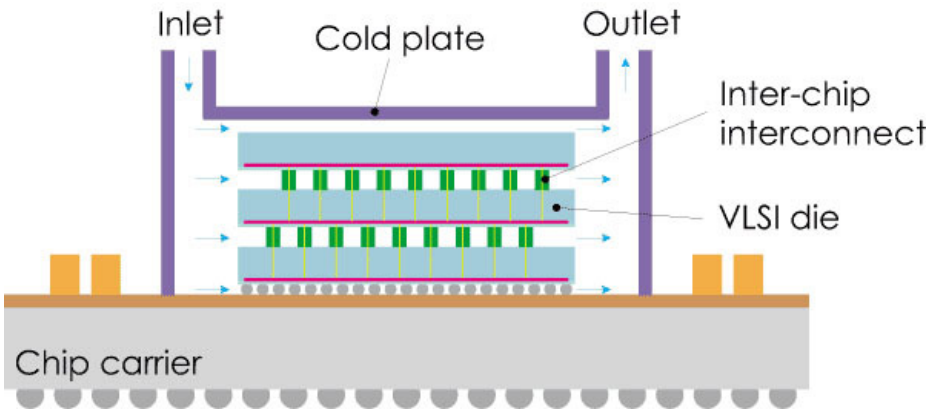


Figure 5.2: Interlayer cooling for 3D IC packages.

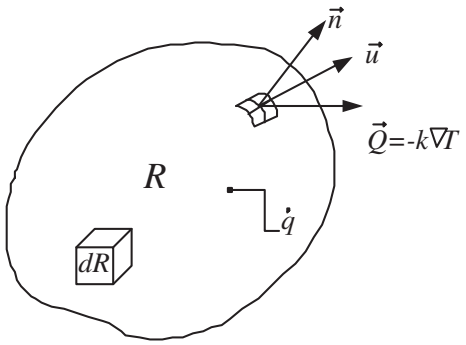


Figure 5.3: Energy conservation for a control volume.

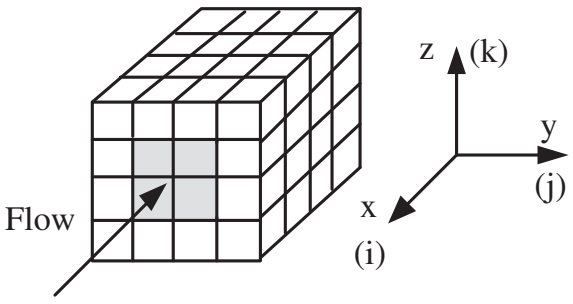


Figure 5.4: Meshed chip cell with coolant channel.

5.2.2 The proposed thermal model

We develop a thermal model for 3D stacked ICs with embedded micro-channels and heat sinks from the basic heat equations. Using energy conservation in a control volume as shown in Fig. 5.3, the heat transfer equation of incompressible material can be written as the following equation according to [70],

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_R \rho\hat{u}\,\mathrm{d}x = -\int_S (\rho\hat{h})\vec{u}\cdot\vec{n}\,\mathrm{d}S - \int_S (-k\nabla T)\cdot\vec{n}\,\mathrm{d}S + \int_R \dot{q}\,\mathrm{d}R, \tag{5.1}$$

where R is the control volume, S is its surface, $\vec{n}$ is normal to the surface, ρ is the density of the material, $\hat{u}$ is the specific internal energy, $\hat{h}$ is the convection coefficient, $\vec{u}$ is the flow rate, k is the thermal conductivity of the material, T is the temperature, and $\dot{q}$ is the volumetric rate of the heat generation inside R. By applying Gauss' theorem to Eq. (5.1) and using $\mathrm{d}u = c_p\,\mathrm{d}T$, the general form of heat equation can be written as

$$\rho c_p\frac{\partial T}{\partial t} = -\rho c_p\vec{u}\cdot\nabla T + k\nabla^2 T + \dot{q}, \tag{5.2}$$

where c_p is the specific heat. Applying finite difference method and assuming the channel is along x-axis, as illustrated in Fig. 5.4, the discretized form of Eq. (5.2) is

$$\begin{aligned}\rho c_p\frac{\mathrm{d}T}{\mathrm{d}t} = {} & \frac{k_{xx}}{\Delta x^2}(T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}) + \frac{k_{yy}}{\Delta y^2}(T_{i,j+1,k} - 2T_{i,j,k} + T_{i,j-1,k}) \\ & + \frac{k_{zz}}{\Delta z^2}(T_{i,j,k+1} - 2T_{i,j,k} + T_{i,j,k-1}) + u_{xx}\frac{\rho c_p}{2\Delta x}(T_{i+1,j,k} - T_{i-1,j,k}) + g(\vec{v},t),\end{aligned} \tag{5.3}$$

where $T_{i,j,k}$ is the temperature of element with index (i,j,k) in the meshed grid, k_{xx}, k_{yy}, and k_{zz} are thermal conductivities along x, y, and z axes, respectively, u_{xx} is the flow rate in x direction, $g(\vec{v},t)$ is the heat generation in volume $\Delta v = \Delta x\Delta y\Delta z$. Therefore, Eq. (5.3) models the thermal behavior of 3D IC stack as a thermal circuit, and it can be rearranged

into ordinary differential equation very similar to MNA form,

$$\mathbf{G}\mathbf{T}(t) + \mathbf{C}\frac{\mathrm{d}\mathbf{T}(t)}{\mathrm{d}t} = \mathbf{B}\mathbf{U}(t) \tag{5.4}$$

where $\mathbf{G}$ and $\mathbf{C}$ are the coefficient matrices that represent thermal conductance and capacitance, $\mathbf{B}$ is the input position matrix of the heat sources, $\mathbf{T}(t)$ is the vector of on-chip temperature values, and $\mathbf{U}(t)$ is the vector of input power sources.

At the four sidewalls of the channel, i.e., side = {top/bottom/left/right}, the boundary conductance is calculate as $g_{\text{side}} = h_{\text{side}}S$. This equation is used to model the heat exchange from the channel sidewall to the coolant as shown in Fig. 5.6, where h_{side} is the convection coefficient at the sidewall of micro-channel [71] and S is the area of the convective surface.

Notice that in micro-channels, each cell has the term

$$I_c = \rho C_p u_{xx} \frac{T_{i+1,j,k} - T_{i-1,j,k}}{2\Delta x}, \tag{5.5}$$

which represents the convective heat transfer along the flow direction and is the dominant term, if compared to all the other conductive terms in Eq. (5.3). The conductive heat flow linearly depends on the temperature difference between the two boundaries of the cell along the flow direction. As a result, they can be viewed as temperature-controlled heat sources, or voltage controlled current sources in circuits. Since we have the controlled sources, the resulting $\mathbf{G}$ matrix is no longer symmetric. For cells in solids (chips or packages), the flow rate $\vec{u}_{xx}$ is 0 and thus there is no convection term.

Note that our model is different from the simplified circuit model proposed in [69] which uses very small thermal resistors R_f in coolant flow direction to model the convective

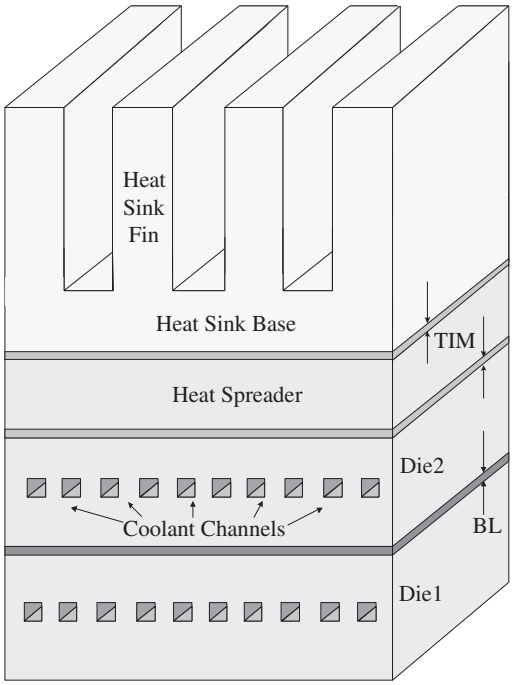


Figure 5.5: A view of 3D IC stack. BL denotes the bounding layer between two dies, and TIM denotes the thermal interface material.

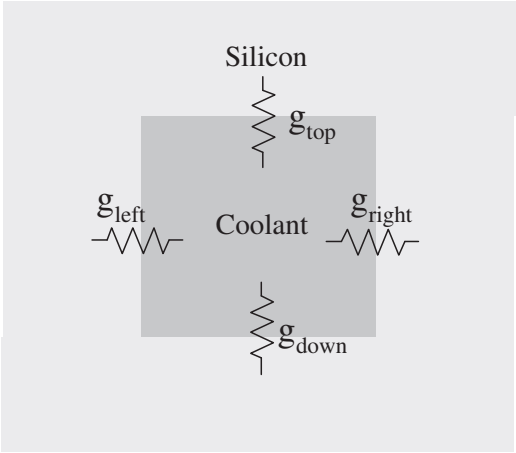


Figure 5.6: Model of heat transfer between coolant and the sidewall of the channel.

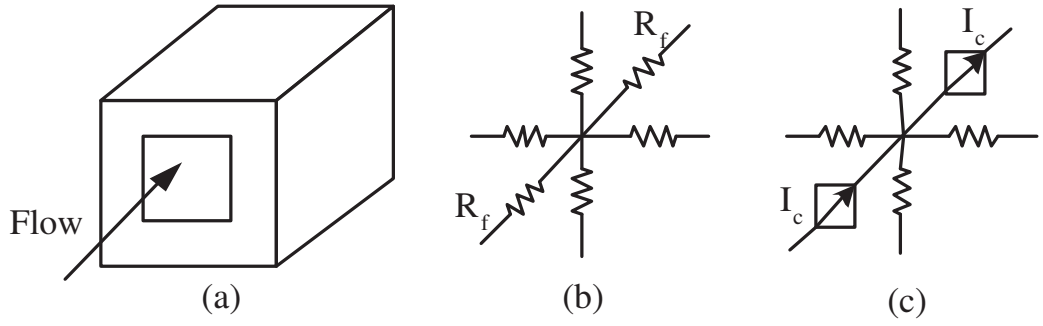


Figure 5.7: (a) Coolant flow inside a channel; (b) Modeling heat convection in the direction of the channel using thermal resistor R_f; (c) Modeling heat convection in the direction of the channel using current sources I_c derived from the energy equation.

heat exchange, as shown in Fig. 5.7(b). We in this work uses the heat-controlled temperature flow (voltage-controlled current sources) naturally derived from the energy equation (5.3) to model the heat convection in the direction of the channel without macro-modeling based approximations. This is also in contrast with the macro-models in [64, 68].

Without loss of generality, the example we used in our testing case is a stack structure shown in Fig. 5.5. It consists of 2 active layers and has a heat sink on the top. Micro-channels are embedded in the active silicon layers. The geometrical and material properties of the test stack are listed in Table 5.1. Inside the channels we assume the coolant flows to move at a constant rate.

5.3 Parallel GMRES solver on GPU-CPU platform

5.3.1 Relevant previous arts

Recently there have been several works published on GPU based circuit analysis and simulation. In [51] a multi-grid solver is used for DC analysis of power grids, while [72] used multi-grid as a preconditioner for the conjugate gradient method for DC analysis of power distribution networks. The same team also applied their multi-grid/conjugate

Table 5.1: Geometrical and material information of the 3D structure.

layers	geometry (mm)	material
die	$8 \times 8 \times 1$	silicon
thermal interface matreiral (TIM)	$8 \times 8 \times 0.25$	indium
bounding layer (BL)	$8 \times 8 \times 0.25$	silicon nitride
heat spreader	$8 \times 8 \times 0.5$	copper
heat sink base	$8 \times 8 \times 0.5$	aluminum
heat sink fin	$8 \times 8 \times 3$	aluminum

gradient solver to the same thermal problem addressed in our work by using a purely resistance based symmetric model [69]. While this model has the benefit of being symmetric, it is unclear how to control the flow rate through micro-channels by adjusting the resistor values. Instead, we base our work on the model proposed in [64], which allows specific rates of flow through the micro-channels by modeling convection along the micro-channels as a controlled current source.

There are also several papers implementing GMRES on GPU. In [73], GMRES with a block ILU preconditioner is parallelized on GPU. However, it only makes a small part of GMRES parallel, leaving many operations in serial. A more thorough GPU implementation of GMRES is proposed in [74], but it uses an inefficient sparse matrix format [75] and does not mention what precision was used for testing, making comparison difficult. The most recent paper [76] measures precision by comparing their GPU GMRES to a customized CPU based solver, rather than to a validated solver, leaving open a question about the actual accuracy of their implementation.

Since we will propose our GPU GMRES as a fast solver to the thermal problem, it is beneficial to quickly review the basics of GMRES. This name is short for generalized minimum residual method, an iterative method for solving large-scale systems of linear equations ($\mathbf{Ax} = \mathbf{b}$), where $\mathbf{A}$ is sparse in our case. Algorithm 8 shows the standard Krylov subspace based GMRES method with left preconditioning [77], which uses projection method to form the m-th order Krylov subspace [66, 77], e.g.,

$$\mathcal{K}_m = \text{span}(\mathbf{r}_0, \mathbf{MA}\mathbf{r}_0, (\mathbf{MA})^2\mathbf{r}_0, \ldots, (\mathbf{MA})^{m-1}\mathbf{r}_0), \tag{5.6}$$

where $\mathbf{r}_0 = \mathbf{b} - \mathbf{Ax}_0$ is the initial residual, and $\mathbf{M}$ is the preconditioner. After orthogonal-

ization and normalization, the orthonormal basis of this subspace is $\mathbf{V}_m$. To generate the Krylov subspace in GMRES, Arnoldi iteration is employed to form the $\mathbf{V}_m$. Each Arnoldi iteration generates a new basis vector and is appended to the previous Krylov subspace basis $\mathcal{K}_j$ to obtain the augmented subspace $\mathcal{K}_{j+1}$. The Arnoldi iteration also creates an upper Hessenberg matrix $\tilde{H}_m$ used to check the solution at the current iteration. As a result, the approximated solution $\mathbf{x}$ becomes the linear combination of $\mathbf{x}_m = \mathbf{x}_0 + \mathbf{V}_m\mathbf{y}_m$, where $\mathbf{y}_m$ is calculated in Line 12 of Algorithm 8.

5.3.2 Parallelization on GPU-CPU platforms

To parallelize the GMRES solver, we need to identify several computation intensive steps in Algorithm 8. There exist many GPU-friendly operations in GMRES, such as vector addition (`axpy`), 2-norm of vectors (`nrm2`), and sparse matrix-vector (SpMV) multiplication (`csrmv`). Based on the examples we focus on, we have noticed that SpMV takes up to 50% of the overall runtime to build the Krylov subspace shown in Eq. (5.6). A lot of efforts have been made already to parallelize these routines in generic parallel algorithms for dense matrix and vector operations (CUBLAS) and sparse matrix computations (CUSPARSE) [55].

We remark that not all computing steps in GMRES are suitable for GPU platforms. GPU is powerful, but only when finding its niche with high computing density with small number of data communication traffics. GPU parallelization should be viewed as the CPU-GPU collaboration, where GPU serves as co-processor for CPU such that GPU-friendly computing steps are identified and put into GPUs. In this collaboration, the consideration of memory transfers and communication costs between GPU and CPU is also nontrivial as

Algorithm 8 GMRES with Left Preconditioning

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

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

1: $\mathbf{r}_0 = \mathbf{M}(\mathbf{b} - \mathbf{A}\mathbf{x}_0)$

2: $\beta = \|\mathbf{r}_0\|_2$, $\mathbf{w} = \mathbf{r}_0/\beta$

3: **for** $j = 1$ to m **do** *// Arnoldi iteration*

4: $\quad \mathbf{w} = \mathbf{M}\mathbf{A}\mathbf{v}_j$

5: $\quad$ **for** $i = 1$ to j **do** *// Orthogonalization*

6: $\quad\quad h_{i,j} = \mathbf{w}_i^{\mathrm{T}}\mathbf{v}_j$

7: $\quad\quad \mathbf{w} = \mathbf{w} - h_{i,j}\mathbf{v}_i$

8: $\quad$ **end for**

9: $\quad h_{j+1,j} = \|\mathbf{w}\|_2$, $\mathbf{v}_{j+1} = \mathbf{w}/h_{j+1,j}$

10: **end for**

11: $\mathbf{V}_m = [\mathbf{v}_1, \ldots, \mathbf{v}_m]$, $\tilde{\mathbf{H}}_m = \{h_{i,j}\}_{1 \le i \le j+1,\ 1 \le j \le m}$

12: $\mathbf{y}_m = \mathrm{argmin}_{\mathbf{y}} \|\beta\mathbf{e}_1 - \tilde{\mathbf{H}}_m\mathbf{y}\|_2$, $\mathbf{x}_m = \mathbf{x}_0 + \mathbf{V}_m\mathbf{y}_m$

13: **if** tolerance satisfied **then**

14: $\quad$ Return

15: **else**

16: $\quad \mathbf{x}_0 = \mathbf{x}_m$ and go to Line 1

17: **end if**

we will see later. As a result, the algorithms are actually implemented on the GPU-CPU platform, which is well illustrated in our proposed GPU-GMRES algorithm.

GPU computing is typically limited by the data transfer bandwidth as GPU favors computationally intensive algorithms [13]. Hence, how to wisely partition the data between CPU memory (host side) and GPU memory (device side) to minimize data traffic is crucial to the overall computing performance. In the sequel, we make some detailed analysis first for GMRES in Algorithm 8. Although GMRES tends to converge quickly for most circuit examples, i.e., the iteration number $m \ll n$, the space needed to store the subspace $\mathbf{V}_m$ with a size of n-by-m, i.e., m column vectors with n-length, is still big. Therefore, transferring the memory of the subspace vectors between CPU memory and GPU memory is not an efficient choice. In addition, every newly generated matrix-vector product needs to be orthogonalized with respect to all its previous basis vectors in the Arnoldi processes. To utilize the data intensive capability of GPU, we keep all the vectors of $\mathbf{v}_m$ in GPU global memory, thus allows GPU to handle those operations, such as `axpy`, `dot`, and `nrm2`, in parallel.

On the other hand, it is better to keep the Hessenberg matrix $\tilde{\mathbf{H}}$, where intermediate results of the orthogonalization are stored, at CPU host side. This comes with the following reasons. First, its size is $(m+1)$-by-m at most, rather small if compared with circuit matrices and Krylov basis vectors. Besides, it is also necessary to triangularize $\tilde{\mathbf{H}}$ and check the residual in each iteration so the GMRES can return the approximate solution as soon as the residual is below a preset tolerance. Hence, in light of the sequential nature of the triangularization, the small size of Hessenberg matrix, and the frequent inspection of

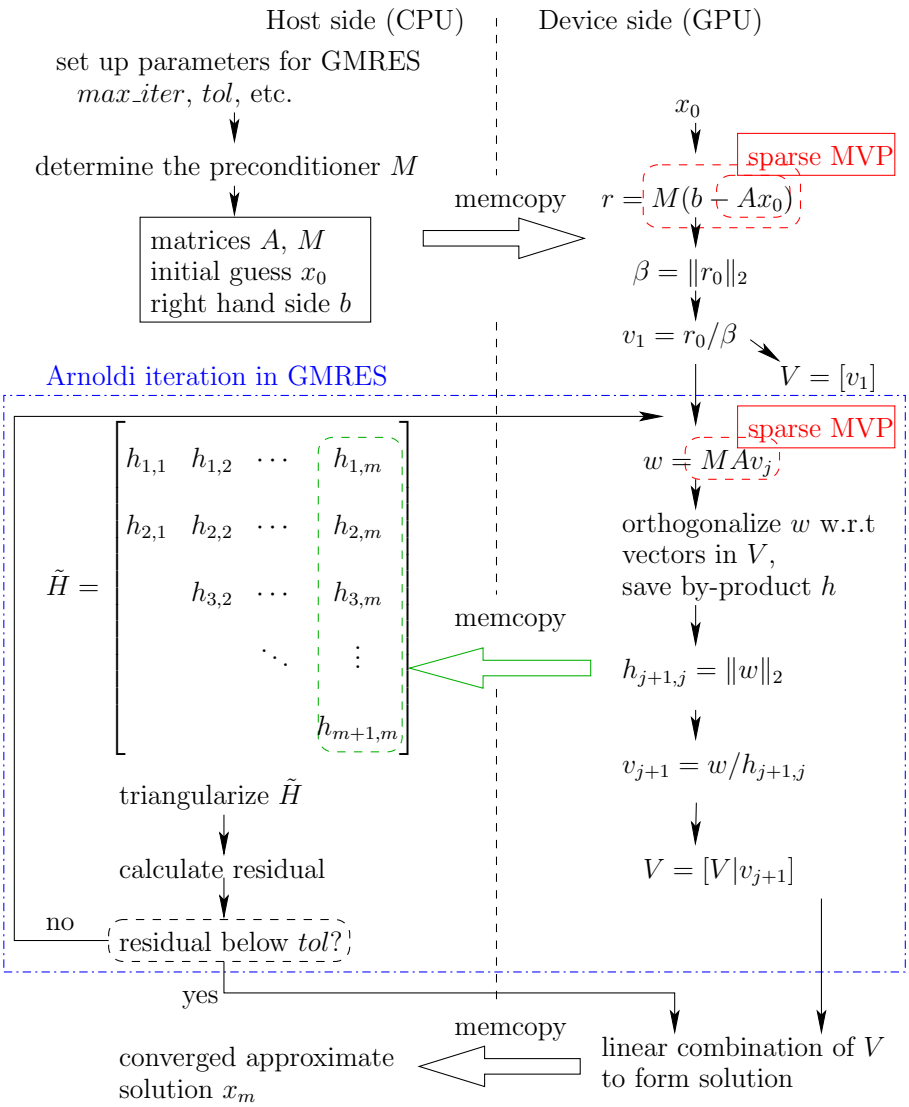


Figure 5.8: The proposed GPU-accelerated parallel preconditioned GMRES solver. We also show the partitioning of the major computing tasks between CPU and GPU here.

values by the host, it is preferable to allocate $\tilde{\mathbf{H}}$ in host memory. As shown in Algorithm 8, the memory copy from device to host is called each time when Arnoldi iteration generates a new vector and the orthogonalization produces a new vector $\mathbf{h}$, which is the $j+1$ column of $\tilde{\mathbf{H}}$, and is transferred to the CPU, where a least square minimization (a series of Givens rotations, in fact) is performed to see if the desired tolerance of residual is met. Our observation shows that the data transfer and subsequent CPU based computation takes less than 5% of the total run time.

Fig. 5.8 illustrates the computation flow, the partitions of the major computing steps and the memory accesses between CPU and GPU during the operations we mentioned above.

5.3.3 GPU-friendly implementation of preconditioners

One important aspect of the iterative solver is the preconditioner. Preconditioners increase the rate of convergence and thus reduce the number of iterations. A well chosen preconditioner will potentially make GMRES much faster than the one without preconditioner. In this section, we discuss the preconditioner for GPU GMRES.

Most existing preconditioners can be broadly classified as being either explicit or implicit [78]. A preconditioner is implicit if its application within each step of the chosen iterative method requires the solution of a linear system (`csrsv`). With implicit preconditioner, we choose a non-singular matrix $\mathbf{M}$ where $\mathbf{M} \approx \mathbf{A}$, and $\mathbf{M}$ should satisfy the requirement that solving a system with matrix $\mathbf{M}$ is easier than solving the original system of $\mathbf{A}$. In contrast, for explicit preconditioner, the approximate inverse form of $\mathbf{A}$ is calculated first and will be known to iteration solvers. Hence, the preconditioning operations

become one or more matrix-vector products (`csrmv`).

Implicit preconditioners have been intensively studied and they have been successfully employed in a number of applications. However, during recently years, an increasing amount of interest has been shifted to explicit preconditioners [78, 79]. There are several reasons for this shift. In the first place, the preconditioning operations of implicit preconditioners, i.e., solving of the linear system $\mathbf{M}$, are serial in nature and difficult to be parallelized. Although sophisticated parallel strategies have been adopted, only limited speedup is gained [74]. Another drawback of implicit preconditioners is the possibility of breakdowns during the incomplete factorization process. The breakdown cannot even be avoided even if pivoting is applied [80,81]. With the evolution of the architectures in modern computers, `csrsv` during implicit preconditioning have been found to be the bottleneck limiting the performance. On the other hand, explicit preconditioning operations, i.e., `csrmv`, can be parallelized easily and efficiently, which is favorable for modern computers. However, the convergency rate of explicit preconditioners is usually slower if compared with implicit ones. To find out which type of preconditioners is suitable for our thermal simulation, we will study the performance of both implicit and explicit preconditioners in this chapter.

The most widely used implicit preconditioner is based on incomplete LU (ILU) decomposition. For ILU preconditioner, $\mathbf{M} = \tilde{\mathbf{L}}\tilde{\mathbf{U}}$, where $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{U}}$ are sparse triangular matrices which approximate the $\mathbf{L}$ and $\mathbf{U}$ factors of $\mathbf{A}$ respectively, i.e., $\mathbf{A} \approx \tilde{\mathbf{L}}\tilde{\mathbf{U}}$. Applying ILU preconditioner requires two triangular solves and as mentioned earlier, its serial nature limits the performance of parallel machines. For ILU preconditioners, the more fill-ins in $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{U}}$, the more similar are the preconditioner $\mathbf{M}$ and the original matrix $\mathbf{A}$, and hence

better convergence GMRES will have. However, this does not simply result in speedup. When parallelizing the `csrsv` of $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{U}}$ matrices, the techniques based on level-scheduling are needed [82]. The denser the triangular matrix is, the more dependency it requires, which will severely degrade the performance. Therefore, we adopt the strategy of ILU0 with the fewest fill-ins. Hence, we can use the non-zero pattern of $\mathbf{A}$ as the symbolic structure to guide the incomplete factorization in parallel [83].

We use the approximate inverse (AINV) preconditioners with tolerance dropping in [78, 81] as our explicit preconditioner. The norm value of $\|\mathbf{I}-\mathbf{MA}\|$ is used to define the similarity of $\mathbf{M}$ and $\mathbf{A}^{-1}$. Normally, the matrix norm is taken as the Frobenius norm or its weighted variants [79, 81],

$$\|\mathbf{I}-\mathbf{MA}\|_F^2 = \sum_{i=1}^{n} \|\mathbf{e}_i - \mathbf{Am}_i\| \tag{5.7}$$

where $\mathbf{e}_i$ is the i-th unit vector and $\mathbf{m}_i$ is the i-th column of $\mathbf{M}$. With AINV preconditioner, we approximate the inverse of the original matrix $\mathbf{A}$ with three matrices [79]:

$$\mathbf{A}^{-1} \approx \mathbf{ZD}^{-1}\mathbf{W}^{\mathrm{T}}, \tag{5.8}$$

where $\mathbf{Z}$ and $\mathbf{W}$ are two unit upper triangular matrices, $\mathbf{D}$ is a diagonal matrix. $\mathbf{Z}$ and $\mathbf{W}$ are similar to $\mathbf{U}^{-1}$ and $\mathbf{L}^{-1}$ triangular parts in LDU decomposition of $\mathbf{A} = \mathbf{LDU}$ respectively. They can be directly obtained by means of a bi-conjugation process [81]. If we define $\mathbf{M}_l = \mathbf{W}^{\mathrm{T}}$ and $\mathbf{M}_r = \mathbf{ZD}^{-1}$, the preconditioning operations are two matrix-vector product of $\mathbf{M}_l$ and $\mathbf{M}_r$ respectively, which is easily to be parallelized. For our program, we first compute the preconditioner $\mathbf{M}$ in CPU, and then transfer it to GPU for parallel computation of preconditioned GMRES.

5.4 Numerical results

The proposed algorithm has been implemented in C and CUDA C for GPU programming. Our GPU card is Tesla C2070 GPU card with 448 cores running at 1.15 GHz with 5 GB of global memory. It has a double precision floating point peak performance of 515 GFLOPS and 1.03 TFLOPS single precision peak performance.

Although the CPU results were tested on a quad-core Xeon E5620 machine at 2.00 GHz with 28 GBytes memory. For a thorough comparison, serial GMRES on CPU, a parallel LU solver (SuperLU_MT), and parallel GMRES on CPU-GPU platform with preconditioners are compared for the full thermal simulation on a number of packages. SuperLU_MT is a commonly preferred and publicly available parallel LU solver [84].

Our test cases consist of five thermal circuit models generated with our proposed physics based thermal modeling method. To demonstrate the practicability and generality of our solver, two circuit examples from the UFL Matrix Collection [85] and four 3D-ICE test cases with package systems [64] are also included in our experiments. These general circuit and thermal circuit examples have varied sizes and density ratios of the sparse patterns. More information on these examples can be found in Table 5.2, where "dim" is the size, i.e., number of rows and number of columns of the matrix, "nnz" is the number of nonzero elements in the matrix, and "density" is the ratio defined as $\text{nnz}/(\text{dim} \cdot \text{dim})$, which is a measure of the fullness of a sparse matrix. A typical sparsity pattern of our proposed thermal circuit model is plotted in Fig. 5.9, which is saved from example "therm5".

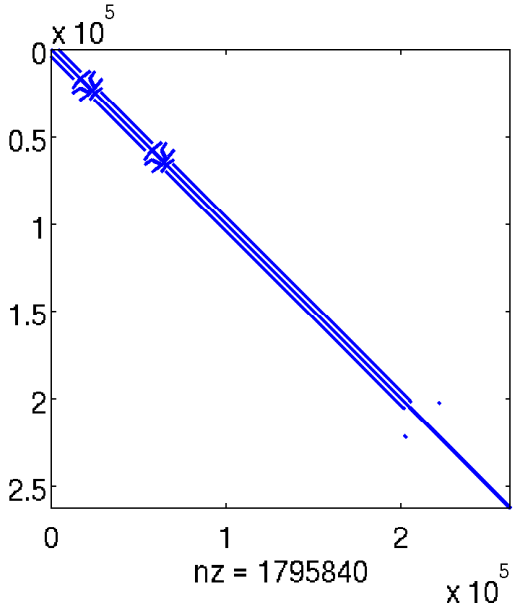


Figure 5.9: Sparsity pattern of thermal circuit model from example "therm5".

Table 5.2: Statistics for thermal circuits. "dim" stands for number of rows and number of columns of the square matrix. "nnz" is the number of nonzero elements, and density is calculated as nnz/(dim · dim).

circuit name	dim	nnz	density
G2_circuit (UFL)	150,102	438,388	1.94e−5
G3_circuit (UFL)	1,585,478	4,623,152	1.83e−6
mc2rm (3D-ICE)	20,000	109,000	2.72e−4
mc4rm (3D-ICE)	50,250	329,140	1.30e−4
pf2rm (3D-ICE)	20,000	104,100	2.60e−4
solid (3D-ICE)	10,000	44,600	4.46e−4
therm1	57,344	390,144	1.18e−4
therm2	114,688	783,872	5.96e−5
therm3	147,456	1,004,032	4.61e−5
therm4	172,032	1,174,528	3.97e−5
therm5	262,144	1,795,840	2.61e−5

Table 5.3: Comparison of solvers on $\mathbf{Ax} = \mathbf{b}$. For the last five examples from our thermal modeling method, steady state equations for the 3D IC thermal profile are used. SuperLU deploys 4 CPU threads in LU factorization. Run time in seconds. On some examples, GMRES fails to converge without preconditioner or with diagonal preconditioner, thus the run time data is not available and is not shown in the table.

1	2	3	4		5	6	7	8
circuit	SuperLU		GMRES				speedup	
name	fact.	sol.	precond		CPU	GPU	$\frac{C2}{C4+C6}$	$\frac{C2}{C6}$
G2	0.8	0.03	NON	0	8.9	2.2	0.4×	0.4×
			DIAG	0.003	0.3	0.04	19×	20×
			ILU0	0.7	0.03	0.06	1×	13×
			AINV	4.1	0.4	0.1	0.2×	8×
G3	30.7	1.9	NON	0	211	16.5	2×	2×
			DIAG	0.02	7.0	.024	697×	1023×
			ILU0	7.5	0.8	0.14	4×	219×
			AINV	42	8.4	0.68	0.7×	45×
mc2rm	0.33	0.01	ILU0	0.14	0.11	0.15	1.2×	2×
			AINV	0.18	0.14	0.15	1×	2×
mc4rm	1.63	0.05	ILU0	0.3	0.5	0.4	2×	4×
			AINV	5.0	0.5	0.2	0.3×	8×
pf2rm	0.3	0.009	ILU0	0.12	0.10	0.13	1×	2×
			AINV	1.8	0.13	0.12	.2×	3×
solid	0.03	0.002	NON	0	0.17	0.35	0.1×	0.1×
			DIAG	0.001	0.08	0.15	0.2×	0.2×
			ILU0	0.06	0.05	0.12	0.2×	0.2×
			AINV	0.5	0.06	0.11	.05×	0.3×
therm1	16.0	0.15	ILU0	0.36	1.17	0.73	15×	22×
			AINV	3.02	1.18	0.63	4×	25×
therm2	78.4	0.42	ILU0	0.72	3.88	1.56	35×	50×
			AINV	5.28	5.51	1.59	11×	49×
therm3	99.6	0.52	ILU0	0.9	5.42	1.82	37×	55×
			AINV	6.5	6.3	2.0	12×	50×
therm4	177.2	0.7	ILU0	1.1	3.37	1.29	74×	137×
			AINV	7.4	10.4	2.51	18×	70×
therm5	554.1	1.45	ILU0	1.62	8.53	2.12	148×	261×
			AINV	11.9	17.0	3.34	36×	165×

5.4.1 Comparison in matrix solving

The performance of GMRES solver with preconditioners is summarized in Table 5.3, where the basic statistics information of circuit examples is described in Table 5.2. For comparison, the same equation $\mathbf{Ax} = \mathbf{b}$ is solved by (1) direct LU method, i.e., LU factorization and triangular solves, (2) GMRES solver using only CPU, and (3) the proposed GMRES solver with GPU parallel computation. The LU method is provided by SuperLU_MT, which runs on our server with 4 threads for the factorization routine `psgstrf`. The GMRES solvers in both CPU and GPU versions can take parameters such as restart number, maximum iteration number, and residual tolerance. In our experiments, these three parameters are set as 32, 10^6, and 10^{-6}, accordingly.

Note that we also test three different preconditioners, such as diagonal (DIAG), incomplete LU (ILU0), and approximate inverse (AINV) in GMRES implementations. Time spent on preconditioner constructions and GMRES solving with preconditioners are all recorded in Table 5.3. As can be observed by the run time measurements, GMRES performs better with the help of preconditioner. Especially for the large matrices, GMRES without preconditioner or with a insufficient preconditioner fails to converge within the given limit of maximum iteration number. These unconverged cases are not shown in the table.

In general, as the problem sizes become larger, GPU GMRES will show more speedup over SuperLU_MT. Table 5.3 shows that the speedup ranges from one to two orders of magnitudes. ILU0 preconditioner usually gives better performance over AINV. Compared to the CPU version of preconditioned GMRES, GPU GMRES delivers about 2–4× speedup for large cases. With larger thermal circuits, more speedups can be expected

as indicated by the table.

Further comments can be made on ILU0 and AINV preconditioners. Both of them show better robustness than the simple DIAG preconditioner, and they succeed on all the demonstrated examples. Their robustness comes from the better knowledge obtained from matrix $\mathbf{A}$ during the preconditioner construction phase. The AINV preconditioner is more expensive than ILU0, since it approximates $\mathbf{A}^{-1}$ and tends to have more non-zero elements than the ILU0 preconditioner, which approximates $\mathbf{A}$. This increased cost of AINV is reflected in both the construction phase and the GMRES iteration phase, as can be seen from Table 5.3.

We have also noticed that for small examples, the GMRES solver does not beat the direct LU method, such as in the example "solid" extracted from 3D-ICE. This is due to the nature of iterative solver and also the characteristic of GPU computing. For small matrices, iterative solvers could have a relatively higher overhead than a direct solver, and this results in longer running time. Meanwhile, CUDA GPU prefers large data throughput to hide data access latency, and it hurts the performance when working on small examples. However, GPU GMRES outperforms SuperLU_MT as soon as the size of test case grows.

It is noteworthy that we use single-precision floating-point representation, i.e., 32 bits, for real numbers in all the calculations. This implementation comes from two reasons. One is because NVIDIA's current generation GPUs support single precision better than double precision, in terms of speed and memory space. The other reason, which is also from the perspective of our application here, is that thermal simulation results, such as temperatures, do not require very high precision. We have compared our single-precision

results with double-precision ones, and the temperature difference between the two is at most 0.1 degree, which is acceptable to most thermal analysis. We will further justify this claim of using single-precision by a temperature waveform comparison in the following part.

5.4.2 Comparison in transient thermal analysis

We now perform the comparison in the transient thermal simulation. As we have seen from Table 5.3, GMRES does not converge to accurate solution without preconditioner or with a diagonal preconditioner. Hence, for the transient simulation, we will only show the runtime measurements for GMRES with incomplete LU and approximate inverse preconditioners.

The construction of preconditioner is a one-time task for the whole transient simulation. This is because the left-hand side matrix $\mathbf{A}$ does not change on the transient time steps. Hence, the preconditioner $\mathbf{M}$ only needs to be calculated once and repeatedly used for several thousand times. So its computing costs can be easily amortized over the later transient computing tasks. As a result, for implicit preconditioner $\mathbf{M}$, we still compute it in the CPU as incomplete LU decomposition is difficult to be parallelized in GPU. We then compare the performance of ILU0 and AINV preconditioners on both CPU and GPU in Table 5.4. The transient temperature response of example "therm5" is shown in Fig. 5.10. Both CPU GMRES and GPU GMRES solvers deliver satisfactory results if compared to the SuperLU waveforms.

All time measurements consider every procedure except the initial file reading of the data ($\mathbf{G}$, $\mathbf{C}$, and $\mathbf{B}$ matrices) into memory. We can make some observations based on Table 5.4. Firstly, GMRES algorithm is very efficient for transient thermal simulation.

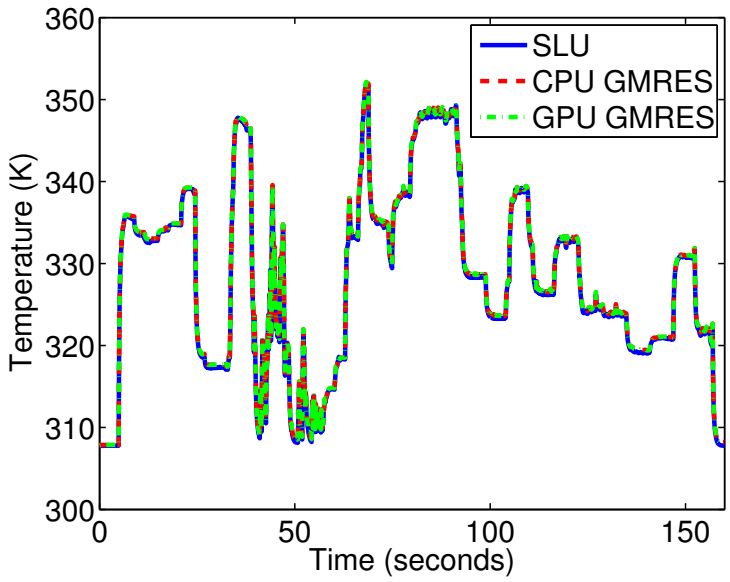


Figure 5.10: Temperature waveform of the 3D IC model "therm5". Solid curve is the result of SuperLU, dashed curve is from CPU GMRES, and dash-dotted curve is from GPU GMRES.

Normally, for transient simulation, direct methods are more efficient than the iterative counterpart because the factorization of the original matrix only needs to be performed for one time. After that, every step only needs to perform the backward and forward substitutions, also called triangular solves, whose computational complexities are much lower than the LU factorization. However, for our transient thermal simulation, the right-hand side $\mathbf{b}$ does not change significantly within a few time steps and usually the new $\mathbf{x}_t$ is quite close to its previous state $\mathbf{x}_{t-1}$. By setting the initial value of each i-th step to be the solution of the previous step, the number of iteration need for each step is fairly small. As we can see from the results for SuperLU and serial GMRES, the serial iterative solver even outperforms the parallel direct solver for some large cases.

Secondly, for GMRES preconditioners on CPU, implicit preconditioners such as

ILU are significantly better than explicit ones such as AINV. However, for preconditioers on GPU, explicit preconditioners have better scalability due to their parallellism nature. We can see that the speedup of the proposed GPU GMRES can be two orders of magnitudes over parallel LU. Again, if compared to the CPU version of preconditioned GMRES, the GPU one can deliver 2–4× speedup for large cases in transient analysis.

Fig. 5.11 shows the thermal profile of the 3D circuit with two active layers, liquid cooling micro-channels, and heat sink with the convective interface. The flow rate inside micro-channels is fast enough to remove heat instantaneously. Thus, the temperature distribution inside micro-channels is almost constant (difference is 0.005 °C) as Fig. 5.12 shows.

5.5 Summary

An efficient finite difference based full-chip simulation method on CPU-GPU platform for 3D ICs with liquid cooling is proposed. Unlike existing fast thermal analysis methods, our new method starts from the basic heat equations to model 3D ICs with inter-tier liquid cooling micro-channels, and directly solves the resulting PDE using iterative GMRES solver. To speed up the analysis process, we further developed a preconditioned GPU-accelerated GMRES solver. We also studied different preconditioners for GPU platforms and compared their performances. Experimental results showed that the proposed method can lead to order of magnitudes speedup over the parallel LU based solver and up to 4× speedup over CPU GMRES for both DC and transient thermal analyses on a number of thermal circuits and other published problems.

Table 5.4: Comparison of solvers in transient analysis of our 3D IC thermal models. SuperLU deploys 4 CPU threads in LU factorization. Time measurements in Column 2 are the sums of LU factorization times and triangular solve times. Runtime in seconds.

1	2	3	4	5	6
circuit name	superLU fact.+sol.	GMRES			Speedup (C2/C5)
		precond	CPU	GPU	
therm1	254.1	ILU0	55.5	34.3	7×
		AINV	100.2	45.8	6×
therm2	829.8	ILU0	154.3	69.5	12×
		AINV	274.2	76.0	11×
therm3	1081.7	ILU0	203.6	75.9	14×
		AINV	357.2	87.4	12×
therm4	1561.8	ILU0	93.4	39.6	40×
		AINV	804.4	209.2	8×
therm5	3463.1	ILU0	154.6	54.9	63×
		AINV	1374.7	227.3	15×

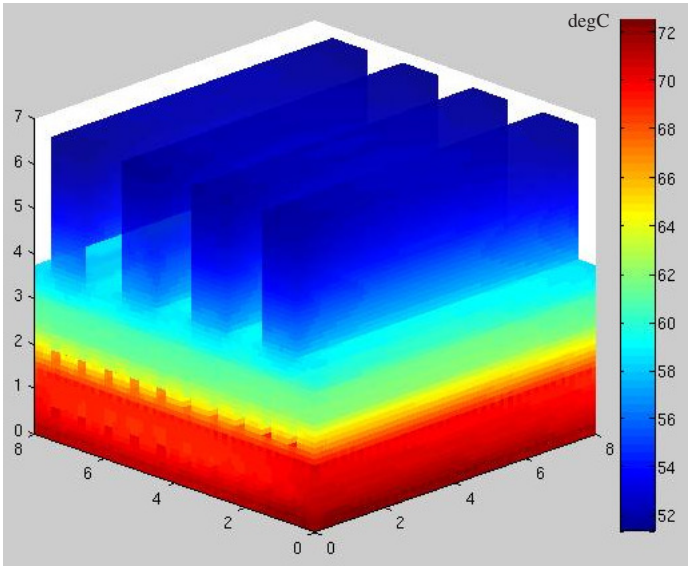


Figure 5.11: Temperature profile of the 3D IC with two active layers.

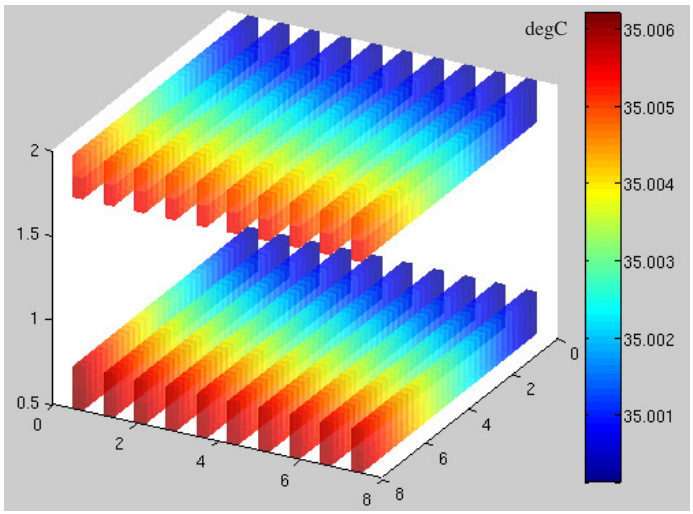


Figure 5.12: Temperature profile inside the coolant channels.

Chapter 6

GPU parallel shooting algorithm for RF/MM ICs

This chapter presents a new shooting-Newton method with GPU-accelerated periodic Arnoldi shooting solver, called GAPAS, for fast periodic steady state analysis of RF/MM ICs. The new method first explores the periodic structure of state matrix by using periodic Arnoldi algorithm, which is a variant of GMRES solver, only a little different from the one in Section 5.3. We show that the resulting implementation of the solver is amenable for GPU parallel computing. CUDA GPU's features, such as coalesced memory access and overlapping transfers with computation, are utilized to further boost the efficiency of our GAPAS. Experimental results from several industrial examples show that when compared to the state-of-the-art implicit GMRES method under the same accuracy, the GPU based shooting-Newton method can bring up to 8 times speedup.

6.1 Introduction

Recent advances in radio-frequency and monolithic microwave integrated circuits (RF/MM IC) operating at 60 GHz [86–89] can provide much higher data rate than today's mobile devices for future smart mobile applications. However, the design of RF/MM-IC front-end circuits at this scale is very challenging [90–96]. At the scale of 60 GHz, all active and passive devices are closely coupled and the resulting post-layout circuit model has a drastically increased complexity. Moreover, in order to follow the high-frequency carrier signal, 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 cannot be compromised due to high precision requirement. Worse, its design complexity increases significantly, 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 [90–95]: finite difference time domain (FDTD) method, harmonic balance (HB) method, and shooting-Newton method. The shooting-Newton method has a better convergence for strongly nonlinear circuits because the underlying transient analysis has an adaptive time step control. However, it involves a Jacobian (sensitivity matrix), which is usually a large-scale dense matrix. GMRES solver with the use of a standard Krylov subspace [66, 97] and an implicit matrix formulation [93] partially alleviates the high computational cost.

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.

GPU parallel computing has been explored recently in many design automation algorithms [98] such as Boolean 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 [52, 99, 100], logic simulation [101], and general SPICE simulation [98, 102], 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 yet.

We propose a GPU accelerated periodic Arnoldi shooting method, GAPAS, for fast periodic steady analysis of RF/MM-IC systems. 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 amenable for parallel computing solutions such as GPU. Secondly, we parallelize the periodic Arnoldi based GMRES solver in the shooting-Newton method on the latest 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 our method. We remark that the existing study in [103] has an initial exploration of the structured Krylov subspace and its parallelization. However, it did not identify the periodic structured Krylov subspace during the shooting-Newton process. This will be discussed later and we further demonstrate the GPU implementation on this parallelism friendly formulation.

Section 6.2 reviews the background of the periodic steady state analysis and shooting-Newton method. Then, we discuss non-structured Krylov subspace based conventional GMRES algorithm with the matrix-free basis generation technique as our baseline. Next, Section 6.3 presents our 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 6.4. Numerical experimental results are demonstrated in Section 6.5, and Section 6.6 closes the chapter.

6.2 Background

In this section, we first review the mathematical description of periodic steady state (PSS) analysis and its solution by shooting-Newton method. Then we discuss the conventional GMRES algorithm based on traditional non-structured Krylov subspace with the matrix-free method.

6.2.1 Review of the shooting-based PSS analysis methods

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,

$$\mathbf{f}(\mathbf{x}(t), t) = \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{q}(\mathbf{x}(t)) + \mathbf{j}(\mathbf{x}(t)) + \mathbf{u}(t) = 0, \tag{6.1}$$

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 constraint

$$\mathbf{x}(T) = \mathbf{x}(0). \tag{6.2}$$

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 state $\mathbf{x}(t)$ 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.* (6.1) *at $t_0 + T$, starting from a guessed initial state $\mathbf{x}(t_0)$ at t_0, or*

$$\mathbf{x}(t_0 + T) = \boldsymbol{\phi}_T(\mathbf{x}(t_0), t_0). \tag{6.3}$$

Using the state transition function, boundary constraint in Eq. (6.2) is reformulated as

$$\boldsymbol{\phi}_T(\mathbf{x}(0), 0) = \mathbf{x}(0) \tag{6.4}$$

for one period with $t_0 = 0$.

Definition 3 *Shooting sensitivity matrix, also called shooting Jacobian, can be defined as*

$$\mathbf{J}_{\boldsymbol{\phi}_T}(\mathbf{x}(0), 0) = \frac{\mathrm{d}\mathbf{x}(T)}{\mathrm{d}\mathbf{x}(0)} = \frac{\mathrm{d}\boldsymbol{\phi}_T(\mathbf{x}(0), 0)}{\mathrm{d}\mathbf{x}(0)}. \tag{6.5}$$

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. (6.6). 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 chapter.

To solve for the periodic steady state $\mathbf{x}_0$ from the two point boundary condition in Eq. (6.4) 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,

$$\mathbf{x}_0^k = \mathbf{x}_0^{k-1} + \left[\mathbf{I} - \mathbf{J}_{\boldsymbol{\phi}_T}(\mathbf{x}_0^{k-1}, 0)\right]^{-1} \left[\boldsymbol{\phi}_T(\mathbf{x}_0^{k-1}, 0) - \mathbf{x}_0^{k-1}\right], \tag{6.6}$$

where $\mathbf{I}$ is the identity matrix and k is the index of Newton iteration. Such a method is called shooting-Newton algorithm [90, 92, 93, 95].

Specifically, $\mathbf{J}_{\boldsymbol{\phi}_T}$ is obtained by the following manner. One first integrates the DAE of Eq. (6.1) 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,

$$\frac{1}{h_j}\left[\mathbf{q}(\mathbf{x}_j) - \mathbf{q}(\mathbf{x}_{j-1})\right] + \mathbf{j}(\mathbf{x}_j) + \mathbf{u}_j = 0, \tag{6.7}$$

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. (6.7) 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

$$\left[\mathbf{G}(\mathbf{x}_j^{l-1}) + \frac{1}{h_j}\mathbf{C}(\mathbf{x}_j^{l-1})\right](\mathbf{x}_j^{l} - \mathbf{x}_j^{l-1}) = -\frac{1}{h_j}\left(\mathbf{q}(\mathbf{x}_j^{l-1}) - \mathbf{q}(\mathbf{x}_{j-1})\right) - \mathbf{j}(\mathbf{x}_j^{l-1}) - \mathbf{u}_j, \tag{6.8}$$

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 in Eq. (6.7) and transient Newton iterations for linearization on each time step in Eq. (6.8) 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 Eq. (6.7) with respect to the initial condition $\mathbf{x}_0$ generates the following relation

$$\left[\mathbf{G}_j + \frac{1}{h_j}\mathbf{C}_j\right]\frac{\mathrm{d}\mathbf{x}_j}{\mathrm{d}\mathbf{x}_0} = \frac{\mathbf{C}_{j-1}}{h_j}\frac{\mathrm{d}\mathbf{x}_{j-1}}{\mathrm{d}\mathbf{x}_0}. \tag{6.9}$$

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

$$\mathbf{J}_{\boldsymbol{\phi}_T} = \prod_{j=1}^{M} \left[\mathbf{G}_j + \frac{1}{h_j}\mathbf{C}_j \right]^{-1} \frac{1}{h_j}\mathbf{C}_{j-1}. \tag{6.10}$$

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-factorized 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 $\mathbf{J}_{\boldsymbol{\phi}_T}$, and the $O(n^3)$ of the direct LU factorization of the matrix in Eq. (6.6).

6.2.2 Traditional GMRES and matrix-free GMRES

Since the shooting Jacobian matrix in the PSS shooting-Newton update equation is dense, 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, iterative solver like GMRES can be applied. The GMRES method is an iterative method for solving a system of large scale linear equations $\mathbf{Ax} = \mathbf{b}$, where, in PSS problem, the left-hand side dense matrix and the right-hand side vector are

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

from the shooting-Newton update in Eq. (6.6).

One can apply the GMRES solver mentioned in Section 5.3 and Algorithm 8 to solve the PSS solution. The only exception here is that we don't have preconditioner as we

had in the thermal simulation. This is because preconditioner requires knowledge of the left-hand side matrix $\mathbf{A}$, which is not explicitly formed in our shooting method due to the use of a more efficient matrix-free method we are going to talk about next. Therefore, all the occurrences of preconditioner $\mathbf{M}$ in Algorithm 8 and its underlying Krylov subspace can be assumed to be identity matrix $\mathbf{I}$.

Algorithm 9 Matrix-free "matrix"-vector multiplication to replace $\mathbf{Av}$ in Line 4 in Algorithm 8.

Input: Previous basis $\mathbf{v}^i$ and prefactorized matrices of (6.11)

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: $\quad \mathbf{w} = \left(\mathbf{G}_j + \frac{1}{h_j}\mathbf{C}_j\right)^{-1} \frac{1}{h_j}\mathbf{C}_{j-1}\mathbf{w}$

4: **end for**

5: $\mathbf{w} = \mathbf{v}^i - \mathbf{w}$

To further reduce the computational cost, one needs to avoid the explicit formulation of matrix $\mathbf{A}$, since forming $\mathbf{A}$ involves $O(n^3M)$ floating point operations and multiplying $\mathbf{Av}^i$ needs another $O(n^2)$ operations for each Arnoldi iteration. The work in [93,95] introduced a matrix-free approach to directly calculate the matrix-vector products (MVP) $\mathbf{Av}^i$ without forming $\mathbf{A}$. This is realized in Algorithm 9. 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

$$\mathbf{G}_j + \frac{1}{h_j}\mathbf{C}_j, \quad j = 1, \ldots, M. \tag{6.11}$$

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 9.

However, as discussed in the later part of this chapter, neither the standard GMRES procedure in Algorithm 8 nor the matrix-free GMRES discussed here 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 [104].

6.3 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.

6.3.1 Periodic Krylov subspaces

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. (6.10)

becomes

$$\mathbf{J}_{\phi_T} = \mathbf{A}_M \mathbf{A}_{M-1} \cdots \mathbf{A}_1. \tag{6.12}$$

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

$$\mathcal{J} = \begin{bmatrix} 0 & & & \mathbf{A}_1 \\ \mathbf{A}_2 & \ddots & & \\ & \ddots & & \\ & & \mathbf{A}_M & 0 \end{bmatrix}, \tag{6.13}$$

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*

$$\mathcal{J}\mathcal{V}^m = \mathcal{V}^{m+1}\mathcal{H}^m, \tag{6.14}$$

where

$$\mathcal{V}^m = \mathbf{V}_1^m \oplus \mathbf{V}_2^m \cdots \oplus \mathbf{V}_M^m, \tag{6.15}$$

and

$$\mathcal{H}^m = \begin{bmatrix} 0 & & & \mathbf{H}_1^m \\ \mathbf{H}_2^m & \ddots & & \\ & \ddots & & \\ & & \mathbf{H}_M^m & 0 \end{bmatrix}. \tag{6.16}$$

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. (6.14), $\mathbf{V}_j^m \in \mathbb{R}^{n\times m}$ matrices are subspace bases on the corresponding 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.,

$$\mathbf{A}_j \mathbf{V}_{j\ominus 1}^m = \mathbf{V}_j^{m+1} \mathbf{H}_j^m, \tag{6.17}$$

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 [105] 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.

The primary observation we have here is to illustrate that the underlying Krylov subspace of shooting-Newton implicitly possesses the periodic structure as shown in Eq. (6.13). 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 Eq. (6.13).

Take $m = 4$ and $p = 3$ for example [103], in the traditional GMRES, the solution is

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

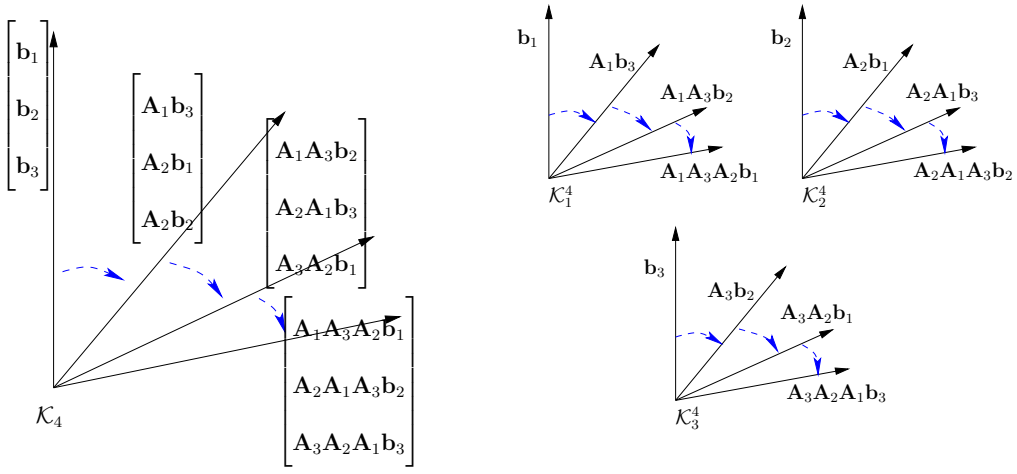
while the p-periodic-block-structured method has the solution

$$\mathbf{x}^4 = \left[\begin{array}{l} \mathbf{x}_1^4 \\ \mathbf{x}_2^4 \\ \mathbf{x}_3^4 \end{array} \right] \left| \begin{array}{l} \mathbf{x}_1^4 \in \mathcal{K}_1^4 = \text{span}(\mathbf{b}_1, \mathbf{A}_1\mathbf{b}_3, \mathbf{A}_1\mathbf{A}_3\mathbf{b}_2, \mathbf{A}_1\mathbf{A}_3\mathbf{A}_2\mathbf{b}_1) \\ \mathbf{x}_2^4 \in \mathcal{K}_2^4 = \text{span}(\mathbf{b}_2, \mathbf{A}_2\mathbf{b}_1, \mathbf{A}_2\mathbf{A}_1\mathbf{b}_3, \mathbf{A}_2\mathbf{A}_1\mathbf{A}_3\mathbf{b}_2) \\ \mathbf{x}_3^4 \in \mathcal{K}_3^4 = \text{span}(\mathbf{b}_3, \mathbf{A}_3\mathbf{b}_2, \mathbf{A}_3\mathbf{A}_2\mathbf{b}_1, \mathbf{A}_3\mathbf{A}_2\mathbf{A}_1\mathbf{b}_3) \end{array} \right. .$$

Their differences can be clearly illustrated by Fig. 6.1. The independency of Krylov subspaces constructed on different time step segments is obviously revealed. In the sequel, 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.

6.3.2 GMRES with periodic Krylov subspaces

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 in the cyclic structured GMRES, which cost a high memory overhead. To reduce this M-cyclic system,



(a) Krylov subspace in traditional GMRES.

(b) Krylov subspace in p-cyclic GMRES.

Figure 6.1: The comparison of standard Krylov subspace and p-periodic-block-structured Krylov subspace. Order $m = 4$, and $p = 3$ time steps.

[103] proposed block Gaussian elimination to derive a p-cyclic system, where $p \ll M$. The time point indices $(1, \ldots, M)$ in one cycle are partitioned as

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

where $q_p = M$, and the selection of q_j, $j = 1, \ldots, p$, usually makes 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. We follow the reduction scheme as described in [103], with only changes to employ matrix-free basis vector computation inspired by [93]. Our algorithm is summarized in Algorithm 10. The krylov subspaces constructed inside this algorithm still satisfy the relationship stated in Eq. (6.17), except that the number of blocks is reduced from M to p, and hence the definition of $j \ominus 1$ is

changed to

$$j \ominus 1 = \begin{cases} j-1, & \text{if } j = 2, \ldots, p, \\ p, & \text{if } j = 1. \end{cases} \tag{6.18}$$

We call the procedure in Algorithm 10 as periodic Arnoldi shooting (PAS) based GMRES. In this algorithm, the subscript j denotes the index of the periodic blocks, $j = 1, \ldots, p$, 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 9 and Line 18, 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. (6.15) and Eq. (6.16).

By employing the Arnoldi decomposition in Eq. (6.14), the generated subspace bases $\mathbf{V}_j^m$ and Hessenberg matrices $\mathbf{H}_j^m$ from Algorithm 10 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.,

$$\begin{bmatrix} \tilde{\mathbf{I}} & & & -\mathbf{H}_1^m \\ -\mathbf{H}_2^m & \tilde{\mathbf{I}} & & \\ & \ddots & \ddots & \\ & & -\mathbf{H}_p^m & \tilde{\mathbf{I}} \end{bmatrix} \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_p \end{bmatrix} \simeq \begin{bmatrix} \delta_1 \mathbf{e}_1 \\ \delta_2 \mathbf{e}_1 \\ \vdots \\ \delta_p \mathbf{e}_1 \end{bmatrix}, \tag{6.19}$$

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

Similar to the concept of matrix-free in Algorithm 9, 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-factorized sparse

Algorithm 10 GPU accelerated periodic Arnoldi method

Input: Sparse LU factors of matrices $\mathbf{G}_j + \frac{1}{h_j}\mathbf{C}_j$, capacitance matrices $\mathbf{C}_j$, vectors $\mathbf{b}_j$, and step length h_j, $j = 1, \ldots, p$.

Output: Updated shooting result, $\mathbf{x} = [\mathbf{x}_1, \cdots, \mathbf{x}_p]$.

for $j = 1$ to p **do** *// parallel in GPU*

  $\hat{\mathbf{v}}_j = [\mathbf{G}_{q_{j-1}+1} + \frac{1}{h_{q_{j-1}+1}}\mathbf{C}_{q_{j-1}+1}]^{-1}\mathbf{b}_{q_{j-1}+1}$

  for $k = q_{j-1} + 2$ to q_j **do**

    $\hat{\mathbf{v}}_j = [\mathbf{G}_k + \frac{1}{h_k}\mathbf{C}_k]^{-1}\frac{1}{h_k}\mathbf{C}_{k-1}\hat{\mathbf{v}}_j + \mathbf{b}_k$

  end for

  $[\mathbf{v}_j^1, \text{dummy}] = \text{orth}(\hat{\mathbf{v}}_j, [\,])$

  $\mathbf{V}_j^1 = [\mathbf{v}_j^1]$, and $\delta_j = \langle \mathbf{v}_j^1, \mathbf{b}_j \rangle$ *// First basis vector.*

end for

for $i = 1$ to m **do** *// Arnoldi iteration: serial loop*

  for $j = 1$ to p **do** *// parallel in GPU*

    $\hat{\mathbf{v}}_j = \mathbf{v}_{j\ominus 1}^i$ *// $\ominus$ is defined in Eq. (6.18).*

    for $k = q_{j-1} + 2$ to q_j **do**

      $\hat{\mathbf{v}}_j = [\mathbf{G}_k + \frac{1}{h_k}\mathbf{C}_k]^{-1}\frac{1}{h_k}\mathbf{C}_{k-1}\hat{\mathbf{v}}_j$

    end for

    $[\mathbf{v}_j^{i+1}, \mathbf{H}_j(1\!:\!i+1, i)] = \text{orth}(\hat{\mathbf{v}}_j, \mathbf{V}_j^i)$

    $\mathbf{V}_j^{i+1} = \begin{bmatrix} \mathbf{V}_j^i & \mathbf{v}_j^{i+1} \end{bmatrix}$ *// New basis vector.*

  end for

end for

Solve least squares problem in Eq. (6.19), and then calculate the shooting result $\mathbf{x}$

LU factors of $\mathbf{G}_j + \mathbf{C}_j/h_j$ in triangular solves, shown in Line 3 to Line 5 and Line 12 to Line 14 in Algorithm 10. Then, these newly computed vectors go through a series of orthogonalization with respect to their predecessors in the corresponding subspaces. 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 10, 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 8 and Algorithm 9 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.

6.3.3 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 10, 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 one signal cycle. As the subspaces grow larger, the orthogonalization step also needs BLAS level-1 operations. Even though matrix-free formulation is applied, the new basis vector generation still requires M/p forward eliminations and backward substitutions (two triangular solves) 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 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 19 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.

6.4 Parallelism strategy on GPUs for p-cyclic GMRES method

In this section, we will present the implementation details of our 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. 6.2, where the shooting iteration is on top of one period simulation of traditional transient analysis, 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 [106].

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) good thread block configuration 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.

We begin our discussion from the parallelism strategy on GPU platforms for the major computing elements in the Arnoldi iteration shown in Algorithm 10. 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 the blocks of the same `blockIdx`, different rows will be computed in parallel by multiple threads, which is similar to [107]. Next, the resulted product vectors go through the sparse triangular

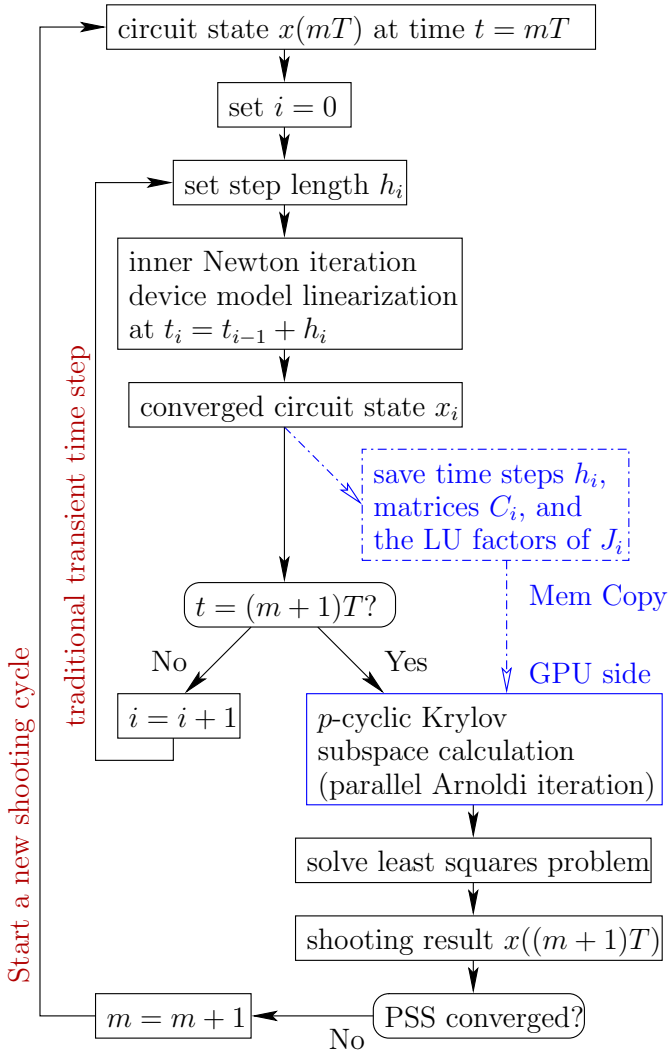


Figure 6.2: The flow of shooting-Newton update with p-cyclic GMRES solver on GPU side.

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 much 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 [82]. 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 nature and fairly regular memory access pattern, GPU usually can almost attain its peak performance.

6.4.1 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 are very amenable for GPU, 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. Using the same strategy in Section 5.3.2, the coefficient matrix $\mathbf{A}$ should be transferred to GPU memory once and used repeatedly during Arnoldi iteration.

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 is highly parallel. This can be easily observed by inspecting the for-loop in Algorithm 10. In a serial CPU computation, it iterates through all the p time step segments, 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 step segments $j = 1, \ldots, p$ are independent. Therefore the for-loop between Line 10 and Line 17 are implemented in parallel on GPU in our parallel version, as illustrated in Fig. 6.4. Although the for-loop at the beginning of the algorithm is a one-time job and it 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 segment, 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 indices

$j = 1, \ldots, p$. This is shown in Fig. 6.5, 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 completed, 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.

6.4.2 Optimization of memory accesses 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 step segments 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 beginning of each loop. The symbol $\ominus$ is defined in Eq. (6.18). 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

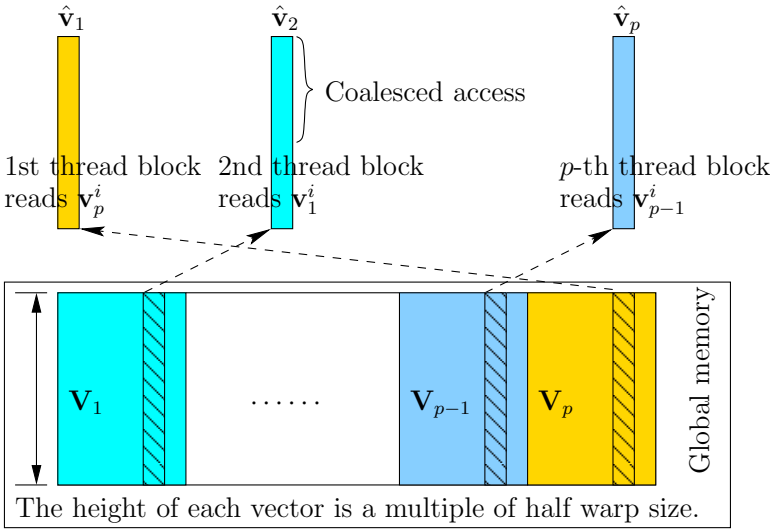


Figure 6.3: Parallel p-cyclic basis vector copy using GPU thread blocks. (Line 11 in Algorithm 10)

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 step segments (blocks) have taken care of this issue, hence the vector copy can be carried out in a coalesced fashion. To be more specifically, 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 [103] that the Krylov subspace construction, i.e., the for-loop from $i = 1$ to m, is the most expensive part of Algorithm 10. 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 step segments 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 11 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. 6.3.

In GAPAS, solving the least squares problem in Eq. (6.19) is done in CPU as we have to check the results from p parallel 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 squares problem is formulated and solved by LAPACK routine `gels` on CPU.

6.5 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

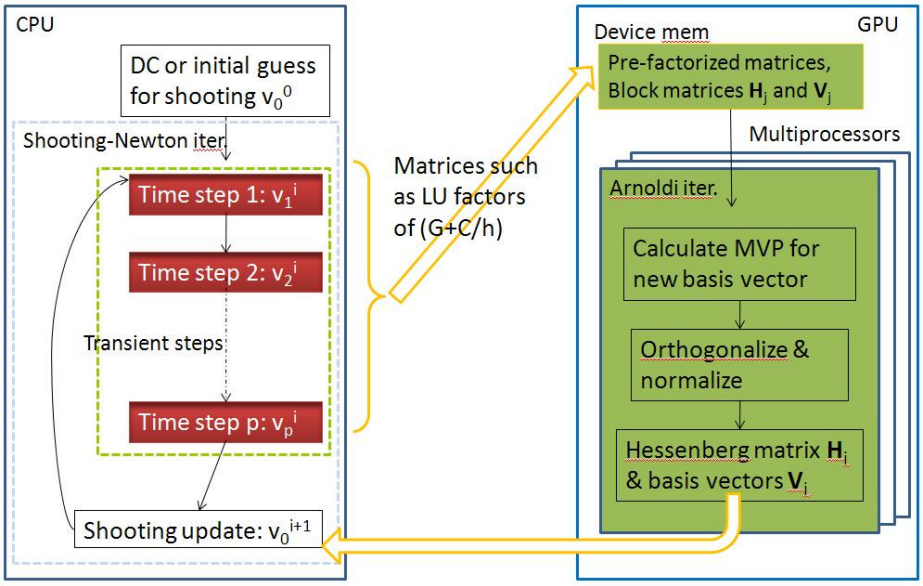


Figure 6.4: CPU-GPU collaboration inside a shooting cycle.

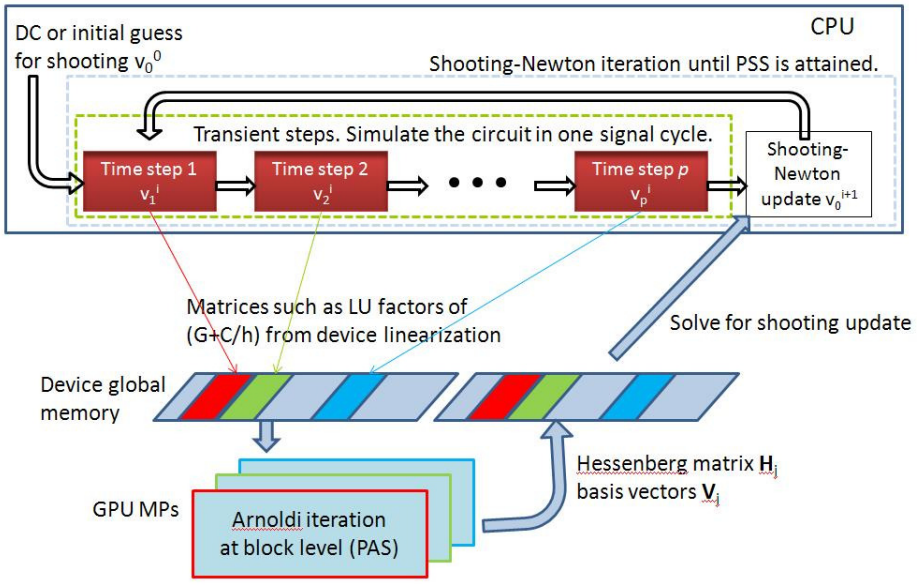


Figure 6.5: Parallel computation of Krylov-subspaces on GPU.

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 (with 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 5 GB global memory and 144 GB/sec memory bandwidth. For the CPU part, the server has a Quad-Core Intel E5504 CPU and 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 [108], with a few modifications to comply with SPICE data structures. The matrix-free GMRES is implemented exactly following the procedure described in [93,95], 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 runtime 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. 6.6 and Fig. 6.7. We increase the complexity by adding extracted parasitic extraction, i.e., the substrate mesh.

Table 6.1: 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.

		1	2	3		4			
circuit	eqn #	direct LU	explicit-GMRES	MF-GMRES		PAS-GMRES		GAPAS	
		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	9.06	1.09	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	4.24	4.8×

Table 6.1 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 speedup over its serial counterpart.

We first show that GAPAS' efficiency on GPU. We can observe from Table 6.1 that, due to the optimized implementation of MVM and the use of the structured PAS, the performance of shooting-GMRES can be 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, PAS-GMRES is better than the traditional MF-GRMES in term of running time given the same accuracy, as can be judged from Table 6.1. Also note that the explicit formulation of matrix in GMRES leads to the similar runtime to direct LU, since the cost of forming the matrix $\mathbf{A}$ explicitly will dominate 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 10 earlier, the two computationally intensive parts

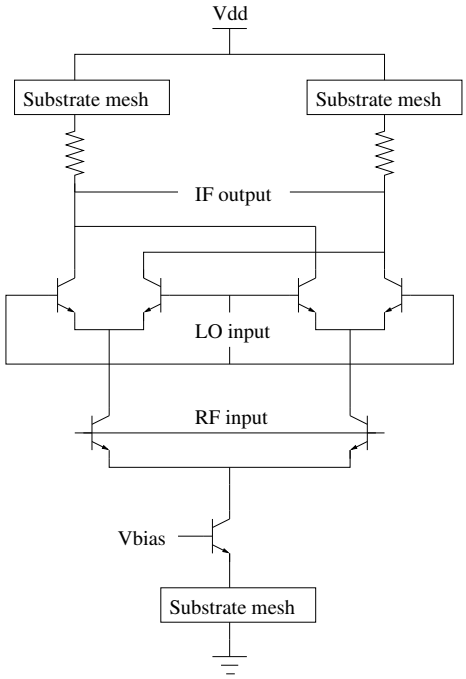


Figure 6.6: A double-balanced BJT mixer.

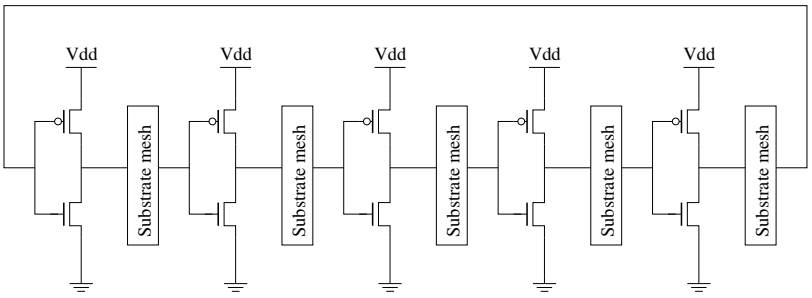


Figure 6.7: A CMOS ring oscillator.

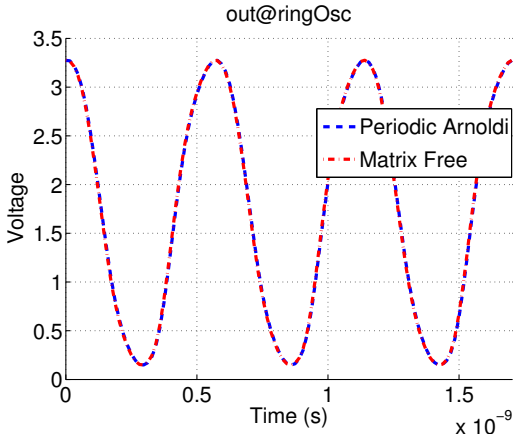


Figure 6.8: The PSS waveform accuracy comparison at the output node of a CMOS ring oscillator. The red line is the non-structured MF-GMRES, and the blue line is the structured PAS-GMRES.

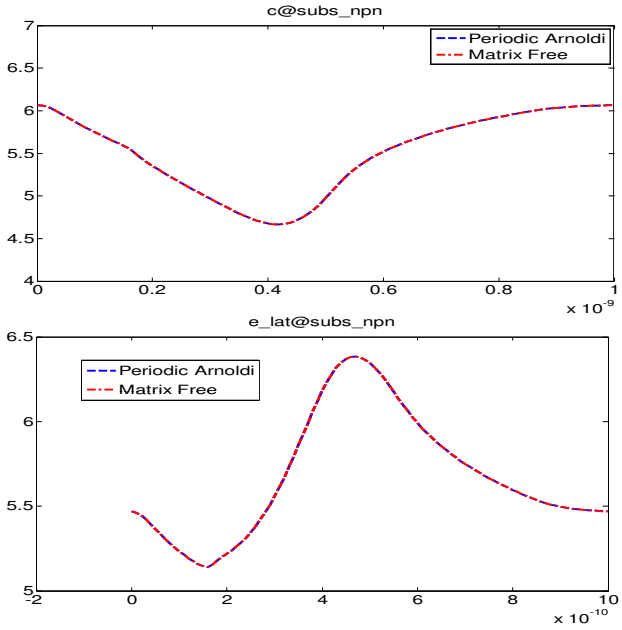


Figure 6.9: The PSS waveform accuracy comparison at two nodes of a BJT mixer. The red line is the non-structured MF-GMRES, and the blue line is the structured PAS-GMRES.

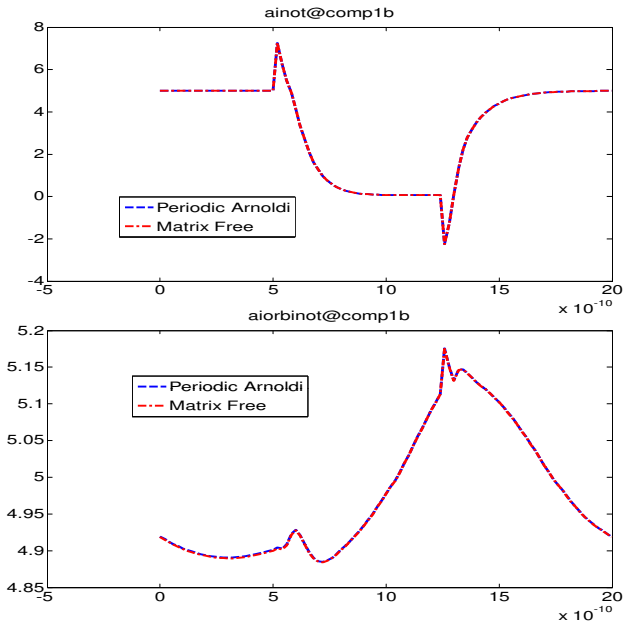


Figure 6.10: The PSS waveform accuracy comparison at two nodes of a CMOS switch cap. The red line is the non-structured MF-GMRES, and the blue line is the structured PAS-GMRES.

Table 6.2: 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	parallel part Arnoldi iteration time (seconds)	sequential part least squares time (seconds)	total time (seconds)
25	4.08	0.01	4.10
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

in GAPAS 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 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 resulted least squares problem is not too big to solve. Table 6.2 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 6.2 suggests that $p = 100$ is the best choice which reconciles well the two contradicting procedures. Note that the time measurements here only take into account the pure CPU or GPU execution time. All other expenses are not included, since we want to emphasize the importance of GPU task balance. The time measurements in Table 6.1 include all time spent in the solver, such as memory allocation and copy and other flow control overheads. The time of host/device memory copy is one concern in GPU program, 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. 6.8 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. 6.9 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. 6.11, we further study the running time scalability by increasing the size of parasitic extraction. The example used is a CMOS DC converter with switching frequency of 1 MHz. The circuit complexity is increased from 200 to 1600 states. The running time is measured for both 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. We note that the benchmark circuits used here may not be large enough if compared to some huge circuits in post-silicon verification. This is partly due to the memory limitation of current GPU cards. With future releases of more advance GPU cards, we expect to test larger examples.

6.6 Summary

A structured shooting algorithm that can fully exploit parallelism in periodic steady state (PSS) analysis has been discussed. Our new algorithm, GAPAS, first explores a periodic structure of the state matrix by using a periodic Arnoldi algorithm for computing the resulting structured Krylov subspace in GMRES solver. We showed that the resulting

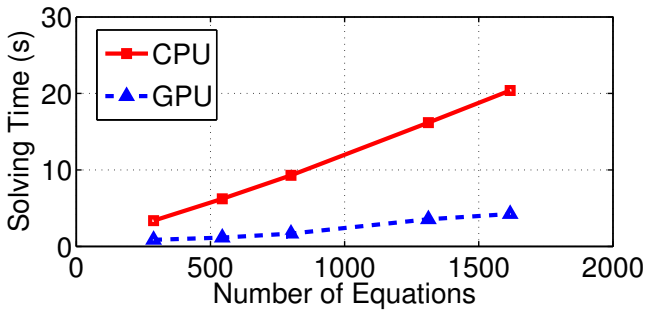


Figure 6.11: The running time scalability comparison for a CMOS DC converter with increased sizes of parasitic. The red line is the scale of running time of the CPU serial version of PAS GMRES, and the blue line is the scale of running time of our GPU parallel GAPAS.

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 latest NVIDIA Tesla GPU platform. We explored the coalesced memory access and overlapping memory transfer with computing to further boost the efficiency of the GAPAS method. Experiment results on a number of radio frequency and microwave circuits have showed that GAPAS can lead up to 8× runtime speedup over its sequential CPU version.

Chapter 7

GPU envelope-following method for power converter simulation

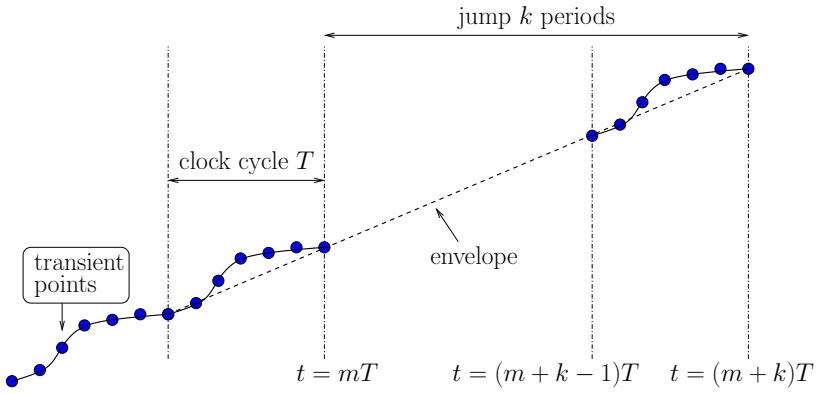
In this chapter, we will propose a new envelope-following parallel transient analysis method for general switching power converters. Similar to shooting problem in Chapter 6, there is a Newton update in envelope-following algorithm, and its solving is also the most computational expensive one in the flow. We employ GMRES for this task, and apply the matrix-free Krylov basis generation technique, which was previously used for RF simulation. The GMRES is implemented on GPU for its data parallel feature. In addition, our new method uses a more robust integration scheme, Gear-2, to compute the sensitivity matrix instead of traditional integration methods. Experimental results from several integrated on-chip power converters show that the proposed GPU envelope-following algorithm leads to about 10× speedup compared to its CPU counterpart, and 100× faster than the traditional envelope-following methods while still keeps the similar accuracy.

7.1 Introduction

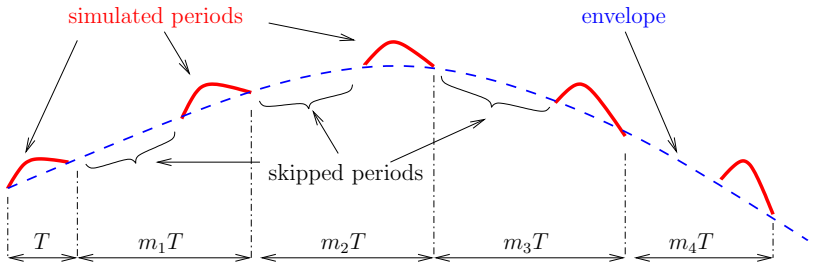
Over the past decades, power electronics and especially switching power converters have seen a surge of new trends and novel applications — from the growing significance of pulse-width modulation (PWM) rectifiers and multi-level inverters to their widespread use in renewable (like solar and wind) energy systems, smart grids and emerging electric and hybrid vehicles [109].

This requires more efficient simulation techniques for power electronics to meet the new application and demanding design constraints. To facilitate the design of typical power electronics circuits, many special-purpose simulation algorithms and tools were developed. Among them is the envelope-following method [110–112], which is able to calculate the slowly changing contour, or envelope, of a carrier waveform with a much higher switching frequency. This is the case for switching power converters, which have fast switching currents to convert powers from one voltage level to another. In these switching power converters, it is the envelope, which is the power voltage delivered, not the fast switching waves in every cycle, that is of interest to designers. As shown in Fig. 7.1(a), the solid line is the waveform of the output node in a Buck converter [113], the dots are the simulation points of SPICE, and the appended dash line is the envelope.

Obviously, traditional SPICE will incur an extraordinarily high simulation time for this task, since it has to integrate the circuit's differential equation with many time points in every clock cycle to get the carrier waveform. However, we don't care about the accurate details of this waveform, since its envelope suffices our appetite. For switching power converters, the waveform of the carrier in consequent cycles does not change



(a) Illustration of one envelope skip.



(b) The envelope changes in a slow time scale.

Figure 7.1: Transient envelope-following analysis. (Both two figures reflect backward-Euler style envelope-following.)

much, envelope-following method is an approximation analysis method, which skips over several cycles (the dash line in Fig. 7.1(b)), the so called envelope step, without simulating them, and then carries out a correction, which usually contains a sensitivity-based Newton iteration or shooting until convergence, in order to begin the next envelope step.

Also, iterative GMRES solver is typically used in the envelope-following method to compute the solution of Newton update due to its efficiency compared to direct LU method. However, as the Jacobian matrix or the sensitivity matrix in the equation to be solved is dense, explicit computing of the Jacobian is a very expensive process. Matrix-free GMRES proposed in [93] for RF shooting simulation can be used here to ameliorate the situation, as we did in Chapter 6. Motivated by the data parallel power of modern GPU, we propose a new parallel envelope-following method for the transient analysis of switching power converter circuits. The main contributions in his work include: (1) Parallelization of the Newton update, which is the most time consuming step, in the envelope-following method, on GPU platforms. (2) Development of the new GMRES solver using matrix-free Jacobian-vector multiplication and the Gear-2 integration for sensitivity based Newton update equation.

Next, the basic algorithm of envelope-following will be briefly reviewed and Newton update equation is derived in Section 7.2. Then Section 7.3 presents the CPU parallelization of matrix-free GMRES with Gear-2 integration. Numerical examples are shown in Section 7.4, and finally, this chapter is summarized in Section 7.5.

7.2 Review of envelope-following method

In transient analysis, a nonlinear circuit's behavior can be represented by a coupled set of nonlinear first-order differential algebraic equations (DAE) of the form

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{q}(\mathbf{x}(t)) + \mathbf{f}(\mathbf{x}(t)) = \mathbf{b}(t), \tag{7.1}$$

where $\mathbf{x}(t) \in \mathbb{R}^N$ is the vector of circuit variables, usually comprising node voltages and possibly branch currents, $\mathbf{f}(\cdot)$ is a nonlinear function that maps $\mathbf{x}(t)$ to a vector of N entries most of which are sums of resistive currents at a node, $\mathbf{q}(\cdot)$ maps $\mathbf{x}(t)$ to a vector of N entries of capacitor charges or inductor fluxes, and $\mathbf{b}(t)$ contains the input source values. For many power electronics circuits, the input switching signal is known and is periodic with clock period T, or called signal cycle.

Assume the time inside one period T has been discretised into M time steps, $0 = t_0 < t_1 < t_2 < \cdots < t_M = T$, and the i-th step length is $h_i = t_i - t_{i-1}$. In practice, the discretisation is non-uniform to control truncation error and convergence. Then, given an initial condition $\mathbf{x}(0)$ at $t = 0$, numerical integration is applied to find the time domain solution of circuit state $\mathbf{x}(t)$ at each time step till $t = T$.

For those circuits whose carrier waveforms vary slowly in a large number of periods, the envelope-following method only integrates the DAE in several selected periods and then jumps over to a new time point. By repeating this "simulate and skip" action, envelope-following method attains its efficiency compared to conventional transient analysis, but still can accurately obtain the slow varying envelope.

For example, if the clock period is T, and the suitable jump interval for the envelope is k periods, then the envelope step is kT. Suppose the state at time $t = mT$ is known

as $\mathbf{x}(mT)$, the goal of envelope-following is to obtain the state at the next envelope step, $\mathbf{x}((m+k)T)$. Therefore, envelope-following will lead to significant saving in simulation time if envelope steps are much larger than the clock period, i.e., k is much bigger than one.

To estimate $\mathbf{x}((m+k)T)$, a forward-Euler style jumping relies only on $\mathbf{x}(mT)$ and $\mathbf{x}((m-1)T)$, i.e.,

$$\mathbf{x}((m+k)T) = \mathbf{x}(mT) + k\left[\mathbf{x}(mT) - \mathbf{x}((m-1)T)\right].$$

However, this approach is inefficient due to its restriction on envelope step k, since larger k usually causes instability. Instead, backward-Euler jumping,

$$\mathbf{x}((m+k)T) - \mathbf{x}(mT) = k\left[\mathbf{x}((m+k)T) - \mathbf{x}((m+k-1)T)\right],$$

allows larger envelope steps. Here $\mathbf{x}((m+k-1)T)$ is the unknown variable to be solved by Newton iteration, and $\mathbf{x}((m+k)T)$ is dependent on $\mathbf{x}((m+k-1)T)$ in each iteration. The Newton update equation in each iteration is thus expressed as

$$\Delta\mathbf{x}((m+k-1)T) = \mathbf{A}^{-1}\left[(k-1)\mathbf{x}^j((m+k)T) - k\mathbf{x}^j((m+k-1)T) + \mathbf{x}(mT)\right], \quad (7.2)$$

where the Jacobian matrix A is computed as a combination of circuit sensitivity matrix and identity matrix, as

$$\mathbf{A} = (k-1)\frac{\mathrm{d}\mathbf{x}((m+k)T)}{\mathrm{d}\mathbf{x}((m+k-1)T)} - k\mathbf{I} = (k-1)\mathbf{J} - k\mathbf{I}. \quad (7.3)$$

In each Newton iteration, $\mathbf{x}((m+k-1)T)$ is used as initial condition to calculate $\mathbf{x}((m+k)T)$ by integrating the DAE in one clock period. Meanwhile, the conductance matrix $\mathbf{G}_i = \mathrm{d}\mathbf{f}(\mathbf{x}(t_i))/\,\mathrm{d}\mathbf{x}(t_i)$ and the capacitance matrix $\mathbf{C}_i = \mathrm{d}\mathbf{q}(\mathbf{x}(t_i))/\,\mathrm{d}\mathbf{x}(t_i)$ at each time step are used to derive the sensitivity matrix $\mathbf{J} = \mathrm{d}\mathbf{x}((m+k)T)/\,\mathrm{d}\mathbf{x}((m+k-1)T)$.

Different integration rules can be applied to the computation of sensitivity matrix. It can be easily derived that, if the DAE is integrated with backward-Euler rule, the sensitivity is

$$\mathbf{J} = \frac{\mathrm{d}\mathbf{x}_M}{\mathrm{d}\mathbf{x}_0} = \prod_{i=1}^{M} \left(\frac{1}{h_i}\mathbf{C}_i + \mathbf{G}_i \right)^{-1} \frac{1}{h_i}\mathbf{C}_{i-1}.$$

In short, the envelope-following method is fundamentally a boosted version of traditional transient analysis, with certain skips over several periods and a Newton iteration to update or correct the errors brought by the skips, as is exhibited by Fig. 7.2.

7.3 New parallel envelope-following method

In this section, we explain how to efficiently use matrix-free GMRES to solve the Newton update problem with implicit sensitivity calculation, i.e., the steps enclosed by the double dashed block in Fig. 7.2. Then implementation issues of GPU acceleration will be discussed in detail. Finally, the Gear-2 integration is briefly introduced.

7.3.1 GMRES Solver for Newton update equation

Both the 3D IC thermal simulation and shooting method in previous chapters used GMRES solver. (Refer to Section 5.3 and Section 6.2.2.) Again, as matrix-free technique does not explicitly formulate a coefficient matrix, we cannot build preconditioner either here. Hence, all the appearances of preconditioner matrix $\mathbf{M}$ mentioned in Algorithm 8 are assumed to be identity matrix in the envelope-following case.

At a first glance, the cost of using GMRES directly to solve the Newton update in Eq. (7.2) seems to come mainly from two parts: the formulation of the sensitivity matrix

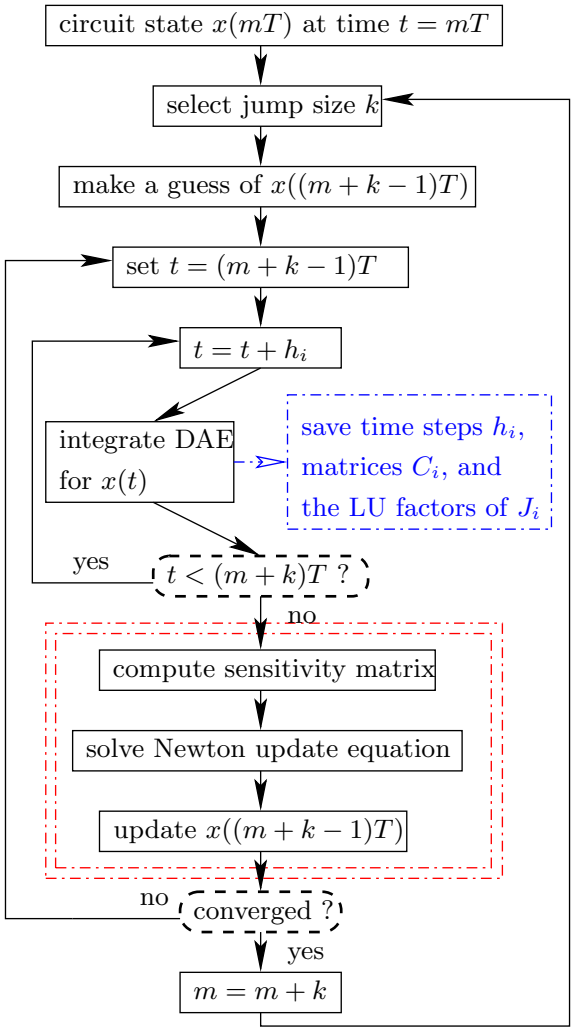


Figure 7.2: The flow of envelope-following method.

$\mathbf{J} = \mathrm{d}\mathbf{x}_M / \mathrm{d}\mathbf{x}_0$ in Eq. (7.9) in Section 7.3.2, and the Arnoldi iteration, especially the matrix-vector multiplication and the orthonormal basis construction (Line 4 through Line 9 in Algorithm 8). In fact, only the matrix-vector product is required in GMRES. Based on this reason, the work in [93] introduces an efficient matrix-free algorithm in the shooting-Newton method, where the equation solving part also involves with a sensitivity matrix. The matrix-free method does not take an explicit matrix as input, but directly passes the saved capacitance matrices $\mathbf{C}_i$ and the LU factorizations of $\mathbf{J}_i$, $i = 0, \ldots, M$, into the Arnoldi iteration for Krylov subspace generation. Therefore, it avoids the cost of forming the dense sensitivity matrix. Briefly speaking, Line 4 will be replaced by a procedure without explict $\mathbf{A}$, and we will talk about the flow of matrix-free generation of new basis vectors in later sections.

There exist many GPU-friendly computing operations in GMRES, such as the vector addition (`axpy`), 2-norm of vector (`nrm2`), and sparse matrix-vector multiplication (`csrmv`), which have been parallelized in the CUDA Basic Linear Algebra Subroutine Library (CUBLAS) and the CUDA Sparse Linear Algebra Library (CUSPARSE) [55].

GPU programming is typically limited by the data transfer bandwidth as GPU favors computationally intensive algorithms [13]. So how to efficiently transfer the data and wisely partition the data between CPU memory and GPU memory is crucial for GPU programming. For these issues, we still use the same strategy mentioned in Section 5.3.2, and the flow of GPU GMRES solver is similar to Fig. 5.8.

As noted in Section 7.2, envelope-following method requires the matrices gathered from all time steps in a period in order to solve a Newton update. At each time step,

SPICE has to linearize device models, stamp matrix elements, and solve circuit equations in its inner Newton iteration. When convergence is attained, circuit states are saved and then next time step begins. This is also the time when we store the needed matrices for envelope-following computation. Since these data are involved in the calculation of Gear-2 sensitivity matrix in the generation of Krylov subspace vectors in Algorithm 11, it is desirable that all of these matrices are transferred to GPU for its data parallel capability.

To efficient transfer those large data, we explores asynchronous memory copy between host and device in the recent GPUs (Fermi architecture), so that the copying overlaps with the host's computing of the next time step's circuit state. The implementation of asynchronous matrices copy includes two parts: allocating page-locked memory, also known as pinned memory, where we save matrices for one time step, and using asynchronous memory transfer to copy these data to GPU memory. While it is known that page-locked host memory is a scarce resource and should not be overused, the demand of memory size of the data generated at one time step can be generously accommodated by today's mainstream server configurations.

7.3.2 Gear-2 based sensitivity calculation

The Gear-2 integration method is a backward differentiation formula (BDF), which approximates the derivative of a function using information from past few steps. Gear-2 method has been shown to be more suitable for many practical problems such as stiff problems where circuit behavior is affected by different time constants (fast ones with large poles and slow ones with small poles) [30]. Switching power converters and RF systems are typically stiff systems as waveforms changing in two different time scales are involved.

Given the DAE in Eq. (7.1), at the i-th time step, Gear-2 approximates the derivative $\mathrm{d}\mathbf{q}(\mathbf{x}(t))/\,\mathrm{d}t$ by a two-step backward finite difference,

$$\frac{\mathrm{d}\mathbf{q}(\mathbf{x}(t_i))}{\mathrm{d}t} = \frac{1}{h_i}\left[\mathbf{q}(\mathbf{x}(t_i)) - \alpha_1^i \mathbf{q}(\mathbf{x}(t_{i-1})) - \alpha_2^i \mathbf{q}(\mathbf{x}(t_{i-2}))\right],$$

where the coefficients α's are chosen to satisfy Gear's backward differentiation formula [114]. For uniformly discretised time steps, the index i in h_i, α_1^i and α_2^i can be omitted.

For the first step t_1, only the initial condition at t_0 is available. Therefore backward-Euler is used, i.e.,

$$\frac{1}{h_1}\left[\mathbf{q}(\mathbf{x}_1) - \mathbf{q}(\mathbf{x}_0)\right] + \mathbf{f}(\mathbf{x}_1) = \mathbf{b}_1. \tag{7.4}$$

Since $\mathbf{x}_0$ directly affects the solution of $\mathbf{x}_1$, the sensitivity matrix up to now is

$$\frac{\mathrm{d}\mathbf{x}_1}{\mathrm{d}\mathbf{x}_0} = \left[\frac{1}{h_1}\mathbf{C}_1 + \mathbf{G}_1\right]^{-1}\frac{\mathbf{C}_0}{h_1} = \mathbf{J}_1^{-1}\frac{\mathbf{C}_0}{h_1}. \tag{7.5}$$

Let $\mathbf{J}_i$ denote $\mathbf{C}_i/h_i + \mathbf{G}_i$ in the remaining of this chapter. In addition, the LU factorizations of $\mathbf{J}_i$ are already calculated since they are used to solve the circuit state at each time step before we calculate the sensitivity.

Next, for the solution at t_2, with the previous two steps information available, Gear-2 integration can be applied,

$$\frac{1}{h_2}\left[\mathbf{q}(\mathbf{x}_2) - \alpha_1\mathbf{q}(\mathbf{x}_1) - \alpha_2\mathbf{q}(\mathbf{x}_0)\right] + \mathbf{f}(\mathbf{x}_2) = \mathbf{b}_2. \tag{7.6}$$

In view of the sensitivity of $\mathbf{x}_2$ with respect to the changes of $\mathbf{x}_0$, Eq. (7.6) indicates that $\mathbf{x}_2$'s perturbation can be traced back to $\mathbf{x}_0$ along two paths: $\mathbf{x}_2$ is directly affected by $\mathbf{x}_0$, and $\mathbf{x}_2$ is also affected indirectly by $\mathbf{x}_0$ via $\mathbf{x}_1$. That is,

$$\frac{\mathrm{d}\mathbf{x}_2}{\mathrm{d}\mathbf{x}_0} = \frac{\partial\mathbf{x}_2}{\partial\mathbf{x}_1}\frac{\mathrm{d}\mathbf{x}_1}{\mathrm{d}\mathbf{x}_0} + \frac{\partial\mathbf{x}_2}{\partial\mathbf{x}_0},$$

where the two partial derivatives are

$$\frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1} = \mathbf{J}_2^{-1}\frac{\alpha_1}{h_2}\mathbf{C}_1, \qquad \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_0} = \mathbf{J}_2^{-1}\frac{\alpha_2}{h_2}\mathbf{C}_0,$$

and $\mathrm{d}\mathbf{x}_1/\,\mathrm{d}\mathbf{x}_0$ is calculated previously in Eq. (7.5). Thus, the sensitivity matrix deduced from Eq. (7.6) is

$$\frac{\mathrm{d}\mathbf{x}_2}{\mathrm{d}\mathbf{x}_0} = \mathbf{J}_2^{-1}\frac{\alpha_1}{h_2}\mathbf{C}_1 \cdot \mathbf{J}_1^{-1}\frac{\mathbf{C}_0}{h_1} + \mathbf{J}_2^{-1}\frac{\alpha_2}{h_2}\mathbf{C}_0. \tag{7.7}$$

Likewise, for t_3, apply the Gear-2 integration formula to DAE (7.1),

$$\frac{1}{h_3}[\mathbf{q}(x_3) - \alpha_1\mathbf{q}(x_2) - \alpha_2\mathbf{q}(x_1)] + \mathbf{f}(x_3) = \mathbf{b}_3, \tag{7.8}$$

and the chain rule for sensitivity is

$$\frac{\mathrm{d}\mathbf{x}_3}{\mathrm{d}\mathbf{x}_0} = \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2}\frac{\mathrm{d}\mathbf{x}_2}{\mathrm{d}\mathbf{x}_0} + \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_1}\frac{\mathrm{d}\mathbf{x}_1}{\mathrm{d}\mathbf{x}_0} = \mathbf{J}_3^{-1}\left[\frac{\alpha_1}{h_3}\mathbf{C}_2\frac{\mathrm{d}\mathbf{x}_2}{\mathrm{d}\mathbf{x}_0} + \frac{\alpha_2}{h_3}\mathbf{C}_1\frac{\mathrm{d}\mathbf{x}_1}{\mathrm{d}\mathbf{x}_0}\right],$$

where both $\mathrm{d}\mathbf{x}_2/\,\mathrm{d}\mathbf{x}_0$ and $\mathrm{d}\mathbf{x}_1/\,\mathrm{d}\mathbf{x}_0$ are ready from the previous two time steps t_1 and t_2 in Eqs. (7.7) and (7.5). Follow this chain rule in the aforementioned recursive style, the sensitivity matrix for Gear-2 integration is computed along the remaining time steps up to the M-th step,

$$\mathbf{J} = \frac{\mathrm{d}\mathbf{x}_M}{\mathrm{d}\mathbf{x}_0} = \mathbf{J}_M^{-1}\left[\frac{\alpha_1}{h_M}\mathbf{C}_{M-1}\frac{\mathrm{d}\mathbf{x}_{M-1}}{\mathrm{d}\mathbf{x}_0} + \frac{\alpha_2}{h_M}\mathbf{C}_{M-2}\frac{\mathrm{d}\mathbf{x}_{M-2}}{\mathrm{d}\mathbf{x}_0}\right]. \tag{7.9}$$

Note that as matrix-free GMRES method is applied, which only requires matrix-vector multiplication and no explicit $\mathbf{J}$ is required, as explained in Algorithm 11.

With matrix-free method, the matrix-vector multiplication (Line 4 of Algorithm 8 in Section 5.3) is replaced by the iteration shown in Algorithm 11, which calculates the multiplication product of the Gear-2 sensitivity we encounter in envelope-following and a

Algorithm 11 Gear-2 style matrix-free method to replace $\mathbf{Av}$ in Line 4 in Algorithm 8.

Input: input vector $\mathbf{v}$, time step lengths h_i, saved $\mathbf{C}_i$ matrices and LU factors of $\mathbf{J}_i$.

Output: matrix-vector product $\mathbf{r}$, such that $\mathbf{r} = \mathbf{Av}$.

1: Solve $\mathbf{J}_1\mathbf{p}_2 = h_1^{-1}\mathbf{C}_0\mathbf{v}$ for $\mathbf{p}_2$

2: Solve $\mathbf{J}_2\mathbf{p}_1 = h_2^{-1}\left(\alpha_1\mathbf{C}_1\mathbf{p}_2 + \alpha_2\mathbf{C}_0\mathbf{v}\right)$ for $\mathbf{p}_1$

3: **for** $i = 3 \ldots M$ **do** *// All remaining time steps*

4: Solve $\mathbf{J}_i\mathbf{p}_0 = \alpha_1 h_i^{-1}\mathbf{C}_{i-1}\mathbf{p}_1 + \alpha_2 h_i^{-1}\mathbf{C}_{i-2}\mathbf{p}_2$ for $\mathbf{p}_0$

5: $\mathbf{p}_2 = \mathbf{p}_1$

6: $\mathbf{p}_1 = \mathbf{p}_0$

7: **end for**

8: $\mathbf{r} = (k-1)\mathbf{p}_0 - k\mathbf{v}$

basis vector in the Krylov subspace. It is noticeable that the scaling and shift of $\mathbf{J}$ in $\mathbf{A} = (k-1)\mathbf{J} - k\mathbf{I}$ as described in Eq. (7.3) is taken into consideration by Line 8.

For the matrix-free generation of new basis vectors, it is straightforward to apply CUBLAS and CUSPARSE routines, and some customized GPU kernel functions to implement Algorithm 11 with the stored LU matrices of $\mathbf{J}_i$ and $\mathbf{C}_i$ mentioned earlier. For example, in Line 4, as the iteration index i traverses all the M time points, sparse matrix-vector multiplication `csrmv` is first called to calculate $\mathbf{C}_{i-1}\mathbf{p}_1$ and $\mathbf{C}_{i-2}\mathbf{p}_2$. And after the two vector results are combined by `axpy` of CUBLAS, $\mathbf{p}_0$ is solved for by `trsv`, since as we noted before, $\mathbf{J}_i$ is already in LU factorization form when transient simulation at time step t_i finished.

7.4 Experimental results

All the experiments have been carried out on a server which has an Intel Xeon quad-core CPU with 2.0 GHz clock speed, and 24 GBytes memory. The GPU card mounted on this server is NVIDIA Tesla C2070, which contains 448 cores (14 MPs $\times$ 32 cores per MP) running at 1.30 GHz and has 4 GBytes on-chip memory.

The envelope-following method is implemented by following the algorithm mentioned in [115], and together with our proposed Gear-2 sensitivity matrix computation, the whole program is built upon an open-source SPICE, implemented in C [116]. To solve the Newton update equation, different methods are used to compare the computation time, such as direct LU, GMRES with explicitly formed matrix, and GMRES with implicit matrix-vector multiplication (matrix-free). Moreover, the matrix-free GMRES is also incorporated to the same SPICE simulator using CUDA C, as described in Section 7.3.

We use several integrated on-chip converters as simulation examples to measure running time and speedup. They include a Buck converter, a quasi-resonant flyback converter (shown in Fig. 7.3), and two boost converters. Each converter is directly integrated

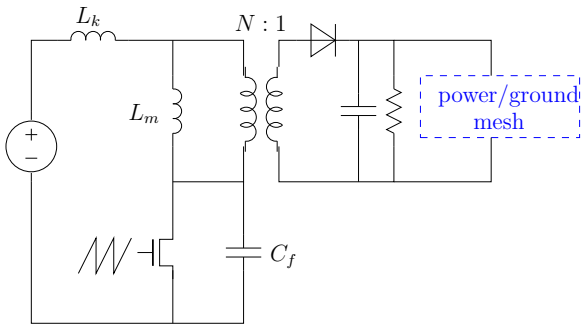


Figure 7.3: Diagram of a zero-voltage quasi-resonant flyback converter.

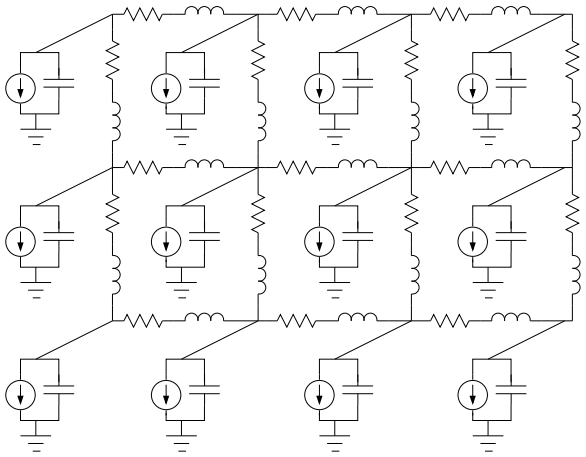
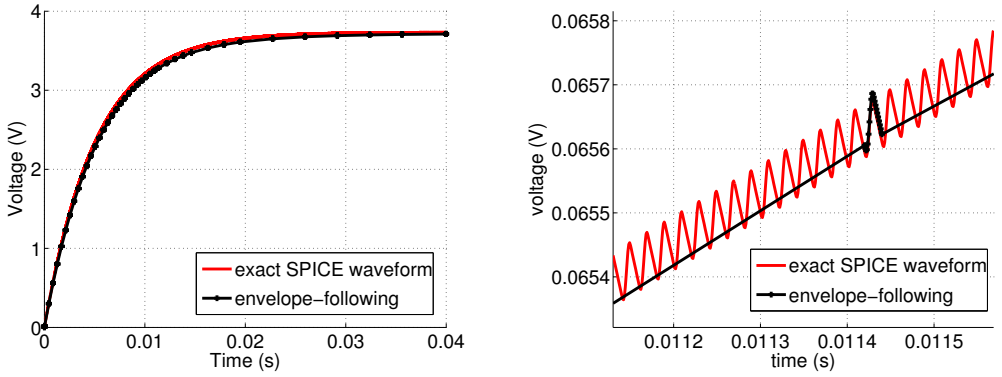


Figure 7.4: Illustration of power/ground network model.



(a) The whole plot

(b) Detail of one EF simulation period

Figure 7.5: Flyback converter solution calculated by envelope-following. The red curve is traditional SPICE simulation result, and the back curve is the envelope-following output with simulation points marked.

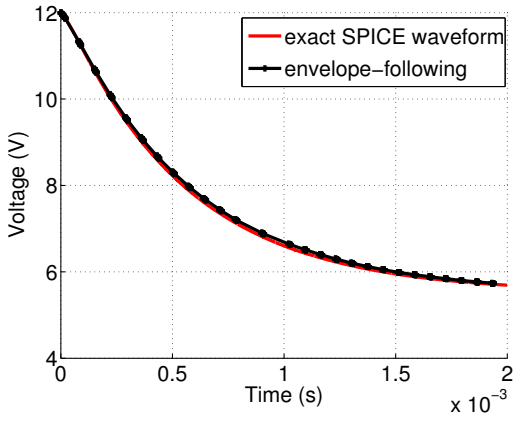


Figure 7.6: Buck converter solution calculated by envelope-following.

with on-chip power grid networks, since the performance of converters should be studied with their loads and we can easily observe the waveforms at different nodes in a power grid (see Fig. 7.4 for a simplified power grid structure).

Fig. 7.5 and Fig. 7.6 show the waveforms at output nodes of the resonant flyback converter and the Buck converter. Note that on the envelope curve, the darker dots in separated segments indicate the real simulation points were calculated in those cycles, and the segments without dots are the envelope jumps where no simulation were done. It can be verified that the proposed Gear-2 envelope-following method produces an envelope matching the original waveform well.

Table 7.1: CPU and GPU time comparisons (in seconds) for solving Newton update equation with the proposed Gear-2 sensitivity.

Circuit	Nodes	Direct	Explicit	Implicit GMRES		
		LU	GMRES	CPU	GPU	X
Buck	910	423.8	420.3	36.8	3.9	9.4×
Flyback	941	462.4	459.6	64.5	7.4	8.7×
Boost-1	976	695.1	687.7	73.2	6.2	11.8×
Boost-2	1093	729.5	720.8	71.0	8.5	9.9×

For comparison of running time spent in solving Newton update equation, Table 7.1 lists the time cost by direct method, explicit GMRES, matrix-free GMRES, and GPU matrix-free GMRES. All methods carry out the Gear-2 based envelope-following method, but they handle the sensitivity and equation solving in different implementation steps. It is obvious that as long as sensitivity matrix is explicitly formed, in direct method and explicit GMRES, the cost is much higher than implicit methods. When matrix-free technique is applied to generate matrix-vector products implicitly, the computation cost is greatly reduced. Thus, for the same example, implicit GMRES would be one order of magnitude faster than explicit GMRES. Furthermore, our GPU parallel implementation of implicit GMRES makes this method even faster, with a further 10× speedup.

7.5 Summary

A new envelope-following method for transient analysis of switching power converters has been introduced. First, the computationally expensive step, the solving of Newton update equation, has been parallelized on CUDA-enabled GPU platforms with iterative GMRES solver to boost performance of the analysis method. To further speed up the GMRES solving for Newton update equation, we have employed the matrix-free Krylov basis generation technique. The proposed method also applies the more robust Gear-2 integration to compute the sensitivity matrix. Experimental results from several integrated on-chip power converters have shown that the proposed GPU envelope-following algorithm can lead to about 10× speedup compared to its CPU counterpart, and 100× faster than the traditional envelope-following methods while still keeps the similar accuracy.

Chapter 8

Conclusion

This chapter concludes the whole thesis. It summarizes the research contributions for statistical analysis and parallel computing techniques we proposed for nano-scale VLSI design in the previous chapters.

8.1 Summary of research contributions

Monte Carlo and non-Monte Carlo analysis based on symbolic technique. In Chapter 2 and Chapter 3, we focused on statistical analysis of integrated circuits. Specifically, the Monte Carlo analysis in Chapter 2 is accelerated using parallel GPU platforms. This new work is based on a graph based symbolic technique, called DDD. Once the DDD of a circuit is constructed, its structure can be reused repeatedly with only the process variation affected parameters modified. The parallel evaluations of Monte Carlo take advantage of this fact and can result in speedup. Several procedures have been made on new data structures of levelized DDD, so that it is more friendly to GPU computing.

This so called levelization is done by first traversing the constructed DDD diagram so that the dependency between the nodes on different levels is known and then saving the node information in a vector-like data structure where nodes on same level are stored sequentially. In this way, the new vector structures allow GPU threads to access memory and compute the value in a coalesced fashion and minimize the branch divergence. The configuration of GPU thread blocks and kernel settings is also investigated to fully unleash the GPU parallel power.

In Chapter 3, still starting from DDD, we endeavor the other research direction in statistical analysis, named the non-Monte Carlo method. We generate the frequency domain performance bounds of the transfer function of a circuit using DDD symbolic expression. The bounds are calculated as a result of nonlinear constrained optimization. Furthermore, using the frequency domain bounds, we proposed a method to compute the time domain performance bound of the output in response to a given input signal. Experiments are run to show the accuracy and efficiency of our method.

Fast power grid simulation with CPU-GPU parallel computing. A new transient analysis method has been discussed in Chapter 4 for general linear dynamic networks, such as on-chip power grid networks, using hybrid GPU and multi-core platforms. The new algorithm, ETBR-GPU, is a reduction based simulation technique and is very amenable for parallelization on the hybrid multi-core CPU and GPU platforms, where coarse-grained task level and fine-grained lightweight thread level parallelism can be both exploited. A truncated balance reduction is carried out to generate the reduced model, whose construction employs parallel frequency domain sampling on multi-core CPU. Then

the reduced system is transferred to GPU. GPU accelerates both the linear algebra operations and parallel current source evaluation. We have given detailed descriptions on the implementation such as GPU thread and kernel configurations for best performance consideration. The proposed method can analyze any linear networks with complicated structures and macro-models. It especially favors transient simulation with long simulation periods as reduction costs in frequency domain is less sensitive to the simulation time. Experimental results show that the new method can achieve about one or two orders of magnitudes speedup when compared to the general LU-based simulation method on some recently published IBM power grid benchmark circuits.

GPU accelerated iterative solvers applied in a variety of EDA applications. In Chapter 5, Chapter 6, and Chapter 7, we conducted several researches all around one central topic — solving efficient linear equation in circuit simulation. The linear equation system, such as $\mathbf{Ax} = \mathbf{b}$, is a natural part of applications in many EDA programs. At the huge dimension of current IC models, the linear equation system is so large that direct solvers such as LU costs long time to compute it. Instead, iterative solvers are more preferred since it converges to a solution close enough to the exact solution and its computation cost is much cheaper than direct methods. We did some research using a popular iterative solver, GMRES, to incorporate our innovations.

In Chapter 5, an efficient finite difference based full-chip simulation algorithm for 3D-ICs with liquid cooling based on CPU-GPU platform is proposed. Unlike existing fast thermal analysis methods, the new method starts from the basic heat equations to model 3D-ICs with inter-tier liquid cooling micro-channels, and directly solves the resulting PDE

using iterative GMRES solver. To speed up the analysis process, we further developed a preconditioned GPU-accelerated GMRES solver. We also studied different preconditioners for GPU platforms and compared their performances. Experimental results showed that the proposed solver can lead to order of magnitudes speedup over the parallel LU based solver and up to 4× speedup over CPU-GMRES solver for both DC and transient thermal analyses on a number of thermal circuits and other published problems.

In Chapter 6, shooting method is investigated for periodic steady state analysis. This simulation algorithm is to obtain the steady state of a radio frequency circuit without running transient simulation over a lot of signal cycles. To reduce the computation cost, a matrix-free technique of subspace basis vector generation is employed. This saving is a result of the virtue of GMRES, which only requires the product of matrix-vector multiplication, but does not require the explicit form of the matrix. We also adopt the p-cyclic subspace construction method, which allows parallel computation of multiple Krylov subspaces to further accelerate the GMRES. We have presented adequate explanation on the GPU implementation of our new solver.

In Chapter 7, GMRES solver is adapted to support a modified transient integration scheme — Gear-2 integration, such that better integration accuracy is attained for envelope following method in the simulation of power converters, where high frequency signals are carried by slow changing signals and traditional simulation method is ineffective. GPU implementation is applied for better speedup.

8.2 Future research topics

Innovations in both EDA algorithms and GPU techniques keep emerging at a breathtaking fast speed. We the people in this community welcome the new solutions and actively involve with them. The works discussed in this thesis may appear to be less innovative or even outdated in the near future. However, we believe their values will not fade in the aforementioned applications. For example, during our wrapping up of this thesis, the ILU0 preconditioner we talked in Chapter 5 has been included in 5.0 version of NVIDIA's CUSPARSE Library [117]. The function interface in this library takes the sparse coefficient matrix $\mathbf{A}$ in the equation $\mathbf{Ax} = \mathbf{b}$ as input, and after the computation, the components of its ILU0 preconditioner, in terms of lower and upper triangular factors, are returned. The incorporation of this function into such a widely distributed library proves and confirms our vision for this specific area.

Our passion in the research and exploration of faster and effective computation solutions will never decrease. Here, in the end of this thesis, we do forecast some of our future works.

GPU computation with hierarchical symbolic technique. The DDD based analysis in Chapter 2 is processed on the whole circuit netlist. While we have shown speedup of our GPU accelerated MC simulation using DDD, the speedup can be further improved by using hierarchical DDD technique. This technique partition the big circuit into proper subsets where DDDs are built locally and independently. Therefore, it gives us another level of parallel computation. We plan to send the evaluation jobs of these sub-DDDs to GPU for better run-time scalability. With the assistance of multiple-GPU support, independent

DDD evaluations can be dispatched to a number of GPU cards, who are all mounted on the same host server, and collect the results after they finishes their corresponding parts. We expect to handle huge sized circuit at an eminently fast speed.

GPU accelerated power grid simulation with partitioning algorithms. Parasitic extraction in power grid interconnection models is growing faster in nano-scale integration, and we have encountered large test cases, such as the IBM benchmarks [54] we saw in Chapter 4. To allow efficient transient simulation of these tough examples, a power grid should be divided into smaller partitions. A well designed partitioning algorithm, such as domain decomposition, minimizes the dependency between the partitioned parts, so that simulation can be done separately in each part and the final merged result remains accurate enough for most specification requirements.

More robust GMRES solver for equation solving in EDA problems. In the last several chapters, we investigated the Krylov subspace based GMRES solver exhaustively. We have seen that a good preconditioner will greatly facilitate the solving of equations. However, the preconditioners in Chapter 5 are still not optimal and we have plans to implement better ones. For instance, we noticed recently that the simple ILU0 preconditioner in thermal analysis does not work well and even fails for power grid benchmarks. We have ruled out the cause of these malfunctions, and it is because the ILU0 preconditioner we used does not use pivoting and permutations. In addition, thermal circuit models are relatively easier since their matrices are diagonal dominant, which is not the case in power grid circuits. The more complicated matrix structure in power grid examples directly results in difficult convergence properties, which ILU0 cannot handle

swiftly because it does not have fill-in elements. Instead, we are currently developing ILU with fill-ins and permutations, and our initial results are satisfactory on all the industrial power grid benchmarks. We will also study more types of suitable GMRES preconditioners besides ILU family.

As new GPU architectures come to market, the related programming and implementation strategies are also updated. This can be observed from the ever changing configurations and interfaces in NVIDIA CUDA SDK. Our efforts are needed to catch up this trend in order that these new proposals deliver attractive results.

Bibliography

[1] H. Masuda, S. Ohkawa, A. Kurokawa, and M. Aoki. Challenge: Variability characterization and modeling for 65- to 90-nm processes. In *Proc. IEEE Custom Integrated Circuits Conf.*, 2005.

[2] J. Kim, K. D. Jones, and M. A. Horowitz. Fast, non-Monte-Carlo estimation of transient performance variation due to device mismatch. In *Proc. IEEE/ACM Design Automation Conference (DAC)*, pages 1746–1755, 2007.

[3] M. J. M. Pelgrom, A. C. J. Duinmaijer, and A. P. G. Welbers. Matching properties of MOS transistors. *IEEE J. of Solid State Circuits*, pages 1433–1439, 1989.

[4] S. W. Director, P. Feldmann, and K. Krishna. Statistical integrated circuit design. *IEEE J. of Solid State Circuits*, pages 193–202, 1993.

[5] J. Oehm and K. Schumacher. Quality assurance and upgrade of analog characteristics by fast mismatch analysis option in network analysis environment. *IEEE J. of Solid State Circuits*, pages 865–871, 1993.

[6] C.C. McAndrew, J. Bates, R.T. Ida, and P.G. Drennan. Efficient statistical BJT modeling, why beta is more than ic/ib. In *Proc. IEEE Bipolar/BiCMOS Circuits and Tech. Meeting*, 1997.

[7] G. Biagetti, S. Orcioni, C. Turchetti, P. Crippa, and M. Alessandrini. SiSMA: A tool for efficient analysis of analog cmos integrated circuits affected by device mismatch. *IEEE TCAD*, pages 192–207, 2004.

[8] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[9] R. Rutenbar. Next-generation design and EDA challenges. In *Proc. Asia South Pacific Design Automation Conf. (ASPDAC)*, January 2007. Keynote speech.

[10] S. Nassif. Model to hardware correlation for nm-scale technologies. In *Proc. IEEE International Workshop on Behavioral Modeling and Simulation (BMAS)*, Sept 2007. Keynote speech.

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

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

[13] David B. Kirk and Wen-Mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2010.

[14] Dominik Göddeke. General-purpose computation using graphics harware. `http://www.gpgpu.org/`, 2011.

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

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

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

[18] G. Gielen, P. Wambacq, and W. Sansen. Symbolic analysis methods and applications for analog circuits: A tutorial overview. *Proc. of IEEE*, 82(2):287–304, Feb. 1994.

[19] C.-J. Shi and X.-D. Tan. Canonical symbolic analysis of large analog circuits with determinant decision diagrams. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 19(1):1–18, Jan. 2000.

[20] C.-J. Shi and X.-D. Tan. Compact representation and efficient generation of s-expanded symbolic network functions for computer-aided analog circuit design. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(7):813–827, April 2001.

[21] X.-D. Tan and C.-J. Shi. Hierarchical symbolic analysis of large analog circuits via determinant decision diagrams. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 19(4):401–412, April 2000.

[22] S. X.-D. Tan, W. Guo, and Z. Qi. Hierarchical approach to exact symbolic analysis of large analog circuits. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 24(8):1241–1250, August 2005.

[23] R. E. Bryant. Binary decision diagrams and beyond: enabling technologies for formal verification. In *Proc. Int. Conf. on Computer Aided Design (ICCAD)*, 1995.

[24] S. Minato. *Binary Decision Diagrams and Application for VLSI CAD*. Kluwer Academic Publishers, Boston, 1996.

[25] A. Singhee and R. A. Rutenbar. Why quasi-Monte Carlo is better than Monte Carlo or Latin hypercube sampling for statistical circuit analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(11):1763—1776, 2010.

[26] B. Liu, J. Messaoudi, and G. Gielen. A fast analog circuit yield estimation method for medium and high dimensional problems. In *Proc. Design, Automation, and Test in Europe (DATE)*, pages 751–756, 2012.

[27] L. V. Kolev, V. M. Mladenov, and S. S. Vladov. Interval mathematics algorithms for tolerance analysis. *IEEE Trans. on Circuits and Systems*, 35(8):967–975, August 1988.

[28] Wei Tian, Xie-Ting Ling, and Ruey-Wen Liu. Novel methods for circuit worst-case tolerance analysis. *IEEE Trans. on Circuits and Systems I: Fundamental Theory and Applications*, 43(4):272–278, April 1996.

[29] C.-J. Richard Shi and Michael W. Tian. Simulation and sensitivity of linear analog circuits under parameter variations by robust interval analysis. *ACM Trans. Des. Autom. Electron. Syst.*, 4:280–312, July 1999.

[30] J. Vlach and K. Singhal. *Computer Methods for Circuit Analysis and Design*. Van Nostrand Reinhold, New York, NY, 1995.

[31] R. Spence and R.S. Soin. *Tolerance Design of Electronic Circuits*. Addison-Wesley, Reading, MA., 1988.

[32] L. Qian, D. Zhou, S. Wang, and X. Zeng. Worst case analysis of linear analog circuit performance based on Kharitonov's rectangle. In *Proc. IEEE Int. Conf. on Solid-State and Integrated Circuit Technology (ICSICT)*, Nov. 2010.

[33] V. L. Kharitonov. Asymptotic stability of an equilibrium position of a family of systems of linear differential equations. *Differential. Uravnen.*, 14:2086–2088, 1978.

[34] Siwat Saibua, Liuxi Qian, and Dian Zhou. Worst case analysis for evaluating VLSI circuit performance bounds using an optimization method. In *IEEE/IFIP 19th International Conference on VLSI and System-on-Chip*, pages 102–105, 2011.

[35] C. Pritchard and B. Wigdorowitz. Improved method of determining time-domain transient performance bounds from frequency response uncertainty regions. *International Journal of Control*, 66(2):311–327, 1997.

[36] Richard H. Byrd, Robert B. Schnabel, and Gerald A. Shultz. A trust region algorithm for nonlinearly constrained optimization. *SIAM Journal on Numerical Analysis*, 24(5):pp. 1152–1170, 1987.

[37] Philip E. Gill, Walter Murray, Michael, and Michael A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Journal on Optimization*, 12:979–1006, 1997.

[38] C. A. Floudas. *Nonlinear and Mixed-Integer Optimization: Fundamentals and Applications (Topics in Chemical Engineering)*. Oxford University Press, 1995.

[39] B. P. Lathi. *Modern Digital and Analog Communication Systems*. Oxford University Press, third edition, 1998.

[40] A. V. Oppenheim and R. W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, 1999.

[41] The Mathworks Inc. *MATLAB Optimization Toolbox*. `http://www.mathworks.com/help/toolbox/optim/`, 2012.

[42] C. Sánchez-López, F. V. Fernández, E. Tlelo-Cuautle, and S. X.-D. Tan. Pathological element-based active device models and their application to symbolic analysis. *IEEE Transactions on Circuits and Systems I: Regular papers*, 58(6):1382–1395, June 2011.

[43] A. A. Palma-Rodriguez, E. Tlelo-Cuautle, S. Rodriguez-Chavez, and S. X.-D. Tan. DDD-based symbolic sensitivity analysis of active filters. In *Proc. Intl. Caribbean Conf. on Circuits, Devices, and Systems (ICCDCS)*, pages 170–173, March 2012.

[44] International technology roadmap for semiconductors (ITRS), 2010 update, 2010. http://public.itrs.net.

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

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

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

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

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

[50] D. Li, S. X.-D. Tan, and B. McGaughy. ETBR: Extended truncated balanced realization method for on-chip power grid network analysis. In *Proc. Design, Automation and Test In Europe. (DATE)*, pages 432–437, 2008.

[51] Zhuo Feng and Peng Li. Multigrid on GPU: tackling power grid analysis on parallel SIMT platforms. In *ICCAD '08*, pages 647–654. IEEE Press, 2008.

[52] 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)*, pages 178–183, July 2009.

[53] Zhuo Feng and Zhiyu Zeng. Parallel multigrid preconditioning on graphics processing units (GPUs) for robust power grid analysis. In *Proc. Design Automation Conf. (DAC)*, pages 661–666, New York, NY, USA, 2010. ACM.

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

[55] NVIDIA. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit.

[56] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense Linear Algebra Solvers for Multicore with GPU Accelerators. In *Distributed Processing, Workshops and PhD Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

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

[58] Robert Patti. 3D integration: New opportunities for advanced packaging. In *Electrical Performance of Electronic Package and Systems (EPEPS)*, pages 1–41, October 2011.

[59] J. Burns. TSV-based 3D integration. In A. Papanikolaou, D. Soudris, and R. Radojcic, editors, *Three Dimensional System Integration*, pages 13–22. Springer, November 2010.

[60] José L. Ayala, Arvind Sridhar, and David Cuesta. Thermal modeling and analysis of 3D multi-processor chips. *Integration, the VLSI Journal*, 43:327–341, September 2010.

[61] IBM Interlayer Cooling Technology for 3D Packages. http://www.zurich.ibm.com/st/cooling/integrated.html.

[62] A. K. Coskun, D. Atienza, et al. Energy-efficient variable-flow liquid cooling in 3D stacked architectures. In *Proc. European Design and Test Conf. (DATE)*, pages 111–116. IEEE Press, 2010.

[63] X. Wei and Y. Joshi. Optimization study of stacked micro-channel heat sinks for micro-electronic cooling. *IEEE Transaction on Components and Packaging Technologies*, 26(1):441–448, 2003.

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

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

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

[67] T. Brunschwiler, B. Michel, et al. Interlayer cooling potential in vertically integrated packages. *Microsyst. Technol.*, 15:57–74, October 2008.

[68] A. M. Sridhar, A. Vincenzi, et al. Compact transient thermal model for 3D ICs with liquid cooling via enhanced heat transfer cavity geometries. In *16th International Workshop on Thermal Investigation of ICs and Systems*, October 2010.

[69] Zhuo Feng and Peng Li. Fast thermal analysis on GPU for 3D-ICs with integrated microchannel cooling. In *Proc. Int. Conf. on Computer Aided Design (ICCAD)*, pages 551–555, November 2010.

[70] J. H. Lienhard IV and J. H. Lienhard V. *A Heat Transfer Textbook*. Phlogiston Press, 2003.

[71] R. Shah and A. London. *Laminar Flow Forced Convection in Ducts*. Academic Press, New York, USA, 1978.

[72] 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*, 19(10):1823–1836, 2011.

[73] Mingliang Wang, Hector Klie, et al. Solving sparse linear systems on NVIDIA Tesla GPUs. In *Proc of the 9th Intl. Conf. on Computational Science*, pages 864–873, 2009.

[74] Ruipeng Li and Yousef Saad. GPU-accelerated preconditioned iterative linear solvers, 2010.

[75] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[76] Raphal Couturier and Stphane Domas. Sparse systems solving on GPUs with GMRES. *The Journal of Supercomputing*, 59(3):1504–1516, March 2012.

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

[78] Michele Benzi, Carl D. Meyer, and Miroslav Tuma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM J. Sci. Comput.*, 17:1135–1149, September 1996.

[79] Shiming Xu, Wei Xue, Ke Wang, and Haixiang Lin. Generating approximate inverse preconditioners for sparse matrices using CUDA and GPGPU. *Journal of Algorithm & Computational Technology*, 5(3):475–500, 2010.

[80] Y. Saad. Preconditioning techniques for nonsymmetric and indefinite linear systems. *J. Comput. Appl. Math*, 24:89–105, 1988.

[81] Michele Benzi and Miroslav Tuma. Numerical experiments with two approximate inverse preconditioners. *BIT Numerical Mathematics*, 38(2):234–241, 1998.

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

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

[84] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

[85] Tim Davis. The University of Florida sparse matrix collection. `http://www.cise.ufl.edu/research/sparse/`.

[86] C. H. Doan et al. Design considerations for 60 GHz CMOS radios. *IEEE Commun. Mag.*, pages 132–140, 2004.

[87] T. C. Chen. Where CMOS is going: trendy hype vs. real technology. In *International Solid-State Circuits Conference*, pages 22–27, Feb. 2006.

[88] A. Hajimiri. Holistic design in mm-wave silicon ICs. *IEICE Trans. on Electronics*, pages 817–828, 2008.

[89] B. Razavi. Design of millimeter-wave CMOS radios: A tutorial. *IEEE Trans. Circuits Syst. I*, pages 4–16, 2009.

[90] T. J. Aprille and T. N. Trick. Steady-state analysis of nonlinear circuits with periodic inputs. *IEEE Proc.*, pages 108–114, 1972.

[91] S. Skelboe. Computation of the periodic steady-state response of nonlinear networks by extrapolation methods. *IEEE Trans. on Circuits and Systems*, pages 161–175, 1980.

[92] K. S. Kundert, J. K. White, and A. Sangiovanni-Vincentelli. *Steady-State Methods for Simulating Analog and Microwave Circuits*. Kluwer Academic Publishers, 1990.

[93] 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.

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

[95] Ognen Nastov, Rircardo Telichevesky, Ken Kundert, and Jacob White. Fundamentals of fast simulation algorithms for RF circuits. *Proceedings of the IEEE*, 95(3):600–621, 2007.

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

[97] G. H. Golub and C. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3rd edition, 1996.

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

[99] Zhuo Feng and Zhiyu Zeng. Parallel multigrid preconditioning on graphics processing units (GPUs) for robust power grid analysis. In *Proc. Design Automation Conf. (DAC)*, pages 661–666, 2010.

[100] Zhuo Feng, Zhiyu Zeng, and Peng Li. Parallel On-Chip Power Distribution Network Analysis on Multi-Core-Multi-GPU Platforms. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 19(10):1823–1836, 2011.

[101] Bo Wang, Yuhao Zhu, and Yangdong Deng. Distributed time, conservative parallel logic simulation on GPUs. In *Proc. Design Automation Conf. (DAC)*, pages 761–766, 2010.

[102] Kanupriya Gulati, John F. Croix, Sunil P. Khatr, and Rahm Shastry. Fast circuit simulation on graphics processing units. In *Proc. Asia South Pacific Design Automation Conf. (ASPDAC)*, pages 403–408, 2009.

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

[104] 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)*, pages 573–578, June 2010.

[105] Daniel Kressner. A periodic Krylov-Schur algorithm for large matrix products. *Numer. Math.*, 103(3):461–483, 2006.

[106] NVIDIA. CUDA C programming guide. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, October 2012.

[107] Muthu Manikandan Baskaran and Rajesh Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. IBM Research Report RC24704, IBM Research Division, April 2009.

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

[109] Andrzej M. Trzynadlowski. *Introduction to Modern Power Electronics*. Wiley, second edition, 2010.

[110] K. Kundert et al. An envelope following method for the efficient transient simulation of switching power and filter circuits. In *Proc. ICCAD*, pages 446–449, Oct. 1988.

[111] J. White and S. Leeb. An envelope-following approach to switching power converter simulation. *IEEE Trans. Power Electron.*, 6(2):303–307, Apr. 1991.

[112] R. W. Freund and P. Feldmann. Reduced-order modeling of large linear subcircuits by means of the SyPVL algorithm. In *Proc. Int. Conf. on Computer Aided Design (ICCAD)*, pages 280–287, 1996.

[113] P. Krein. *Elements of Power Electronics*. Oxford University Press, 1997.

[114] C. W. Gear. *Numerical Initial Value Problems in Ordinary Differential Equations*. Prentice-Hall, Englewood Cliffs, NJ, 1971.

[115] T. Kato et al. Envelope following analysis of an autonomous power electronic system. In *IEEE COMPEL'06*, pages 29–33, July 2006.

[116] NGSPICE. `http://ngspice.sourceforge.net/`.

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