ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη, Υλοποίηση και Βελτιστοποίηση Αλγορίθμου για Προσομοίωση Μη Γραμμικών Κυκλωμάτων σε Παράλληλες Αρχιτεκτονικές

Design, Development and Optimization of a SPICE Model Evaluation Algorithm on Massively Parallel Architectures

Διπλωματική Εργασία

Άγγελος Τρίγκας

Επιβλέποντες Καθηγητές : Νέστωρ Ευμορφόπουλος
Επίκουρος Καθηγητής

Γεώργιος Σταμούλης
Καθηγητής

Βόλος, Ιούνιος 2013

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ
ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη, Υλοποίηση και Βελτιστοποίηση Αλγορίθμου για Προσομοίωση Μη Γραμμικών Κυκλωμάτων σε Παράλληλες Αρχιτεκτονικές

Διπλωματική Εργασία

Άγγελος Τρίγκας

Επιβλέποντες : Νέστωρ Ευμορφόπουλος
Επίκουρος Καθηγητής

Γεώργιος Σταμούλης
Καθηγητής

Εγκρίθηκε από την διμελή εξεταστική επιτροπή την 5η Ιουλίου 2013

.....	.....
Ν. Ευμορφόπουλος	Γ. Σταμούλης
Επίκουρος Καθηγητής	Καθηγητής

Διπλωματική Εργασία για την απόκτηση του Διπλώματος του Ηλεκτρολόγου Μηχανικού και Μηχανικού Υπολογιστών του Πανεπιστημίου Θεσσαλίας, στα πλαίσια του Προγράμματος Προπτυχιακών Σπουδών του Τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Πανεπιστημίου Θεσσαλίας.

……..

Άγγελος Τρίγκας

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών,

Πανεπιστημίου Θεσσαλίας

Στην οικογένεια & στους φίλους μου

Ευχαριστίες

Με την περάτωση της παρούσας εργασίας, θα ήθελα να ευχαριστήσω θερμά τους επιβλέποντες της διπλωματικής εργασίας κ. Νέστωρ Ευμορφόπουλο και κ. Γεώργιο Σταμούλη για την εμπιστοσύνη που επέδειξαν στο πρόσωπό μου με την ανάθεση του συγκεκριμένου θέματος και την άριστη συνεργασία μας.
Επίσης θα ήθελα να ευχαριστήσω του φίλους και συνεργάτες του εργαστηρίου Ε5 για την υποστήριξη και την δημιουργία ενός ευχαριστου και δημιουργικού κλίματος. Ιδιαίτερα θα ήθελα να ευχαριστήσω τον υποψήφιο διδάκτορα Κωνσταντή Νταλούκα και τον μεταπτυχιακό φοιτητή Χαράλαμπο Αντωνιάδη για την συνεχή καθοδήγηση και τις ουσιώδεις υποδείξεις και παρεμβάσεις, που διευκόλυναν την εκπόνηση της πτυχιακής εργασίας.
Μεγάλη βοήθεια προήλθε και από τον συμφοιτητη μου Πέτρο Καλό ο οποίος πρώτος ασχολήθηκε με το συγκεκριμένο θεμα και πάνω στην δουλειά του βασίστηκε η δικιά μας υλοποίηση.
Τέλος, οφείλω ένα μεγάλο ευχαριστώ στην οικογένειά μου και στους φίλους μου για την αμέριστη υποστήριξη και την ανεκτίμητη βοήθεια που μου παρείχαν τόσο κατά την διάρκεια των σπουδών μου όσο και κατά την εκπόνηση της διπλωματικής εργασίας.

Τρίγκας Άγγελος
Βόλος, 2013

Contents

List of Tables

List of Figures

List of Abbreviations

SPICE Simulation Program with Integrated Circuit Emphasis

VLSI Very-Large-Scale Integration

CUDA Compute Unified Device Architecture

PC Personal Computer

CAD Computer Aided Design

CPU Central Processing Unit

GPU Graphics Processing Unit

EDA Electronic Design Automation

DC Direct Current

AC Alternative Current

MNA Modified Nodal Analysis

HPC High Performance Computing

GPGPU General Purpose Graphics Processing Unit

SM Streaming Multiprocessor

SIMT Single Instruction Multiple Thread

SIMD Single Instruction Multiple Data

BJT Bipolar Junction Transistor

MOSFET Metal–Oxide–Semiconductor Field-Effect Transistor

JFET Junction gate Field-Effect Transistor

BSIM Berkeley Short-channel IGFET Model

DMA Direct Memory Access

Περίληψη

Το SPICE αποτελεί ένα κοινό βιομηχανικό πρότυπο για την προσομοίωση πολύ μεγάλης κλίμακας ολοκληρωμένων κυκλωμάτων. Η προσομοίωση με το SPICE αποτελεί έναν πρότυπο τρόπο επαλήθευσης της λειτουργίας του κυκλώματος στο επίπεδο των τρανζίστορ προτού ξεκινήσει η διαδικασία κατασκευής του. Το SPICE αποτελεί μια υπολογιστικά απαιτητική εφαρμογή, ειδικά όταν πρόκειται για μεγάλα κυκλώματα ή προσομοίωση σε πολλές χρονικές στιγμές. Ακριβείς προσομοιώσεις μπορεί να χρειαστούν μέρες ή και εβδομάδες εκτέλεσης σε σύγχρονους επεξεργαστές. Ως αποτέλεσμα υπάρχει τεράστιο κίνητρο για την επιτάχυνση του SPICE μέσω της παραλληλοποίησης των υπολογιστικά και χρονικά ‘ακριβών’ κομματιών του. Η αποτίμηση μοντέλων για τις συσκευές του κυκλώματος είναι η πιο δαπανηρή εργασία που επιτελεί ένας προσομοιωτής κυκλωμάτων σαν το SPICE. Σε αυτή την εργασία προσπαθούμε να επιταχύνουμε την αποτίμηση μοντέλων με την χρήση σύγχρονων καρτών γραφικών.

Abstact

SPICE is the de facto industry standard for circuit level simulation of VLSI integrated circuits. Simulating the circuit with SPICE is the industry-standard way to verify circuit operation at the transistor level before committing to manufacturing an integrated circuit. SPICE simulation is a computationally demanding application when it comes to large number of circuit elements or when very large number of simulation time points are needed. Accurate simulations might take days or weeks of runtime on modern microprocessors.

As a result, there is a significant motivation to speed up SPICE by parallelizing its compute intensive parts. Device model evaluation is the most time-consuming task in analog circuit simulators such as SPICE. In this thesis, we try to accelerate model evaluation using modern GPUs. For this purpose we use Ngspice, an open-source circuit simulator based on SPICE3 and a 'GeForce GTX560 Ti' GPU from Nvidia with 384 CUDA cores. Our experiments demonstrate speedups up to 1.83x.

Chapter 1

Introduction

1.1 Introduction and Problem Description

SPICE (Simulation Program with Integrated Circuit Emphasis) [7] is an analog circuit simulator that can take days or weeks of runtime on real-world problems. It models the analog behavior of semiconductor circuits using a compute-intensive non-linear differential equation solver. SPICE is notoriously difficult to parallelize due to its irregular, unpredictable compute structure, and a sloppy sequential description.

As shown in figure 1.1, SPICE simulation is an iterative computation that consists of two key computationally-intensive phases per iteration: Model Evaluation (2 in figure 1.1) followed by Matrix Solve (3 in figure 1.1). The iterations themselves are managed by the third phase, which is the Iteration Controller (1 in figure 1.1). Our profiling showed that on average 71% of SPICE runtime is spent in performing Model Evaluation. This is because these evaluations are performed for each device, many times per time-step, until the convergence of the NR based non-linear equation solver. The total number of such evaluations can easily run into billions. Therefore the speed of the Model Evaluation phase is a significant determinant of the speed of the overall SPICE simulator.

Furthermore, over the past decades, we have relied on innovations in computer architecture (clock frequency scaling, out-of-order execution, complex branch predictors) to speedup application like SPICE. However, diminishing transistor-speed scaling and practical energy limits have disrupted the rising clock frequency and threatened to annul Moore's Law. This lead to the rise of multicore and manycore systems. Recent multiprocessors with heterogeneous architectures have emerged as mainstream computing platforms, which typically integrate a variety of processing elements of different computing performance, programming flexibility and energy efficiency characteristics. Typical examples of heterogeneous platforms are today's personal computers (PCs) with multicore processors and manycore GPUs.VLSI CAD developers have the opportunity to fully utilize such heterogeneous computing architectures and gain unprecedented high performance.

1.2 Thesis Contribution

The contribution of this thesis is the design, implementation and optimization of a SPICE model evaluation algorithm targeting modern GPUs. In particular, we use the Ngspice [2], an

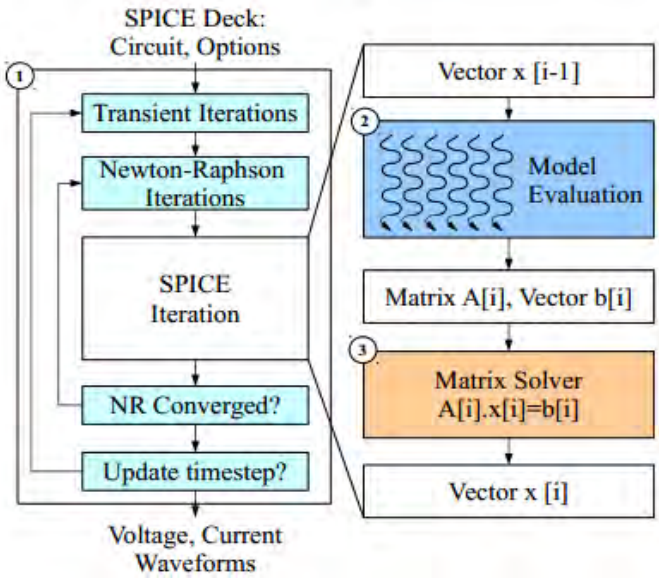


Figure 1.1: Flowchart of a SPICE Simulator

open-source circuit simulator and our implementation handles BSIM4 device models, but can easily handle a variety of other models. For the implementation we use the Compute Unified Device Architecture (CUDA) [1], which is an open-source programming and interfacing tool provided by NVIDIA. The GPU device we have used for the benchmarking is the NVIDIA 'GeForce GTX 560 Ti' with 384 CUDA cores.

1.3 Thesis Organization

Chapter 2 consist a presentation of a SPICE circuit simulator. It includes a general introduction, followed by the execution flowchart of SPICE, the parallelism potential and a few words about Ngspice.
Chapter 3 consist an overview of the architecture of modern NVIDIA GPUs and the CUDA programming model.
Chapter 4 presents the implementation of the model evaluation algorithm separated in stages and there is a reference to the problems we faced during this process.
Chapter 5 presents the results from the experiments conducted.
Chapter 6 finally gives some insights for future work.

Consequently, in the Appendix we have added the results of all the experiments conducted in every stage of the implementation.

Chapter 2

SPICE Review

2.1 Introduction

In modern Very Large-Scale Integrated circuit (VLSI) design, Electronic Design Automation (EDA) tools are used to accelerate the design cycle and to help engineers improve their design. SPICE is the most important EDA tool in circuit design. It is a general purpose circuit simulation program for DC, transient, linear AC, pole-zero, sensitivity and noise analysis. It was developed at the EECS Department of the University of California, Berkeley by Donald Pederson, Larry Nagel, Richard Newton, and many other contributors to provide a fast and robust circuit simulation program capable of verifying integrated circuits. It is used to simulate circuits for various applications from switching power supplies to SRAM cells and sense amplifiers. Doing so requires the simultaneous solution of a number of equations that capture the behavior of electronic circuits. The number of equations can be quite large for a modern electronic circuit with transistors count from several hundred thousands to some millions. Thus the simulation of circuits has become complex and quite time-consuming.

2.2 SPICE Algorithm

We will now present a brief overview of the sequential algorithm that SPICE follows. SPICE simulates the dynamic analog behavior of a circuit described by its constituent non-linear differential equations. SPICE circuit equations model the linear (e.g. resistors, capacitors, inductors) and non-linear (e.g. diodes, transistors) behavior of devices and the conservation constraints (i.e. Kirchoff's current laws|KCL) at the different nodes and branches of the circuit. SPICE solves the non-linear circuit equations by alternately computing small-signal linear operating-point approximations for the non-linear elements and solving the resulting system of linear equations until it reaches a fixed point. The linearized system of equations is represented as a solution of $A\vec{x} = \vec{b}$, where A is the matrix of circuit conductances, $\vec{b}$ is the vector of known currents and voltage quantities and $\vec{x}$ is the vector of unknown voltages and branch currents.
Spice3f5 [8] uses the Modified Nodal Analysis (MNA) technique [5] to assemble circuit equations into matrix A. The MNA approach is an improvement over conventional nodal analysis by allowing proper handling of voltage sources and controlled current sources. It requires the application of Kirchoff's Current Law at all the nodes in the circuit with voltage at each

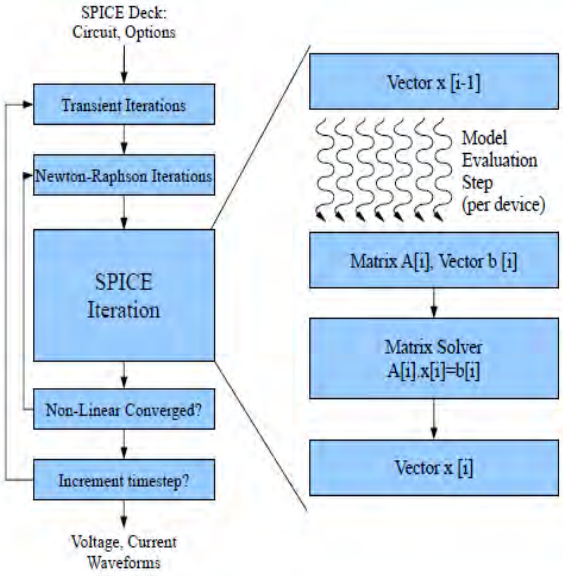


Figure 2.1: Flowchart of a SPICE Simulator

node being an unknown. It then introduces unknowns for currents through branches to allow voltage sources and controlled current sources to be represented.

The simulator calculates entries in A and $\vec{b}$ from the device model equations that describe device transconductance (e.g., Ohm's law for resistors, transistor $I - V$ characteristics) in the Model-Evaluation phase. It then solves for $\vec{x}$ using a sparse-direct linear matrix solver in the Matrix-Solve phase. We show the steps in the SPICE algorithm in figure 2.1.

2.3 Model Evaluation

The Model-Evaluation phase of SPICE calculates the currents and conductances of all the devices in the circuit. The computed currents and conductances are used to update the matrix A and the vector $\vec{b}$ (in $A\vec{x} = \vec{b}$). At the start, the simulator processes all the devices in the circuit. At subsequent timesteps, only the non-linear and time-varying elements change and must be recalculated during the Model-Evaluation phase. The resistors and uncontrolled sources have fixed conductances and do not need to update the matrix in every iteration. A device updates the matrix according to its stamp that specifies which entries it defines in the matrix. For an N-terminal device, we need to update at most N^2 entries in the matrix. Thus, each device in the circuit updates a constant number of entries in the matrix corresponding to its node terminals. The resulting stamps update the shared matrix entry corresponding to the device terminals.

For non-linear elements like diodes and transistors, the simulator must search for an operating-point using the Newton-Raphson iterations shown in the outer loop of figure 2.2. The conductance of the non-linear and time-varying elements is a function of the terminal voltages $\vec{x}$. Since the terminal voltages $\vec{x}$ are computed by the $A\vec{x} = \vec{b}$ solve, we need to use Newton-Raphson algorithm to iteratively compute the consistent solution vector $\vec{x}$. This requires repeated evaluation of the non-linear model equations multiple times per timestep. For

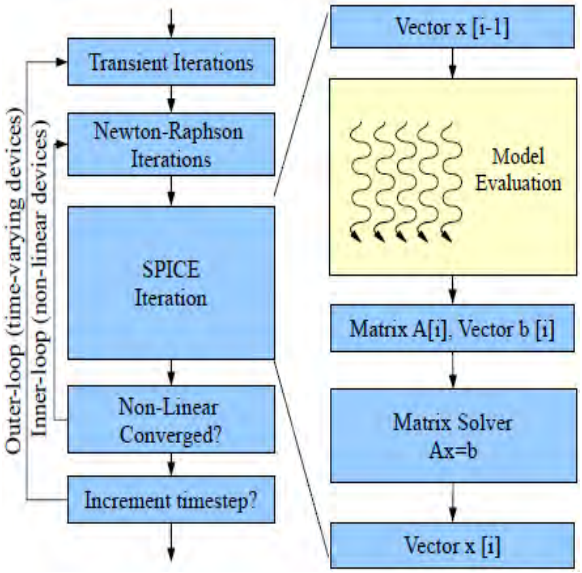


Figure 2.2: Flowchart of a SPICE Simulator with emphasis on Model-Evaluation

time-varying components like capacitors and inductors, the simulator must recalculate their contributions at each timestep based on voltages and charges at several previous timesteps (e.g. Trapezoidal integration). This also requires a re-evaluation of the device-model in each timestep.

2.4 Matrix Solve

The simulator spice3f5 uses the Modified Nodal Analysis (MNA) technique [5] to assemble circuit equations into matrix A. Since circuit elements (N) tend to be connected to only a few other elements, there are a constant number ($O(1)$) of entries per row of the matrix. Thus, the MNA circuit matrix with $O(N^2)$ entries is highly sparse with $O(N)$ nonzero entries (99% of the matrix entries are 0). The matrix structure is mostly symmetric with the asymmetry being added by the presence of independent sources (e.g. input voltage source) and inductors. The underlying non-zero structure of the matrix is defined by the topology of the circuit and consequently remains unchanged throughout the duration of the simulation. In each iteration of the loop shown in figure 2.1, only the numerical values of the non-zeroes are updated in the Model-Evaluation phase of SPICE with contributions from the non-linear elements. The matrix solve stages are demonstrated in figure 2.3. To find the values of unknown node voltages and branch currents $\vec{x}$, we must solve the system of linear equations $A\vec{x} = \vec{b}$ as shown in Equation 2.14. The sparse, direct matrix solver used in spice3f5 first reorders the matrix A to minimize fill-in using a technique called Markowitz reordering [6]. This tries to reduce the number of additional non-zeroes (fill-in) generated during LU factorization. It then factorizes the matrix by dynamically determining pivot positions for numerical stability (potentially adding new non-zeros) to generate the lower-triangular component L and upper-triangular component U such that $A = LU$ as shown in Equation 2.15. Finally, it calculates $\vec{x}$ using Front-Solve $L\vec{y} = \vec{b}$ (see Equation 2.16) and Back-Solve $U\vec{x} = \vec{y}$ operations (see Equation

$$A \cdot \vec{x} = \vec{b} \tag{2.10}$$

$$L \cdot U \cdot \vec{x} = \vec{b} \tag{2.11}$$

$$L \cdot \vec{y} = \vec{b} \tag{2.12}$$

$$U \cdot \vec{x} = \vec{y} \tag{2.13}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{2.14}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{2.15}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{2.16}$$

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \tag{2.17}$$

Figure 2.3: Matrix Solve Stages

2.17).

2.5 Iteration Controller

As discussed in earlier, SPICE is an iterative algorithm that solves non-linear differential equations. SPICE solves these equations using an iterative approach that first linearizes the non-linear circuit elements and then performs a numerical integration involving time-varying quantities. The space of algorithms for linearization and numerical integration is vast, and it covers conflicting requirements of convergence speed, accuracy and stability while demanding different amounts of computation and memory storage costs. The choice of a suitable algorithm applicable to circuit simulation is the subject of continued research and is beyond the scope of this thesis. Using the framework and methodology described in this chapter, we can support newer algorithms for managing the SPICE simulation. For the purpose of this thesis and as a proof-of-concept, we pick the algorithms used in the spice3f5 package: the Newton-Raphson algorithm for handling non-linear elements and the Trapezoidal approximation for numerical integration.

The spice3f5 iteration controller manages two kinds of iterative loops: (**1**) a loop for linearizing the non linear elements of the circuit, and (**2**) another loop for advancing the timestep of the simulation. We show these loops in figure 2.4. The convergence conditions for the Newton-Raphson algorithm are implemented in the block (**a**) of the figure. SPICE employs the Newton-Raphson algorithm for computing the linear operating points of non-linear devices like diodes and transistors. The equation for next timestep calculation is implemented in block (**b**) of the figure. SPICE uses an adaptive timestep-control algorithm that adjusts the timestep of the simulation based on an estimate of local truncation error. In block (**c**), SPICE implements the dynamic breakpoint processing logic for handling source transition timesteps in the voltage and current sources. Finally, in block (**d**), the analysis state ma-

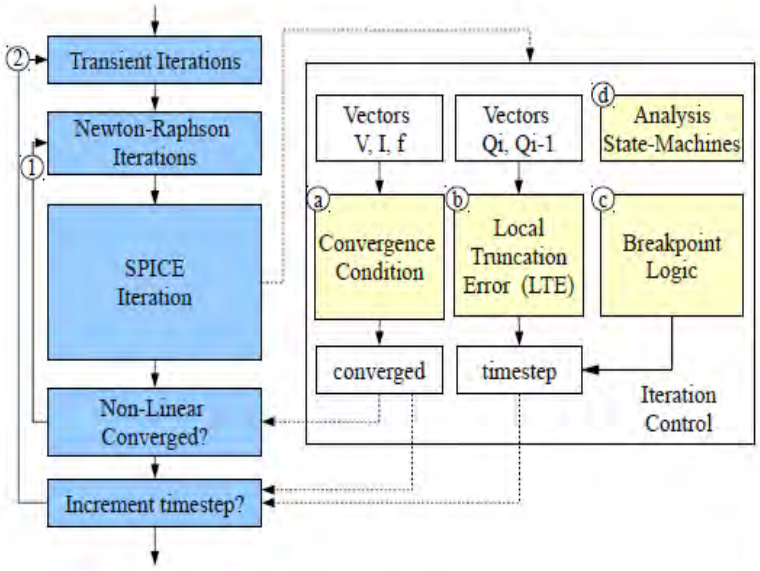


Figure 2.4: Flowchart of a SPICE Simulator with emphasis on SPICE Analysis Control Algorithms

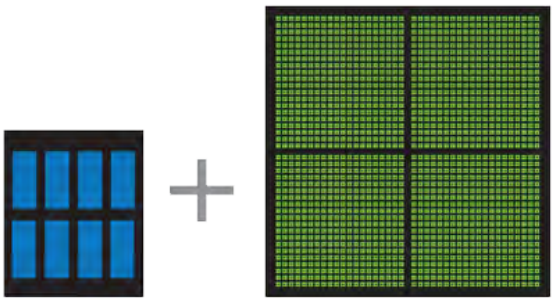
chines implement the loop control algorithms for performing DC and transient analysis. The Iteration Control computation constitutes a small (7%) fraction of total SPICE runtime.

Chapter 3

GPU Architecture and the CUDA Programming Model

3.1 Introduction

The last years we have witnessed the High Performance Computing community shift to General-Purpose Graphics Processing Unit (GPGPU) Computing. GPGPU computing is the use of a GPU (graphics processing unit) together with a CPU to accelerate general-purpose scientific and engineering applications. GPU computing offers unprecedented application performance by offloading compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU. From a user's perspective, applications simply run significantly faster. CPU + GPU is a powerful combination because CPUs consist of a few cores optimized for serial processing, while GPUs consist of thousands of smaller, more efficient cores designed for parallel performance. Serial portions of the code run on the CPU while parallel portions run on the GPU.



3.2 GPU Architecture

In this section, we discuss the architectural aspects of the NVIDIA GPUs in general and of GeForce GTX560 Ti, GPU device used in our experiments, in particular.

Hardware Model

Hardware Implementation

The NVIDIA GPU architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors.

A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs a unique architecture called SIMT (Single Instruction, Multiple-Thread) that is described in SIMT Architecture. The instructions are pipelined to leverage instruction-level parallelism within a single thread, as well as thread-level parallelism extensively through simultaneous hardware multithreading as detailed in Hardware Multithreading. Unlike CPU cores they are issued in order however and there is no branch prediction and no speculative execution.

SIMT Architecture

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. The term warp originates from weaving, the first parallel thread technology. A half-warp is either the first or second half of a warp. A quarter-warp is either the first, second, third, or fourth quarter of a warp.

When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0. Thread Hierarchy describes how thread IDs relate to thread indices in the block.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.

The SIMT architecture is akin to SIMD (Single Instruction, Multiple Data) vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with

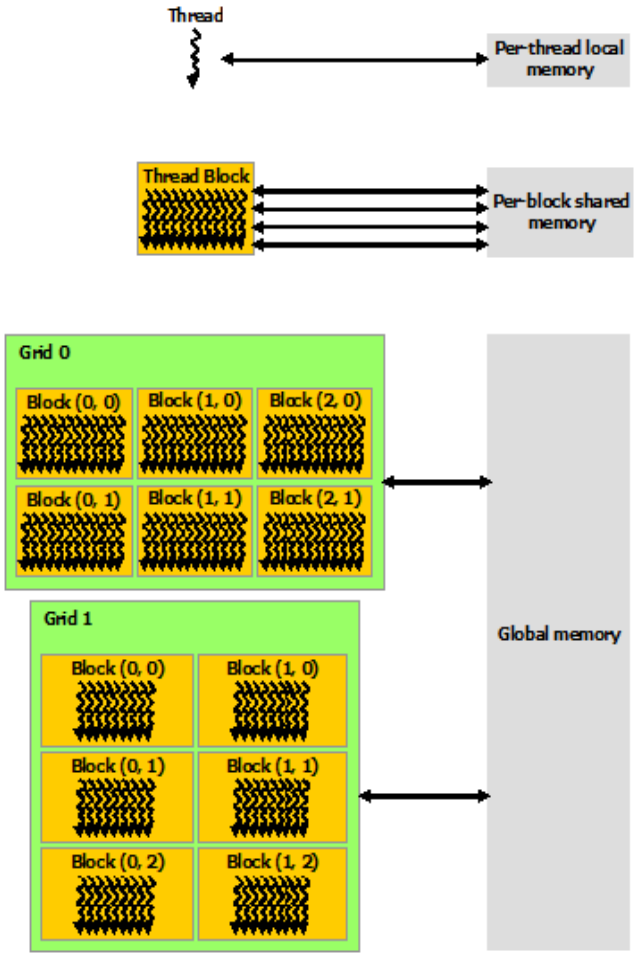


Figure 3.1: Memory Hierarchy

SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional code: Cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance. Vector architectures, on the other hand, require the software to coalesce loads into vectors and manage divergence manually.

Memory Model

CUDA threads may access data from multiple memory spaces during their execution as illustrated by figure 3.1. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.

There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages. Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats. The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

An instruction that accesses addressable memory (i.e., global, local, shared, constant, or texture memory) might need to be re-issued multiple times depending on the distribution of the memory addresses across the threads within the warp. How the distribution affects the instruction throughput this way is specific to each type of memory and described in the following sections. For example, for global memory, as a general rule, the more scattered the addresses are, the more reduced the throughput is.

Global Memory

Global memory resides in device memory and device memory is accessed via 32-, 64-, or 128-byte memory transactions. These memory transactions must be naturally aligned: Only the 32-, 64-, or 128-byte segments of device memory that are aligned to their size (i.e., whose first address is a multiple of their size) can be read or written by memory transactions.

When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly. For example, if a 32-byte memory transaction is generated for each thread's 4-byte access, throughput is divided by 8.

To maximize global memory throughput, it is therefore important to maximize coalescing by:

- Following the most optimal access patterns based on the Compute Capability of the device being used
- Using data types that meet the size and alignment requirement detailed in Device Memory Accesses
- Padding data in some cases, for example, when accessing a two-dimensional array as described in Device Memory Accesses

Local Memory

Local memory accesses only occur for some automatic variables. Automatic variables that the compiler is likely to place in local memory are:

- Arrays for which it cannot determine that they are indexed with constant quantities
- Large structures or arrays that would consume too much register space
- Any variable if the kernel uses more registers than available (this is also known as register spilling)

The local memory space resides in device memory, so local memory accesses have same high latency and low bandwidth as global memory accesses and are subject to the same requirements for memory coalescing as described in Device Memory Accesses. Local memory is however organized such that consecutive 32-bit words are accessed by consecutive thread IDs. Accesses are therefore fully coalesced as long as all threads in a warp access the same relative address (e.g., same index in an array variable, same member in a structure variable).

Shared Memory

Because it is on-chip, shared memory has much higher bandwidth and much lower latency than local or global memory.

To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module.

However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests. If the number of separate memory requests is n, the initial memory request is said to cause n-way bank conflicts.

Constant Memory

A constant memory fetch costs one memory read from the device memory only on a cache miss, otherwise it costs one read from the constant cache. The memory bandwidth is best utilized when all instructions that are executed in parallel access the same address of the constant memory.

Texture Memory

The texture cache is optimized for spatial locality. In other words if instructions that are executed in parallel read texture addresses that are close together, then the texture cache can be optimally utilized. A texture fetch costs one memory read from device memory on a cache miss, otherwise it costs one read from the texture cache. Device memory reads through texture fetching routines (provided in CUDA for accessing texture memory) present several benefits over reads from global or constant memory.

3.3 The CUDA Programming Model

This chapter introduces the main concepts behind the CUDA programming model by outlining how they are exposed in C.

CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. A kernel is defined using the global declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new $<<<$ … $>>>$ execution configuration syntax (see C Language Extensions). Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable.

For convenience, threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume. The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size (Dx, Dy),the thread ID of a thread of index (x, y) is $(x+y*Dx)$; for a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is $(x + y * Dx + z * Dx * Dy)$.

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads. However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as illustrated by figure 3.3. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed.

The number of threads per block and the number of blocks per grid specified in the $<<<$ ... $>>>$ syntax can be of type int or dim3. Two-dimensional blocks or grids can be specified as in the example above. Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in blockIdx variable. The dimension of the thread block is accessible within the kernel through the built-in blockDim variable.

A thread block size of 16x16 (256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before. For simplicity, this example assumes that the number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension, although that need not be the case.

Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write code that scales with the number of cores.

Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the *syncthreads()* intrinsic function; *syncthreads()* acts as a barrier at which all threads in the block must wait before any is allowed to proceed. Shared Memory gives an example of using shared memory.

For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and *syncthreads()* is expected to be lightweight.

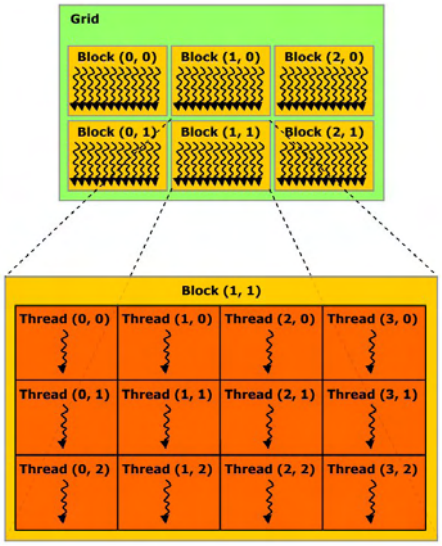


Figure 3.2: Grid of Thread Blocks

3.4 GeForce GTX560 Ti

GPU Engine Specifications

The GeForce GTX560 Ti is a Fermi-based architecture GPU from NVIDIA. It has 8 Multiprocessors per chip and 48 CUDA cores per Multiprocessor. It has Compute Capability 2.1. The warp size is 32 and can support 1536 threads per Multiprocessor and 1024 threads per block. Maximum sizes of each dimension of a block and grid are 1024 x 1024 x 1024 and 65535 x 65535 x 65535 respectively. It can support concurrent execution of multiple kernels.

Memory Specifications

The total amount of global memory for this device is 1GB. The amount of constant memory is 65KB. Each block has available 32768 registers and 49KB of shared memory. GeForce GTX560 Ti has available caching. The on-chip memory per multiprocessor is used for both L1 and shared memory, and how much of it is dedicated to L1 versus shared memory is configurable for each kernel call. Additionally, it has a global L2 cache of 524KB.

Figure 3.3: GeForce GTX560 Ti

Chapter 4

Implementation and Optimization

This chapter presents the implementation of the model evaluation algorithm on the GPU. At first, we will give some details about Ngspice, the open-source circuit simulation tool, which was used as a basis for the implementation. In particular, our approach parallelizes and accelerates the transistor model evaluation for the BSIM4 models [3]. Then, follow some details about the compilation system and the optimizations separated in stages.

4.1 Ngspice

Ngspice [2] is an open-source project. It is a general-purpose circuit simulation program for nonlinear and linear analyses. Circuits may contain resistors, capacitors, inductors, mutual inductors, independent or dependent voltage and current sources, loss-less and lossy transmission lines, switches, uniform distributed RC lines, and the five most common semiconductor devices: diodes, BJTs, JFETs, MESFETs, and MOSFETs.

Ngspice is an update of Spice3f5, the last Berkeley's release of SPICE3 simulator family. Ngspice is being developed to include new features to existing Spice3f5 and to fix its bugs. Improving a complex software like a circuit simulator is a very hard task and, while some improvements have been made, most of the work has been done on bug fixing and code refactoring.

Ngspice has built-in models for the semiconductor devices, and the user need specify only the pertinent model parameter values. There are three models for bipolar junction transistors, all based on the integral-charge model of Gummel and Poon; however, if the Gummel-Poon parameters are not specified, the basic model (BJT) reduces to the simpler Ebers-Moll model. In either case and in either models, charge storage effects, ohmic resistances, and a current-dependent output conductance may be included. The second bipolar model BJT2 adds dc current computation in the substrate diode. The third model (VBIC) contains further enhancements for advanced bipolar devices.

The semiconductor diode model can be used for either junction diodes or Schottky barrier diodes. There are two models for JFET: the first (JFET) is based on the model of Shichman and Hodges, the second (JFET2) is based on the Parker-Skellern model. All the original six MOSFET models are implemented: MOS1 is described by a square-law $I-V$ characteristic, MOS2 is an analytical model, while MOS3 is a semi-empirical model; MOS6 is a simple analytic model accurate in the short channel region; MOS9, is a slightly modified Level 3 MOS-

FET model - not to confuse with Philips level 9; BSIM 1; BSIM2 are the old BSIM (Berkeley Short-channel IGFET Model) models. MOS2, MOS3, and BSIM include second-order effects such as channel-length modulation, subthreshold conduction, scattering limited velocity saturation, small-size effects, and charge controlled capacitances. The recent MOS models for submicron devices are the BSIM3 (Berkeley BSIM3 web page) and BSIM4 (Berkeley BSIM4 web page) models. Silicon-on-insulator MOS transistors are described by the SOI models from the BSIMSOI family (Berkeley BSIMSOI web page) and the STAG one. There is partial support for a couple of HFET models and one model for MESA devices.

The ngspice simulator supports the following different types of analysis:

1. DC Analysis (Operating Point and DC Sweep)
2. AC Small-Signal Analysis
3. Transient Analysis
4. Pole-Zero Analysis
5. Small-Signal Distortion Analysis
6. Sensitivity Analysis
7. Noise Analysis

Ngspice on multi-core processors using OpenMP

Today's computers typically come with CPUs having more than one core. It will thus be useful to enhance ngspice to make use of such multi-core processors.

Using circuits comprising mostly of transistors and e.g. the BSIM3 model, around 2/3 of the CPU time is spent in evaluating the model equations (e.g. in the *BSIM3Load()* function). The same happens with other advanced transistor models. Thus this function should be paralleled, if possible. Resulting from that the parallel processing has to be within a dedicated device model. Interestingly solving the matrix takes only about 10% of the CPU time, so paralleling the matrix solver is of secondary interest here.

A recent publication [9] has described a way to exactly do that using OpenMP, which is available on many platforms and is easy to use, especially if you want to parallel processing of a for-loop.

Some results on an inverter chain with 627 CMOS inverters, running for 200ns, compiled with Visual Studio professional 2008 on Windows 7 (full optimization) or gcc 4.4, SUSE LINUX 11.2, -O2, on a i7 860 machine with four real cores (and 4 virtuals using hyperthreading) are shown in the table in figure 4.1.

So we see a ngspice speed up of nearly a factor of two! Even on an older notebook with dual core processor, I have got more than 1.5x improvement using two threads. Similar results are to be expected from BSIM4.

Threads	CPU time [s]	CPU time [s]
	Windows	LINUX
1 (standard)	167	165
1 (OpenMP)	174	167
2	110	110
3	95	94-120
4	83	107
6	94	90
8	93	91

Figure 4.1: OpenMP performance

BSIM4

Ngspice implements many of the BSIM models developed by Berkley's BSIM group. BSIM stands for Berkeley Short-Channel IGFET Model and groups a class of models that is continuously updated.

This is the newest class of the BSIM family and introduces noise modeling and extrinsic parasitics. BSIM4, as the extension of BSIM3 model, addresses the MOSFET physical effects into sub-100nm regime. It is a physics-based, accurate, scalable, robust and predictive MOSFET SPICE model for circuit simulation and CMOS technology development.

4.2 System Setup

In this chapter we will give some details about Ngspice installation and the changes in the compilation system for CPU + GPU execution.

Ngspice installation

Ngspice can be obtained from the Ngspice webpage < http://ngspice.sourceforge.net/>. The installation for LINUX or MS Windows is described in the file INSTALL to be found in the top level directory.

We have used the *ngpice-24* release. By this time *ngspice-25* has also been released.

Briefly we installed Ngspice on the top level directory, issuing the commands:

- $ ./configure [options]
- $ make
- $ sudo make install

Regarding the options we used the '–enable-debug' at first to keep debugging information, but disabled it later to measure execution time, and '–enable-openmp' to measure times for

parallel execution on the multicore processor.

Ngspice Execution

After installation *ngspice-24/* is the top level directory. It contains many folders, but the one that interests us is the folder src/. This folder contains the source code of the application and the executables. The directory *ngspice-24/src/spicelib* contains three folders. Folder *parser/* includes the source code for the front end parser which reads the circuit's netlist and creates the main data structures. The folder *analysis/* includes routines for various types of analyses offered by Ngspice. Ultimately, the folder *devices/* includes the device models supported by Ngspice. It contains many folders, one for each model. For example, the directory *ngspice-24/src/spicelib/devices/bsim4* contains the source code for the evaluation of the BSIM4 model which is the one being used in this thesis.

One way to run Ngspice is the interactive mode. If we issue the command '*$ ngspice'* at the terminal ngspice will start and wait for manual input. For example:

- ngspice 1 -> source adder-mos.cir

  loads a circuit named 'adder-mod.cir'. Next

- ngspice 2 ->run

  the 'run' command will simulate the circuit

- ngspice 3 ->plot allv

  the 'plot' command will plot the selected node.

- ngspice 4 -> quit with the quit command we exit ngspice simulator

However, Ngspice can be executed as any simple executable in Linux. As mentioned above, the directory *ngspice-24/src/* contains the executable files for the simulator. The executable that interests us is the ngspice, which conducts the simulation. For example, a simulation for a netlist named add20.sp can be achieved by issuing the following command at the terminal:

```
$ ./ngspice < add20.sp > OUT
```

The results will be written in the text file 'OUT'.

Ngspice Compilation

For any change performed onto the source code, to recompile the program we need to issue a make command inside the directory *ngspice-24/src/*. This way the system recognizes the changes and recompiles the particular folders.

Ngspice has a complex compilation system based on the GNU build system, also known as Autotools. Autotools, is a suite of programming tools designed to assist in making source-code packages portable to many Unix-like systems. Autotools consists of the GNU utility programs Autoconf, Automake and Libtool. Figure 4.2 shows the Flow diagram of autoconf and automake.

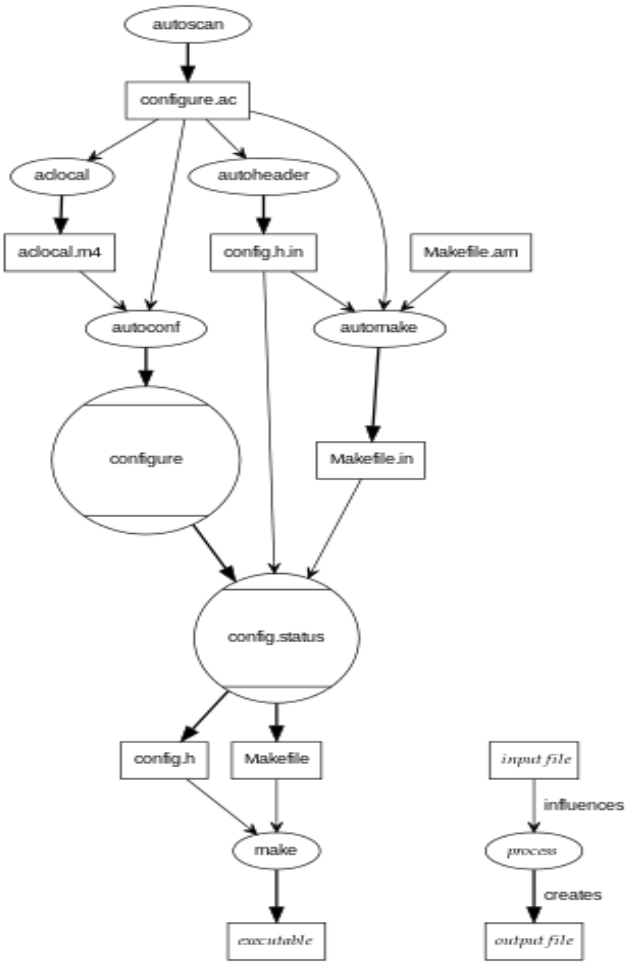


Figure 4.2: Flow diagram of autoconf and automake

Compiling CUDA code inside Ngspice

Expanding the Ngspice compilation system to compile CUDA code was a non-trivial task. As mentioned earlier we are going to deal with the BSIM4 model. So we are going to add CUDA code inside the directory *ngspice-24/src/spicelib/devices/bsim4*. This directory contains the all the .c and .h that handle the BSIM4 model evaluation along with a Makefile that compiles the source code. The Makefile was created automatically from the Makefile.in and Makefile.am files during the configuration step of Ngspice installation.

We have added a folder named GPUA inside the directory ngspice-24/src/spicelib/devices/bsim, which contains the CUDA source code. The source code consists of one file, cudabsim4.cu, which evaluates the BSIM4 models on the GPU. The header file '*gpua.h*' includes function and variable declarations. To compile *cuda_bsim4.cu* we need to invoke nvcc compiler inside the Makefile.

We have added the following inside the Makefile in the directory *ngspice-24/src/spicelib/devices/bsim4*:

```
cuda:
    nvcc -I$(AM_CPPFLAGS) -c GPUA/*.cu -o GPUA/*.o -lcudart -arch=sm_21
    ar rcs GPUA/libgpua.a GPUA/*.o
```

cuda is the target and the following two commands are the rules. The first rule compiles all the source files inside the *GPUA/* directory, but do not link. The ultimate output is in the form of an object file for each source file. This is performed via the c flag. The I flag adds the specified directory to the head of the list of directories to be searched for header files. The *-lcudart* flag imports the CUDA runtime system. The arch flag specifies the class of NVIDIA GPU architecture for which the cuda input files must be compiled. In our case we use sm_21, because our device has compute capability 2.1.

The second rule produces a static library named, *libgpua.a*, out of the object files that were created from the first rule.

To invoke the target *cuda*, we add it as a dependency to the target all, defined higher in the Makefile.

Finally, we need to link our object file (*cudabsim4.o*, which is inside the static library *libgpua.a*) with the rest of the object files, to create the final executable file. This process takes place in the top-level Makefile (the Makefile in the directory *ngspice-24/*). For this purpose we have added the *libgpua.a* among the *DYNAMIC_DEVICELIBS*.

4.3 Program Architecture

This chapter provides information about the Ngspice program architecture, the basic data structures and the basic functions that are being used.

To begin with, Ngspice's heart is the SPICE3 circuit simulator [8]. SPICE3 is designed using a toolbox approach. Each package of routines is relatively independent of every other package, thus allowing that maintain and develop the program to select routines which best fit the task from a wide range of available options. SPICE3 has been designed for easy configuration. By changing relatively small number of routines, different simulators can be produced and radical changes can be made to the behavior of the program. The interface from the simulator to the device modeling routines has been made as simple as possible to allow new devices and device models to be added to the program in a very short time and without difficulty.

By decomposing SPICE3 into modules, all the routines which handle one aspect of the problem can be grouped together but can be isolated from code which must deal with other parts of the problem. The modules identified in SPICE3 are listed in figure 4.3.

Nevertheless, SPICE3 and therefore Ngspice is a very difficult program to understand, due its large size and the complexity of its organization. To understand the structure of such a program it is necessary to break it down into modules, study them individually and finally observe how they interact. Figure 4.4 shows the basic calling structure of the program.

In this figure, the block labeled "Devices" represents all the per-device-type packages which are incorporated into the program. These packages use and are used by the numerical algorithms of SPICE3. Both the device code and the numerical routines manipulate

Command input parsing
Circuit description parsing
User interaction/batch step scheduling
Sparse matrix handling
Simulator coordination/dispatch routines and common numerical algorithms
Analyses (one package for each analysis type)
- operating point
- ac
- dc
- transient
- transfer function
- pole-zero
- sensitivity

Devices (one package for each device type)
- common support routines
- voltage and current sources (independent, voltage, and current controlled)
- resistors
- capacitors
- inductors and mutual inductors
- transmission lines
- uniform distributed R-C lines
- diodes
- bipolar junction transistors
- JFETs
- MESFETs
- voltage and current controlled switches
- level-1 MOSFETs
- level-2 MOSFETs
- level-3 MOSFETs
- BSIM MOSFETs

Graphics
- device independent
- underlying graphics system interface routines (one per graphics system)
  - X window system version 10[Gett86a]
  - X window system version 11[Sche88a]
  - Model Frame Buffer package (MFB)[Bill83a]

Figure 4.3: Major modules in SPICE3

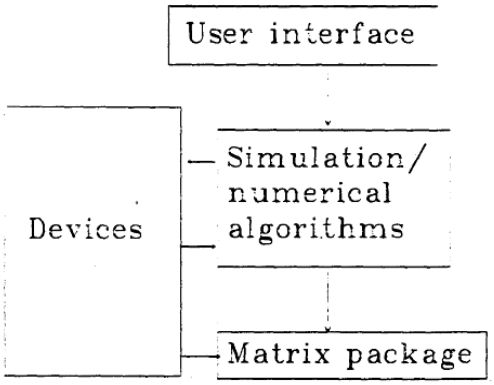


Figure 4.4: Basic Calling Structure

sparse matrix through the matrix package.

Basic Data Structures

- Circuit Structure

  The circuit package contains structures which describe a circuit as a whole, but not specific of any given device or analysis. The basic structure is the CKTcircuit, which is the primary data structure in SPICE3. It contains all variable data related to the description and operation of the circuit.

- Device Structures

  Each device type is represented by three structures describing its needs and capabilities. Ngspice contains numerous device models e.g. MOS1, MOS2, MOS3, MOS6, BSIM1, BSIM2, BSIM3, BSIM4, BSIM3soi, bjt, jfet, hfet1, hfet2. All the device models have a standard prefix that must appear in the beginning of all of them. The standard prefix is the following:

```
struct GENmodel {        /* model structure for a resistor */
    int GENmodType;             /* type index of this device type */
    GENmodel *GENnextModel;     /* pointer to next possible model in
                                 * linked list */
    GENinstance *GENinstances;  /* pointer to list of instances that have this
                                 * model */
    IFuid GENmodName;           /* pointer to character string naming this model
};


struct GENinstance {
    GENmodel *GENmodPtr;     /* backpointer to model */
    GENinstance *GENnextInstance;   /* pointer to next instance of
                                     * current model*/
    IFuid GENname;  /* pointer to character string naming this instance */
};
```

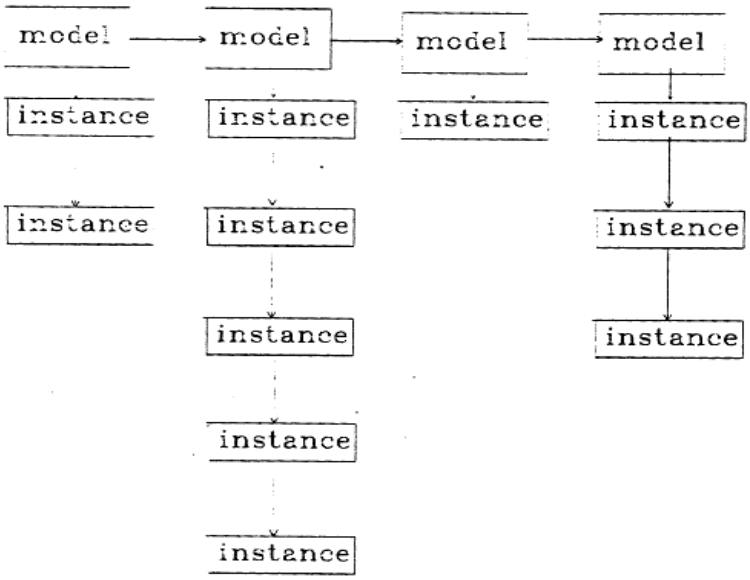


Figure 4.5: Model-Instance Data Structure

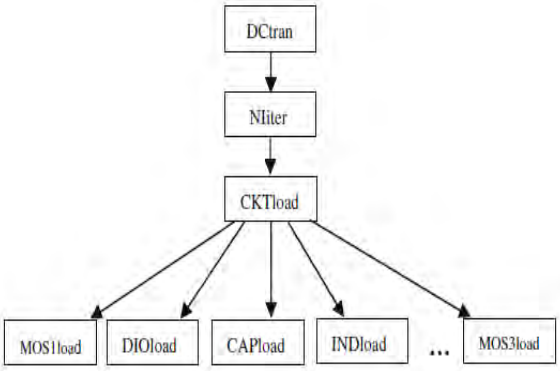


Figure 4.6: Structure of a partial callgraph for a transient simulation of SPICE3 simulator program

Each model has a pointer to the next model. Also it has a pointer to the beginning of the list of instances that have this model. Each instance has a pointer to the next instance and a back-pointer to its model. BSIM4 models are described by the same structs, with the difference that instead of ”GEN” there is ”BSIM4”. Figure 4.5 shows a general Model-Instance structure.

Ngspice Execution Flow

The type of analysis that we have dealt with is transient analysis. Transient analysis is the largest and most complicated simulation currently supported by SPICE3. The list of functions called when performing transient analysis is demonstrated in figure 4.6

- DCtran() This routines orchestrates the DC transient analysis. It mainly handles the increment of the time index and calls NIiter() to perform the analysis.

- NIiter() This routine performs the actual numerical iteration. It handles the operation that need to be done in every time step. It also tests for convergence.
- CKTload() This routine iterates through all the various load functions provided for the circuit elements in the given circuit.
- BSIM4load() This routine performs the evaluations for all the transistors in the circuit and updates the circuit matrix and the RHS vector.

Ngspice BSIM4 model evaluation

Now we will give some code details about how Ngspice performs the device evaluation for the BSIM4 model. BSIM4 models have an already implemented OpenMP version.

Serial

At first we will describe the serial BSIM4 model evaluation. *CKTload()* calls the *BSIM4load()* function (implemented inside the b4ld.c file).The following code shows the double for loop which evaluates every transistor in the circuit in serial mode for BSIM4 mdels:

```
int BSIM4load(GENmodel *inModel, CKTcircuit *ckt) {
        BSIM4model *model = (BSIM4model*) inModel;
        BSIM4instance *here;

        for (; model != NULL; model = model->BSIM4nextModel) {
                for (here = model->BSIM4instances; here != NULL;
                                here = here->BSIM4nextInstance) {

                .....

                /*
                 *  Loading RHS
                 */
                m = here->BSIM4m;
                (*(ckt->CKTrhs + here->BSIM4dNodePrime) += m
                        * (ceqjd - ceqbd + ceqgdtot - ceqdrn - ceqqd + Idtoteq));

                 .....

                /*
                 *  Loading matrix
                 */

                if (here->BSIM4rgateMod == 1) {
                        (*(here->BSIM4GEgePtr) += m * geltd);
                        .....
                }
                .....

                } /* End of MOSFET Instance */
```

```
        } /* End of Model Instance */

        return (OK);
}
```

The outer for loop traverses the linked list of the different models and the nested for loop traverses the linked list of the instances of the particular model. Inside the double for loop resides the code that evaluates the transistor parameters. For BSIM4 models this is approximately 5,000 lines of code, including a lot of computations along with numerous branches. At the end of the evaluation code, there is some code that updates the circuit matrix and the right hand side vector (RHS).

Parallelization with OpenMP

As every transistor model evaluation is an independent procedure we can harness the potential of the multicore processors to increase performance. For example, for a circuit containing 1,000 transistor, each core can evaluate 250 transistors, thus giving an significant overall speedup. At the same time, the parallelization procedure seems embarrassingly easy, as there is a simple loop to be parallelized.

However, the loop is traversing linked lists and currently it is not possible to parallelize a pointer-chasing loop by just adding an OpenMP directive. For this reason, a new data structure is introduced, the *BSIM4InstanceArray[]*. This array is included inside the struct *sBSIM4model* and stores the address of each linked-list element of an instance in an array of pointers. The variable *BSIM4InstCount* keeps track of the total number of elements in the lists. The new loop for BSIM4 model evaluation can be seen in the following code fragment. It now involves an array of pointers and an integer index instead of pointer.

```
int BSIM4load(GENmodel *inModel, CKTcircuit *ckt) {

        .....

        #pragma omp parallel for num_threads(nthreads) private(here)
        for (idx = 0; idx < model->BSIM4InstCount; idx++) {
                here = InstArray[idx];
                good = BSIM4LoadOMP(here, ckt);
        }

        BSIM4LoadRhsMat(inModel, ckt);
}

int BSIM4LoadOMP(BSIM4instance *here, CKTcircuit *ckt) {

        .....

        /* model evaluation code */

        .....
}
```

```
void BSIM4LoadRhsMat(GENmodel *inModel, CKTcircuit *ckt) {

        .....

        /* update circuit matrix and RHS vector */

        .....
}
```

Moreover, matrix and RHS update can no more be part of the code inside the loop, because more than one threads may try to write to the same location. As a result the result the update takes place in a separate function, *BSIM4LoadRhsMat()*.

More information about the OpenMP implementation of the BSIM4 models can be found in [9].

4.4 Bottlenecks for Parallelization on GPUs

Communications Bottlenecks

This is the biggest and most classical bottleneck of accelerated computing. Most of the data structures are resident on the system main CPU memory. To conduct operations on the GPU, the data must be sent back and forth to and from the card over the communication bus. Overhead to initiate the communication is best minimized by sending as much data as possible in burst mode. The overhead also includes reading the data from system memory and writing it to the GPU local memory. DMA transfers usually increases the bandwidth, but needs lots of data to show pronounced effect, plus the availability of data. Model evaluation takes place numerous times during transient analysis and each times data need to be transferred to and from the GPU.

Conditional Control Flow

SIMD architectures requires conducting the same instructions on all data points. In order to execute programs with conditional if-else statements, it can put some of the processing units in idle state till the specific elements takes alternative routes, or by doing “standby” execution of predicted branches. This is one of the major drawbacks for SPICE applications, due to their irregular and unpredictable compute structure, consisting of numerous branches.

Kernel Invocation Overheads

There is an advantage of breaking the code to very small number of kernels, ideally with 2 inputs and on output. This targets the nature of an ALU element, which has 2 inputs and one element. It also simplifies the generation of threads. However, this will add new overheads. These overheads include both the typical setup required to call the kernel, as well as overheads for fetching operands from memory. Thus, the number of kernels, the number of inputs and outputs per kernel, and size of each kernel need to be balanced. As referred

above, model evaluation can take place billion times for large circuits. Thus, for a GPU implementation the cost of launching the kernel billion times can be large.

Data Structures

CPU-oriented programs tend to have their code in the form of linked lists, using pointers and dynamic memory allocation. On the other hand, stream computing (or SIMD/vector computing in general) requires data to be arranged in contiguous compressed arrays (i.e. streams or vectors). This also helps to maximize burst communications over IO buses, as well as burst memory accesses. This is the case for SPICE applications. Data structures need to be converted to GPU friendly structures, resulting in extra overhead.

4.5 Implementation

The implementation is based upon the parallel OpenMP implementation. In general, in all the stages of the implementation the whole model evaluation code (i.e the code inside the *BSIMLoadOMP()* function) corresponds to a GPU kernel. Thus, each thread that executes the kernel represents a transistor. There are no data dependences between the evaluation of different transistors and as a result every thread is independent.

The basic routine that orchestrates the model evaluation on GPU is the *cudastartKernel()*. It is called if the option '-g' is enabled during execution. Otherwise, the parallel OpenMP version is used. The flow of the routine is the following:

1. Memory allocation: Happens only on the first execution of the function.
2. Memory transfer to device: All the required data are transferred from the CPU to the GPU.
3. Kernel invocation: The kernel is invoked with the appropriate geometry and arguments.
4. Memory transfer from device: The results are transferred back to the CPU.

Then the control flow returns back to the *BSIM4load()* and the circuit matrix and RHS vector are updated.

Stage 1

In the first version of the implementation, the whole *BSIM4InstArray* array was transferred to the GPU. As it is an array of pointers it cannot be transferred with just one *cudaMemcpy()*. Multiple *cudaMemcpy()* are needed, equal to the to the total number of the transistors in the circuit, each one transferring one *BSIM4Instance*. Another drawback is that for each *BSIM4Instance*, its corresponding *BSIM4model* is transferred with a separate *cudaMemcpy()*.

Next, the circuit is transferred to the GPU. Here, take place again multiple *cudaMemcpy()* as the *CKTcircuit* struct, contains pointers to arrays. In particular, *CKTstates* and *CKTrhsOld* are transferred first and then the rest of the circuit is transferred.

In the same way, after the completion of the kernel the data are transferred back from the GPU to the CPU.

However, the above way of moving data between the host and the device leads to very poor performance. The GPU execution for the first implementation was even 10 times slower than CPU execution.

Stage 2

To harness the available bandwidth between the host and the device it is better to integrate multiple small memory copies into a big one. This is extremely essential for the particular application, as model evaluation takes place multiple times per time-step. As result a result we end up having a vast number of *cudaMemcpy()* calls, which dominate the execution time.

In this stage we focused on merging the memory copies needed to transfer the transistor instances to the GPU. For this purpose, two new arrays were introduced, *h_InstArray* and *d_InstArray* of size *BSIM4InstCount*, for the CPU and the GPU correspondingly. At first, we use the *memcpy()* routine to transfer the transistor data to *h_InstArray* and then one *cudaMemcoy()* is used to transfer the whole *h_InstArray* to *d_InstArray*.

What is more, only one copy of each BSIM4 model is transferred to the device and latter on, inside the kernel each transistor is linked with its corresponding model. This way, we avoid moving a considerable amount of Mbytes.

The result was a dramatic performance increase between the GPU versions, but we have not managed to reach the levels of CPU execution times.

Stage 3

A profiling of the application showed that memory transfers still dominated the execution time. For example, for a circuit with 7,454 transistors, CUDA memory copies form Host to Device represented the 48.69% of the overall GPU execution time, CUDA memory copies from Device to Host represented the 45.34% and the kernel execution represented only the 5.97%.

After a careful examination of the model evaluation code we noticed that only some variables were needed from the host code to update the circuit matrix and the right hand side – RHS vector. Furthermore, the host code does not change the value of any of the *BSIM4instance* variables. As a result, there is no need in moving the whole *BSIM4InstArray* to the GPU and then back to the CPU in every model evaluation. In this stage, we transferred the *BSIM4InstArray*, only one time, at the begging of the simulation and then transferred back to the CPU, after every evaluation only the variables that were needed.

Moreover, neither the host nor the device code changes the *BSIM4model* parameters during the transient analysis. As a result, the models are transferred to the GPU memory space at the beginning of the simulation and reside there till the end of the simulation.

Another optimization implemented concerned the transfer of the circuit structure to the GPU. As the struct *CKTcircuit* contains pointers (e.g. *CKTstates*, *CKTrhsOld*), the data pointed by them was transferred with separate *cudaMemcpy()*. We copy these data in a continuous memory region and moved them with a single *cudaMemcpy()*.

These optimizations brought a huge performance increase, resulting to better execution times of the application against the CPU serial version. What is more, the kernel execution

time was approximately the same with the time needed for memory transfers.

Stage 4

This is the final optimization step. As memory transfers are almost fully optimized, in this step we tangled with kernel optimizations. As mentioned earlier, GPUs operate on a SIMD computation model, were each thread executes the same instruction on different data. On the other hand, SPICE has an irregular and unpredictable compute structure, consisting of numerous branches. In order to execute programs with conditional if-else statements, some threads may remain idle while other threads execute alternative routes.

As a result, we decided to deal with branch divergence which is one of the main bottlenecks of the SPICE program. We profiled the kernel with the NVIDIA Visual Profiler (NVVP) and in particular we used the option *Analyze Kernel -> Divergent Branch*. This option points out in the source code branches with high level of divergence.

Next, as the branches leading in high overhead were identified we used inlining of if-then-else code. For example a code fragment such as:

```
if(cond) { CODE_A; }
else { CODE_B; }
```

would be converted into the following code fragment for execution on the GPU:

```
CODE_A;
CODE_B;
if(cond) { return result of CODE_A; }
else { return result of CODE_B; }
```

This way each thread executes instructions from all the alternative paths. We do not avoid branch divergence, but instead of threads waiting other threads to finish executing multiple instructions of compute intensive code they now only wait for fewer instructions which are simple result assignments.

The result was an average speedup of 1.2x, over the previous GPU version of the application.

Chapter 5

Experimental Results

Our device model evaluation engine is implemented and integrated in an open source SPICE tool, Ngspice. In this chapter we present the result from our profiling on Ngspice and execution times for both CPU and GPU execution, using a variety of circuits.

In all our experiments, the CPU used was a quad core processor from Intel (Intel® Core™2 Quad CPU Q9300) running at 2.50GHz, with 3GB of RAM memory. The operating system we run is a linux Kubuntu 12.10 and the Kernel version 3.5.0-17-generic.

The GPU card used for our experiments is the NVIDIA GeForce GTX560 Ti, as mentioned in chapter 3.

Table 5.1 shows the characteristics of the circuits used for the experiments. The second column shows the number of transistor that each circuit contains. The third column shows the time points of the transient analysis. The last two column show the number of total model evaluations that take place for the GPU and the CPU. At first, it is noticed that the total number of model evaluations differ from the number of time points, i.e. each time step can take place more than one model evaluations. Furthermore, the total number of evaluations differ between CPU and GPU. This is due to convergence issues.

Circuit name	#Transistors	#timePoints	#evaluations GPU	#evaluations CPU
add20	958	498	1573	1573
mem_plus	7454	1123	4245	4240
ram2k	13880	2006	7749	7741
voter	4244	2825	27720	12889

Table 5.1: Circuits

5.1 Profiling

This section presents the results of our profiling on the Ngspice application, provided by the Intel VTune Amplifier. The results showed that for transient analysis of a circuit dominated by non-linear devices, such as transistors, three are the main time consuming tasks: Evaluating the transistors, Updating the circuit matrix and the RHS vector and Sparse matrix factorization. These tasks are represented by the functions *BSIM4load()*, *BSIM4loadRHSMat()* and *spFactor()*

Table 5.2 shows the time in seconds that each task consumes, and in parentheses the percentage of this time to the total execution time.

Circuit name	Total runtime	BSIM4load (%)	BSIM4loadRHSMat (%)	spFactor (%)
add20	13.98	9.25 (66.10%)	1.12 (8.04%)	1.02 (7.29%)
mem_plus	278.23	192.773 (69.28%)	19.2 (6.9%)	25.95 (9.32%)
ram2k	881.239	664.16 (75.36%)	66.74 (7.57%)	56.7 (6.43%)
voter	438.26	321.28 (73.3%)	35.36 (8.06%)	33.6 (7.66%)

Table 5.2: Ngspice Profiling

Consequently, Table 5.3, shows the average percentage of the total execution time that each task represents, for the 4 circuits.

Task	Percentage
BSIM4load	71.03%
BSIM4loadRHSMat	6.64%
spFactor	7.67%

Table 5.3: Average

5.2 Performance Analysis

This section provides information about the performance of the application. It presents and compares execution times for both CPU and GPU execution. All the execution times are measured in seconds

CPU Execution Times

Table 5.4 shows the execution times for the serial version and for the OpenMP version using 1, 2 and 4 threads. Table 5.5 shows the speedups obtained by the OpenMP version and the graph in figure 5.1 shows how the execution time evolves relatively to the number of threads that are active.

Circuit	Serial	1 thread	2 threads	4 threads
add20	12.76	13.55	9.20	7.81
mem_plus	260.80	288.67	187.61	142.95
ram2k	814.33	874.79	551.15	396.38
voter	406.87	435.84	280.75	216.83

Table 5.4: CPU execution times

Ngspice runs approximately 2 times faster with OpenMP enabled and 4 threads activated. It is also noticed that the performance of the OpenMP version using 1 thread is worse than the serial version. This is due the fact that each version uses 1 thread but the OpenMP version has the extra overhead of the OpenMP runtime system.

Circuit	1 thread	2 threads	4 threads
add20	0.94	1.38	1.63
mem_plus	0.903	1.40	1.82
ram2k	0.93	1.47	2.05
voter	0.93	1.44	1.87

Table 5.5: OpenMP Speedups

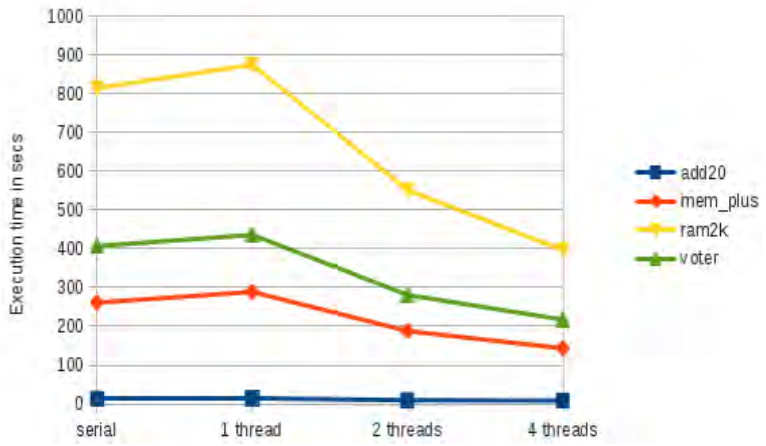


Figure 5.1: OpenMP execution times

GPU Execution Times

Table 5.6 shows the execution times of Ngspice when the model evaluation phase is performed on the GPU. Also it shows the speedups obtained over the serial CPU version. Execution times between CPU and GPU execution are presented graphically by the bar chart in figure 5.2.

Circuit	Time	Speedup
add20	10.54	1.21
mem_plus	165.52	1.57
ram2k	443.87	1.83
voter	540.31	0.75

Table 5.6: GPU version execution times with memory transfers

Table 5.7 shows again the GPU version execution times, but now without counting the time needed to transfer data between the CPU and GPU.
Finally, figure 5.3 shows the speedups obtained when counting communication overheads and when not.
The maximum speedup obtained was 1.83x when counting the overhead of the memory transfers and almost 3x when memory transfers where excluded from the total execution time. It is noticed that the circuit *voter* has worse performance when simulated using the GPU version than the CPU version. This is because in the GPU version, model evaluation is invoked

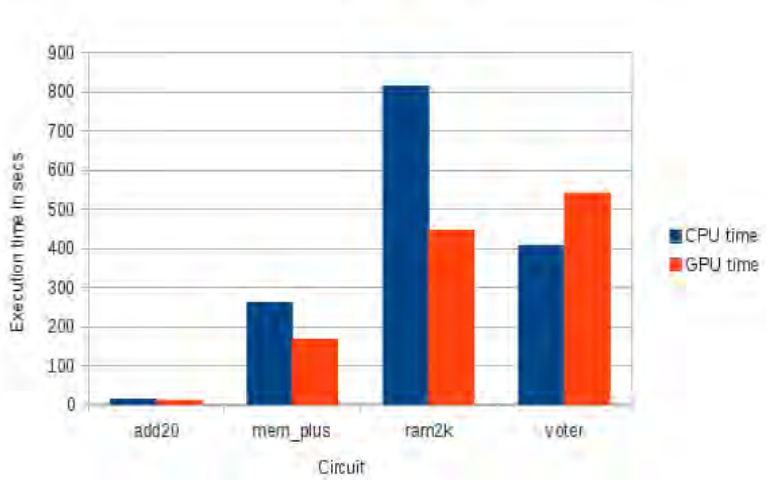


Figure 5.2: CPU and GPU execution times

Circuit	Time	Speedup
add20	6.40	2.00
mem_plus	107.453	2.42
ram2k	273.21	2.98
voter	325.64	1.25

Table 5.7: GPU version execution times without memory transfers

more than double the times it is invoked by the CPU version. As a result the overheads (communication ,kernel invocation) dominate the execution time.

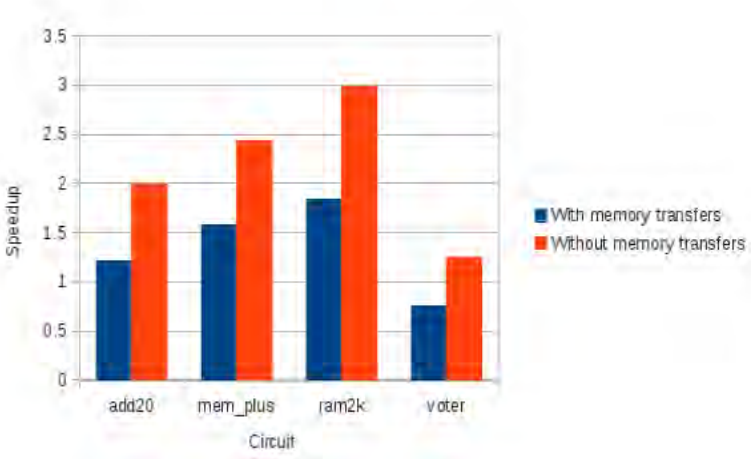


Figure 5.3: Comparison of speedups when counting the memory transfers in measurements and when excluding them

Chapter 6

Conclusions

Given the key role of SPICE in the design process, there has been a significant interest in accelerating SPICE. A large fraction of SPICE runtime (over 70% on average) is spent in evaluating transistor model equations. This thesis reports our efforts to accelerate transistor model evaluations using modern GPUs. We have integrated our transistor model evaluation accelerator into an open source SPICE tool, Ngspice and have shown maximum speedups of 1.83x counting memory transfers and approximately 3x without counting memory transfers.

6.1 Future Extensions

Possible extensions in this project may be the following:

- **Memory coalescing**
  In the CUDA programming model it is essential for threads inside a warp to access adjacent memory addresses in global memory. This is called coalesced memory accesses. For example, one modification that may bring extra performance is to convert the array of data structures that describe transistors, from array of structs to stuct of arrays.

- **Further minimize communication overhead**
  In the current implementation there are 3 *cudaMemcpy()* happening before the kernel invocation and 4 *cudaMemcpy()* happening after, except the first time the kernel is invoked. These *cudaMemcpy()* may be further reduced, as it is better to have one big *cudaMemcpy()*, than multiple small.

- **Port the entire simulation on the GPU**
  This way memory transfers between CPU and GPU will be eliminated. One effort to implement such a system is described in [4].

Chapter 7

Appendix

7.1 Comprehensive Experimental Results

In this section we present the experimental results of every implementation stage. In particular, each one of the following tables presents the execution time of the GPU version, for the 4 circuits used for the experiments. The speedup for each stage is measured over the previous stage.

Circuit	Time
add20	101.90
mem_plus	2148.53
ram2k	7052.90
voter	3577.55

Table 7.1: Stage 1

Circuit	Time	Speedup
add20	27.10	3.76
mem_plus	512.83	4.19
ram2k	1624.02	4.34
voter	964.85	1.97

Table 7.2: Stage 2

Circuit	Time	Speedup
add20	15.096	2.01
mem_plus	272.08	2.71
ram2k	807.50	3.12
voter	964.85	2.80

Table 7.3: Stage 3

Circuit	Time	Speedup
add20	13.43	1.27
mem_plus	188.73	1.16
ram2k	519.69	1.17
voter	645.93	1.23

Table 7.4: Stage 4

Bibliography

[1] http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[2] http://ngspice.sourceforge.net/.

[3] http://www-device.eecs.berkeley.edu/bsim/?page=bsim4.

[4] Lengfei Han, Xueqian Zhao, and Zhuo Feng. Tinyspice: a parallel spice simulator on gpu for massively repeated small circuit simulations. In *DAC*, page 89, 2013.

[5] Chung-Wen Ho, Albert E. Ruehli, and Pierce A. Brennan. The modified nodal approach to network analysis. In *IEEE Transactions on Circuits and Systems*, pages 504–509. IEEE Computer Society, 1975.

[6] K. S. Kundert and A. Sangiovanni-Vincentelli. Sparse user's guide: A sparse linear equation solver,. 1988.

[7] Laurence W. Nagel and D.O. Pederson. Spice (simulation program with integrated circuit emphasis). Technical Report UCB/ERL M382, EECS Department, University of California, Berkeley, Apr 1973.

[8] Thomas L. Quarles. *Analysis of Performance and Convergence Issues for Circuit Simulation*. PhD thesis, EECS Department, University of California, Berkeley, 1989.

[9] Tien-Hsiung Weng, Ruey-Kuen Perng, and Barbara Chapman. Openmp implementation of spice3 circuit simulator. In *International Journal of Parallel Programming*, pages 493–505, October 2007.