Accelerated Circuit Simulation: Harmonic Balance and Logic Partitioning

A Dissertation

Submitted in partial fulfillment of the requirements

for the degree of

Bachelor and Master of Technology

by

Shashank Vijayakumar Obla

(Roll No. 14D070021)

Under the guidance of

Prof. Sachin B. Patkar

Department of Electrical Engineering

Indian Institute of Technology Bombay

May 2019

Dissertation Approval

This Dissertation entitled "Accelerated Circuit Simulation: Harmonic Balance and Logic Partitioning" by Shashank Vijayakumar Obla is approved for the degree of Dual Degree in Electrical Engineering with Specialization in Microelectronics.

____

____

____

Examiners

____

____

____

Supervisor (s)

____

Chairman

Date: ____________

Place: ____________

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I declare that I have properly and accurately acknowledged all sources used in the production of this report. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Shashank Vijayakumar Obla
(14D070021)

Date: June 26, 2019

Abstract

The tremendous technology scaling over the past decade has allowed chip designers to accomplish ultra-large scale integration and develop multi-functional chips using the same die area. Along with the opportunity to build more complex designs, this also increases pressure on the supporting software and CAD tools used in the simulation and testing of these circuits.

This work mainly focuses on the Harmonic Balance algorithm for Periodic Steady State simulations which are mainly used in Microwave and RF circuits analysis which are dominated by non-linear devices such as MOSFETs. Though commercial tools implement the algorithm, to our knowledge no tool implements the algorithm in the public-domain, which inhibits research from focusing on acceleration. This work contributes a fully automated harmonic balance simulator which is able to simulate a circuit supplied in a format similar to a SPICE netlist, with optimizations for parallelization of the sparse matrix solution, and analysis of causes of numerical instabilities. This thesis also proposes an acceleration the algorithm on Intel's Many-core architecture: Xeon Phi Knights Landing edition, and an analysis on feasibility of its implementation on an FPGA as an accelerator, which span low-level architecture-aware optimizations to high-level integration with tools such as SPICE.

Different techniques of logic partitioning of distributed logic simulation of large digital circuits have also been explored. Methods involving combinatorial optimization using Sub-modular function optimization as a dual to convex optimization, and eigenvalue based Spectral Partitioning graph algorithm have been implemented and their efficacy compared based on several metrics: Replication, Communication and Balance among partitions.

Contents

Abstract iii

List of Figures vii

List of Tables ix

I A Harmonic Balance Simulator 1

1 Introduction 2

2 Previous Work and Motivation 3

3 The Harmonic Balance Formulation 4

3.1 Charge-Oriented Modified Nodal Analysis 4

3.2 Discrete Fourier Transformation 5

3.3 Harmonic Balance for Non-Linear Circuits 5

3.3.1 Newton-Raphson Interations 6

4 Non-Linear Function Evaluation 9

4.1 Model Requirements 9

4.2 ngspice Shared Library [1] 10

4.3 Simple Square-Law Model 10

4.3.1 Discontinuity in the Model 12

4.4 Implementation 13

5 Block-Aware Sparse LU Decomposition 15

5.1 Left-Looking LU Decomposition 15

5.2 Sparse Matrix Storage Schemes 17

5.3 Symbolic LU Decomposition 18

5.3.1 Fill - In 19

5.3.2 Row Pivoting 20

5.3.3 Column Dependency Analysis 21

5.4 Dense LU Decomposition 23

6 Automatic Circuit Parser 24

6.1 Matrix Construction 24

6.2 Syntax 25

6.3 Example 26

7 Custom Hardware Implementation 27

7.1 HLS Hardware Optimization 28

7.1.1 Function Instantiate 28

7.1.2 Array Partitioning 28

7.1.3 Pipelining 30

7.2 Data Movement Optimizations 31

7.2.1 Data Zero Copy 31

7.2.2 Data Access Pattern - Sequential 31

8 Results and Discussions 32

8.1 Fill - In Reduction 32

8.2 Numerical Instability 32

8.3 Simulation Setup 33

8.3.1 Intel Xeon Phi: Knights Landing Edition 33

8.3.2 Xilinx ZedBoard: Zynq-7000 SoC 35

8.4 Dense LU Decomposition 35

8.5 FPGA Implementation 36

8.5.1 Performance Comparison 36

8.5.2 Timing Closure 37

8.6 Simulation Results 38

8.6.1 Example: Operational Transconductance Amplifier 38

8.6.2 Output 39

8.6.3 Performance Analysis 39

9 Future Work 42

II Circuit Partitioning for Distributed Logic Simulation 43

10 Introduction 44

11 Previous Work and Motivation 45

12 Sub-Modular Function based Minimization **47**
12.1 Sub-Modular Set Functions 47
12.2 Matlab SFO Toolbox [2] 47
12.2.1 Queyranne's Algorithm 48
12.2.2 The Oracle Function 48

13 Ratio-Cut using Spectral Partitioning **49**
13.1 Spectral Graph Theory 49
13.2 Circuit Conditioning 51
13.2.1 Replication 51
13.2.2 Communication 51
13.2.3 Combining the Costs 52

14 Results and Discussion **53**
14.1 MATLAB interface with C++ 53
14.2 Greedy Splitting using SFO Toolbox 54
14.3 Spectral Partitioning 54
14.4 Spectral Partitioning with Communication 56

15 Future Work **57**

III Appendix **58**

I Harmonic Balance C++ Codes **59**
I.1 Sparse Matrix Storage 59
I.1.1 sparseCOO.h 59
I.1.2 sparseCSC.h 60
I.1.3 sparseBCCS.h 60
I.2 Circuit Parser and Evaluation 61
I.2.1 ngspice.h 61
I.2.2 device.h 62
I.2.3 circuit.h 66
I.3 Extracting Column Parallelism 68
I.3.1 Plotting Column Dependency Graph 68
I.3.2 Get Next Column 69

II Circuit Partitioning C++ Codes **70**
II.1 dcircuit.h 70

III BSIM4 Model **73**

List of Figures

3.1 Block Diagram [3] of the required steps to compute $F(X)$ 7
3.2 Harmonic Balance Jacobian matrix structure [4] 8

4.1 Output current waveform showing glitch due to discontinuity in the model 12
4.2 Flowchart showing usage of ngSPICE shared library 13

5.1 Block Matrix structure when computing Block Column j [5] 15
5.2 Tri-Diagonal Banded Matrix 18
5.3 Example Symbolic Decomposition 19
5.4 Example Circuit matrix showing extensive Fill-In after LU Decomposition 20
5.5 Matrices showing effect of COLAMD and Pivoting on diagonal elements . 20
5.6 Example Matrix and its U factor 21
5.7 Column Dependency Graph for Matrix in Figure 5.6 22

6.1 J Matrix of the OTA with parasitics generated from the Circuit Parser . 26

7.1 Types of Array Partitioning 29
7.2 Matrix Multiplication showing computation of a single element of Matrix C 29
7.3 Example program showing effects and benefits of pipelining the program [6] 30

8.1 Intel Xeon Phi: Knights Landing Architecture [7] 34
8.2 Dense Tiled LU Decomposition Comparison 36
8.3 Speedup of the BLAS algorithms on the FPGA over CPU 36
8.4 Speedup of the LAPACK algorithms on the FPGA with block size of 64 . 37
8.5 Operational Transconductance Amplifier Schematic 38
8.6 Output waveforms of the OTA for different input voltage amplitudes . . 39
8.7 Speedup of extracting Column-Level Parallelism normalized to serial case 40
8.8 Performance degradation with increasing harmonics normalized to 20 Harmonics 40
8.9 Performance degradation with using ngSPICE calls for Model Evaluation 41
8.10 Performance Profiling results with the two different Model Evaluations . 41

11.1 Example of Partitioning and Minimization Problem 46

11.2 Example of Partitioning and Communication Problem 46

13.1 Modes of a Vibrating String [8] 50
13.2 Example Graph of s526 circuit from ISCAS'85 benchmarks representing input to Spectral Partitioning embedding Replication metric information 51
13.3 Example Graph of s713 circuit from ISCAS'85 benchmarks representing input to Spectral Partitioning embedding Communication metric information 52

14.1 Example Circuit c432 showing lack of good Balanced Partitions 55

List of Tables

5.1 Levelized List of nodes based on the Example Matrix 22

6.1 Device description and synatx for Circuit Parser 25

8.1 Fill-In w/ and w/o COLAMD column permutations for Sparse Matrices . 32
8.2 Xilinx ZedBoard Frequency Specifications 35
8.3 Matrix sizes and No. of Non-Linear Elements in circuits used to analyze performance 39

14.1 Comparison of replication metric of Partitioning Algorithms on ISCAS Combinational Benchmarks 54
14.2 Comparison of Replication and Balance metric of Partitioning Algorithms on ISCAS Combinational Benchmarks 55
14.3 Comparison of All metrics of the Spectral Partitioning Algorithm on ISCAS Sequential Benchmarks 56

III.1 Operating Point Parameters available in BSIM4 Models 73

Part I

A Harmonic Balance Simulator

Chapter 1

Introduction

Though the aggressive technology scaling has been fueled by the increasing demands of digital circuit design, CMOS analog circuits have also seen a huge growth with the advent of SoC design. This also brings new challenges such as mixed-mode simulation, fast simulation of circuits with large number of non-linear components and simulation of post-layout extracted circuits which are performance and memory demanding applications.

In Microwave and RF Circuits, there is often a requirement for steady-state analysis for periodic circuits such as mixers. But, these circuits have a large number of non-linear components, which makes traditional periodic steady-state simulation less feasible. In the Harmonic Balance (HB) method, the circuit equations are formulated in the frequency domain using fourier coefficients. For non-linear circuits, the matrices are formed by evaluation of the non-linear function in the time-domain followed by Fourier transforms. This method produces the Jacobian matrices of size very much dependent on the number of non-linear elements and which are strongly Block Sparse in nature making Block Sparse matrix solution ideal for the LU decomposition step.

On another note, Intel Xeon Phi Knights Landing (KNL) is a new Many-Integrated Core (MIC) architecture from Intel, which is becoming more popular in present-day High-Performance systems. It is a 64 core system which features SIMD instruction execution capability of 512 Bits using the Advanced Vector Extensions (AVX) of Intel. It also has the ability for fast all-to-all communication based on a mesh routed Network-on-Chip and being a full-fledged processor, has definite advantages over GPUs and other co-processor based systems by bridging the data transfer and synchronization barrier.

Chapter 2

Previous Work and Motivation

The general harmonic balance algorithm which is a mathematical technique and has a wide variety of applications has been worked out in literature such as [9]. Other work such as [3][10] show implementation of the matrix solution part of the algorithm on platforms such as GPUs. They discuss the Harmonic Balance for non-linear circuits and also for multi-tone excitation (with multiple input frequencies which are not harmonics of each other). Commercial tools have been quick to implement the algorithm given its promising benefits, but in the open-source community, there is no existing full fledged implementation to our knowledge. Some tools exist such as [11] which target at implementing Harmonic Balance in the future.

But open problems still existing are to develop faster and accelerated implementation of the kernels on different architectures such as the Many-Integrated Core architecture. The lack of a readily available simulator makes the entry into the research problem difficult. Also, none of the existing implementations or research work mention about handling non-linear capacitance which comes with non-linear devices such as MOSFETs which are ubiquitous in the current CMOS regime for analog circuit design.

The report is organized as follows. Chapter 3 introduces the Harmonic Balance technique and the entire flow involved in implementing it. The basis of the acceleration opportunities is also developed in that Chapter. The succeeding chapters talk about individual components of the algorithm in more detail such as the non-linear function evaluation, circuit parsing, the matrix solution phase and the FPGA implementation of the core functions. This is followed by the Chapter on Results and Discussions which contains results of the different algorithms, heuristics and techniques used. Future Work discusses different directions in which the work can be extended. Finally the Appendix section contains a subset of the codes which are a part of the simulator to assist in understanding the report.

Chapter 3

The Harmonic Balance Formulation

In this section, a basic discussion about the existing method is provided along with some detailed analysis which forms my contribution to the existing pool of work in this field.

3.1 Charge-Oriented Modified Nodal Analysis

Modified Nodal analysis is the basic algorithmic way to obtain the circuit equations as a function of the node voltages and some edge currents (mainly Inductor currents). In a basic form the equation can be written as Equation 3.1

$$G \cdot x(t) + E \cdot \frac{d}{dt}(x(t)) + f(x(t)) + \frac{d}{dt}(E_d(x(t)) \cdot x(t)) = b(t) \tag{3.1}$$

where G is the conductance matrix which includes resistance and transconductance elements, E includes the linear capacitances and inductances, $f(x(t))$ are the non-linear component equations, E_d includes the non-linear capcitances and inductances and $b(t)$ are the independent sources.

The issue with the above formulation is that usually the non-linear capacitances and inductances equations are not available and a more natural way to represent them is using charges or fluxes. For the purpose of this report, only non-linear capacitances are considered because inductances are not significant in all major non-linear devices and not modeled hence. More detailed analysis is presented in [12]

$$q = q_C(x(t)) \tag{3.2}$$

$$\frac{d}{dt}(E_d(x(t)) \cdot x(t)) = \frac{d}{dt}q(x(t)) \tag{3.3}$$

where q are the terminal charges for each non-linear device.

3.2 Discrete Fourier Transformation

As mentioned earlier, the Harmonic Balance method deals in frequency domain representation of the variables, a method is need to transform from time domain to frequency domain for the evaluation of non-linear circuits. This is achieved by taking advantage of the possibility of performing matrix operations by formulating the DFT as a matrix multiplication. For the rest of the report, the number of harmonics is taken to be M. This is an approximation because a non-linear function in theory creates infinite harmonics, but it can be shown that the number can be limited without much loss of accuracy for weak non-linear behaviour based on Fourier Analysis. It is important to note that, for circuits exhibiting switching, the harmonic balance matrices can be very large due to the requirement of a large number of harmonics to accurately represent step-like functions.

$$x = \Gamma^{-1} X \tag{3.4}$$

$$\Gamma^{-1} = \begin{bmatrix} 1 & \cos\omega_0 t_0 & \sin\omega_0 t_0 & \cdots & \cos M\omega_0 t_0 & \sin M\omega_0 t_0 \\ 1 & \cos\omega_0 t_1 & \sin\omega_0 t_1 & \cdots & \cos M\omega_0 t_1 & \sin M\omega_0 t_1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & \cos\omega_0 t_{2M} & \sin\omega_0 t_{2M} & \cdots & \cos M\omega_0 t_{2M} & \sin M\omega_0 t_{2M} \end{bmatrix} \tag{3.5}$$

3.3 Harmonic Balance for Non-Linear Circuits

$$\begin{aligned} G\left(X_0 + \sum_{m=1}^{M}\left(X_m^C \cos m\omega_0 t + X_m^S \sin m\omega_0 t\right)\right) + \\ C\left(X_0 + \sum_{m=1}^{M}\left(-m\omega_0 X_m^C \sin m\omega_0 t + m\omega_0 X_m^S \cos m\omega_0 t\right)\right) + \\ \left(F_0 + \sum_{m=1}^{M}\left(F_m^C \cos m\omega_0 t + F_m^S \sin m\omega_0 t\right)\right) + \\ \left(Q_0 + \sum_{m=1}^{M}\left(-m\omega_0 Q_m^C \sin m\omega_0 t + m\omega_0 Q_m^S \cos m\omega_0 t\right)\right) \\ = \left(B_0 + \sum_{m=1}^{M}\left(B_m^C \cos m\omega_0 t + B_m^S \sin m\omega_0 t\right)\right) \end{aligned} \tag{3.6}$$

Without going into detailed derivation which are discussed in [3] and [10], here the time-domain expressions are replaced by the Fourier coefficients to obtain Equation 3.6.

The equation 3.6 is represented in the matrix form with the non-linear components showing up as column vectors as in Equation 3.7 where Y includes the linear elements contributions, $F(X)$ and $Q'(X)$ are the non-linear elements and $B(X)$ are the independent sources.

$$Y \times X + F(X) + Q'(X) = B \tag{3.7}$$

The differentiation transformation of the non-linear charges can be represented by a matrix. Each element in the vector shown is itself a vector whose length is the number of equations in the original time-domain MNA. But for the purpose of illustration and simple formation fo the matrix Λ, they are considered to be scalars. Solving the set of equations as described in 3.7 can be solved using the Newton Raphson method.

$$Q(X) = \begin{bmatrix} Q_0(X) \\ Q_1^C(X) \\ Q_1^S(X) \\ \vdots \\ Q_M^C(X) \\ Q_M^S(X) \end{bmatrix} \tag{3.8}$$

$$Q'(X) = \Lambda \times Q(X) = \begin{bmatrix} 1 & 0 & 0 & & 0 & 0 \\ 0 & 0 & \omega_0 & \cdots & 0 & 0 \\ 0 & -\omega_0 & 0 & & 0 & 0 \\ & \vdots & & \ddots & \vdots & \\ 0 & 0 & 0 & \cdots & 0 & M\omega_0 \\ 0 & 0 & 0 & & -M\omega_0 & 0 \end{bmatrix} Q(X) \tag{3.9}$$

3.3.1 Newton-Raphson Interations

Skipping the details of Newton-Raphson(NR) iterations, here the formation of the vector and its Jacobian is focused on. At the core of the NR loop is the error vector and the Jacobian matrix which is the differential of the error as in Equation 3.10

$$\Phi(X) = Y \times X + F(X) + Q'(X) - B \tag{3.10}$$

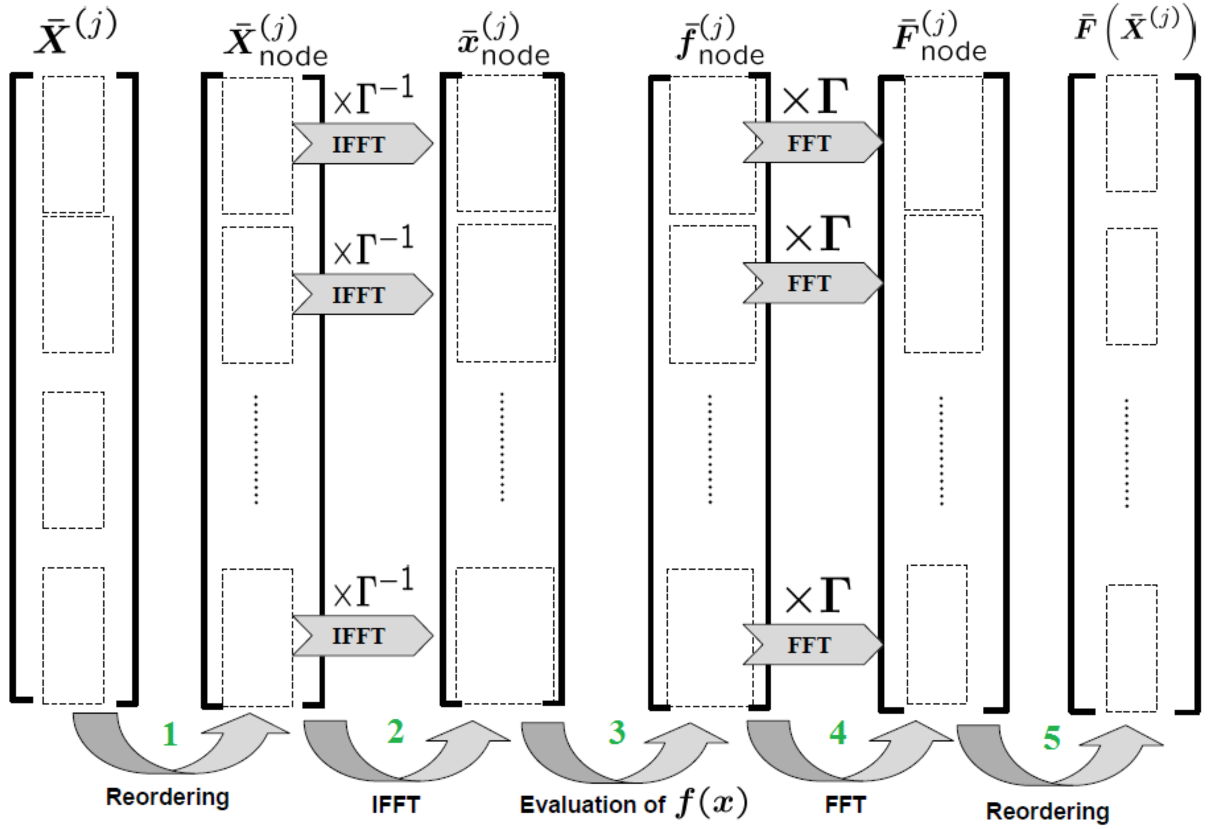

Figure 3.1: Block Diagram [3] of the required steps to compute $F(X)$

At each step of the NR iterations the value of $F(X)$ and $Q'(X)$ is obtained as follows and is illustrated in 3.1 (The reordering is an implementation consideration and more details can be obtained in [3]):

1. X in the frequency domain is converted to $x(t)$ using the Γ^{-1} matrix as described earlier

2. The non-linear functions $f(x(t))$ and $q(x(t))$ are evaluated based on the device models

3. $f(x(t))$ and $q(x(t))$ are back converted to the frequency domain using the Γ matrix to obtain $F(X)$ and $Q(X)$ and consequently using Λ matrix, $Q'(X)$

The Jacobian matrix now is given by

$$\frac{d}{dX}\Phi(X) = Y + \frac{d}{dX}F(X) + \frac{d}{dX}Q'(X) \tag{3.11}$$

Based on the Jacobian in Equation 3.11, a problem arises. Because $F(X)$ and $Q'(X)$ are not available as a function which can be differentiated in the frequency domain, it is necessary to again take assistance of the DFT and IDFT matrices and perform the differentiation in the time domain.

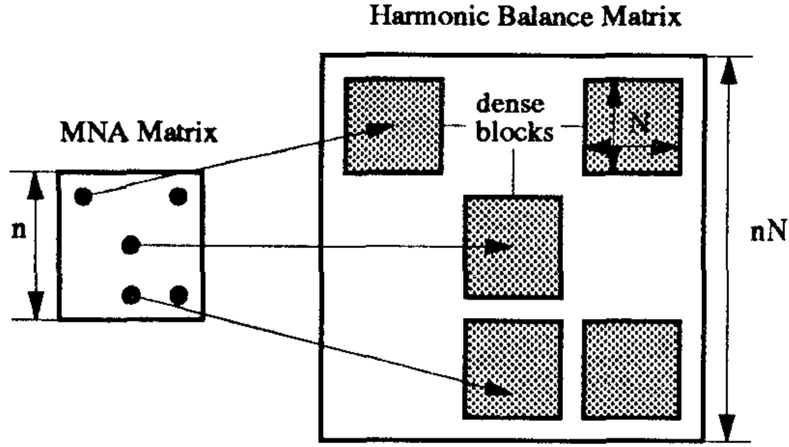

Figure 3.2: Harmonic Balance Jacobian matrix structure [4]

The partial differential 3.12 can be represented as a block matrix when N is the number of MNA equations.

$$\frac{\partial}{\partial X}F(X) = \begin{bmatrix} \Gamma\frac{\partial f_1}{\partial x_1}\Gamma^{-1} & \cdots & \Gamma\frac{\partial f_1}{\partial x_N}\Gamma^{-1} \\ \vdots & \ddots & \vdots \\ \Gamma\frac{\partial f_N}{\partial x_1}\Gamma^{-1} & \cdots & \Gamma\frac{\partial f_N}{\partial x_N}\Gamma^{-1} \end{bmatrix} \tag{3.12}$$

Each block of this element can be further written as in Equation 3.13 which is a diagonal matrix. This completes the discussion on the Jacobian matrix and hence from this, the structure of the matrix is obtained as shown in Figure 3.2.

$$\frac{\partial f_i}{\partial x_j} = \begin{bmatrix} \frac{\partial f_i(x_j(t_0))}{\partial x_j(t_0)} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{\partial f_i(x_j(t_{2M}))}{\partial x_j(t_{2M})} \end{bmatrix} \tag{3.13}$$

The blocked structure of the matrix clear shows its affinity to sparse matrix operations and more specifically using the Block Sparse storage format. The size of the blocks is given by 2M + 1 where M is the number of harmonics chosen and hence can be very large especially in the case of 2-tone input where the number combined harmonics blow up.

Chapter 4

Non-Linear Function Evaluation

As described in the procedure earlier in Step 2 of 3.3.1, at every iteration, to obtain the error function and also the Jacobian it is necessary to evaluate the non-linear elements of the circuit. For devices such as diodes, it is easy to obtain a closed form expression for the IV relation as well its differential. But when it comes to MOSFETs and BJTs it is difficult to get accurate results using simple closed form expressions especially with the aggressive changes in the technology every coming year and development of More-than-Moore [13] technologies hand-crafted for areas such as RF and sensors.

For this purpose, it was necessary to look for the ability to utilise the industry standard model files such as BSIM4, etc. which are detailed enough for accurate simulations. For this purpose, in this section I describe using ngspice simulator as the interface to communicate with the BSIM [14] MOSFET models. Though it is not the fasted choice, but being open-source and the available documentation and community make using it easy.

Though the evaluation corresponds to time-domain evaluation of the devices, the resulting functions are independent of time and hence the function evaluations reduce to DC analysis. A transient or periodic steady-state analysis(pss) to obtain these values would be counter-productive because through Harmonic Balance, pss analysis is what is trying to be achieved.

4.1 Model Requirements

In this section, the requirements to simulate MOSFETs using the aforementioned Harmonic Balance is discussed. They can be listed as follows:

1. Non-Linear Function Evaluation: $f(x(t))$ and $q(x(t))$

(a) $f(x(t))$: MOSFET exhaustive IV Relationship

(b) $q(x(t))$: MOSFET exhaustive terminal charge relationship

2. Differential of the non-linear functions with respect every node potential: $f'(x(t))$ and $q'(x(t))$

(a) $f'(x(t))$: MOSFET conductances

(b) $q'(x(t))$: Small Signal capacitances

4.2 ngspice Shared Library [1]

Because performance is of importance, the communication with the BSIM Models can become a bottleneck. Hence, externally initializing the SPICE program and communicating through files is not a suitable option. To hasten the link, the ngspice shared library is used.

To extract the values of the dc operating points through the shared library it is necessary to build a vector out of them. The `compose` command in ngspice comes in handy. An example is shown below:

```
compose cg values @mnmos[cgd] @mnmos[cgg] @mnmos[cgs] @mnmos[cgb]
```

This line creates a vector cg containing all the gate charge related differentials. And in this fashion all the values can be extracted. This vector does not prompt a call in the ngspice API in C++ and hence must be accessed asynchronously. For this purpose, the following function provided by the shared library can be used.

```
pvector_info ngGet_Vec_Info(char*)
```

More information about the API can be found in the ngspice User Manual[1]. It requires a build of the software from source to create the shared library and provides interface functions to send and receive data from the API.

There is also a possibility of parallelism, as noted earlier: the Model Evaluation phase operations can be highly independent of each other. The shared library does offer some support but is incomplete and its usage questionable.

4.3 Simple Square-Law Model

To take small steps into the algorithm, first the model evaluation to kept at the bare minimum and hence the charge-oriented model and the ngspice integration were omitted.

The following equations were used for NMOS and PMOS evaluations. Sub-threshold was assumed to be entirely non-conducting.

NMOS in Linear Region:

$$I_D = \mu C_{ox}\frac{W}{L}\left((V_{GS} - V_{TH})V_{DS} - \frac{1}{2}V_{DS}^2\right) \tag{4.1}$$

$$g_m = \mu C_{ox}\frac{W}{L}V_{DS} \tag{4.2}$$

$$g_{ds} = \mu C_{ox}\frac{W}{L}\left((V_{GS} - V_{TH}) - V_{DS}\right) \tag{4.3}$$

NMOS in Saturation Region:

$$I_D = \mu C_{ox}\frac{W}{2L}(V_{GS} - V_{TH})^2\left(1 + \lambda_0\frac{L_0}{L}V_{DS}\right) \tag{4.4}$$

$$g_m = \mu C_{ox}\frac{W}{L}(V_{GS} - V_{TH}) \tag{4.5}$$

$$g_{ds} = \mu C_{ox}\frac{W}{2L}(V_{GS} - V_{TH})^2\left(\lambda_0\frac{L_0}{L}\right) \tag{4.6}$$

It can be clearly seen from the above set of equations that there is a discontinuity in the g_{ds} parameter when the regions switch. This discontinuity has been removed from g_m to prevent oscillations causing convergence issues.

PMOS in Linear Region:

$$I_D = -\mu C_{ox}\frac{W}{L}\left((V_{SG} - V_{TH})V_{SD} - \frac{1}{2}V_{SD}^2\right) \tag{4.7}$$

$$g_m = \mu C_{ox}\frac{W}{L}V_{SD} \tag{4.8}$$

$$g_{ds} = \mu C_{ox}\frac{W}{L}\left((V_{SG} - V_{TH}) - V_{SD}\right) \tag{4.9}$$

PMOS in Saturation Region:

$$I_D = -\mu C_{ox}\frac{W}{2L}(V_{SG} - V_{TH})^2\left(1 + \lambda_0\frac{L_0}{L}V_{SD}\right) \tag{4.10}$$

$$g_m = \mu C_{ox}\frac{W}{L}(V_{SG} - V_{TH}) \tag{4.11}$$

$$g_{ds} = \mu C_{ox}\frac{W}{2L}(V_{SG} - V_{TH})^2\left(\lambda_0\frac{L_0}{L}\right) \tag{4.12}$$

4.3.1 Discontinuity in the Model

Thought the simple square-law model is very lucrative and can speed up the process significantly there is an issue of discontinuity in the model between the linear and the saturation region. This occurs because of the channel-length modulation effect added to the saturation equation to introduce the drain voltage dependent current of the MOSFET in the Saturation region.

This does not cause much of a problem when the circuit has devices which operate only in one region as for convergence, the differential terms: the transconductances, are continuous. But when looking at large signal behavior of the circuit, the discontinuity prevents convergence. An example of the output waveform of a One-Stage Operational Transconductance Amplifier is shown in Figure 4.1 when using the Square Law model. The glitches can be observed when the NMOS and PMOS switch between their linear and saturation regions.

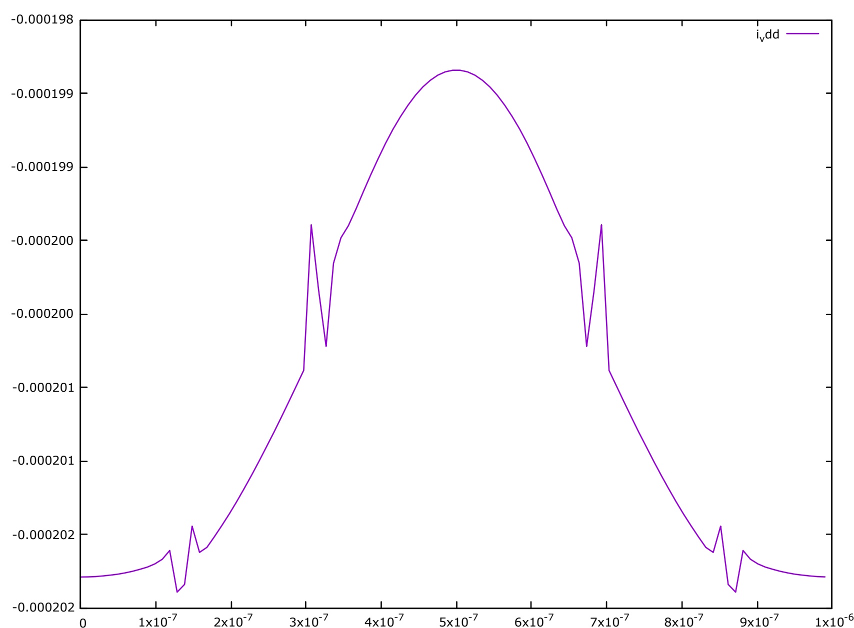

Figure 4.1: Output current waveform showing glitch due to discontinuity in the model

One way to avoid this would be to make the model more ideal and just ignore the channel-length modulation component of the equations. But this would mean that the drain current in saturation is independent of the drain voltage. This introduces issues of numerical convergence and cancellation when performing LU Decomposition with static pivoting which is discussed further in the chapter on Results and Discussions.

4.4 Implementation

In the Harmonic Balance method, in every iteration of the Newton-Raphson iterations, all the non-linear devices are to be evaluated at $2K + 1$ time instances where K is the number of harmonics being considered. The ngspice shared library requires a circuit to be provided which is parses and also the analysis type before the running the simulator. Because here we only require the operating points, we perform an `op` analysis. But there are multiple ways in which the circuit can be provided, given M non-linear devices:

1. A circuit with all non-linear devices and all their instances instantiated altogether. This would mean that the circuit has $M \times K$ non-linear devices.
2. A circuit with all K instances of one non-linear device.
3. A circuit with one instance of all M non-linear devices.

We choose the third form of circuit presentation to SPICE because this does not require us to alter the circuit structure and devices anytime throughout the iteration process (unline the second form) and also reduces the load on spice as it now needs to handle only a small circuit (unline the first form). This significantly reduces the matrix size as it will require smaller number of voltage sources and at the same time it does not increase the complexity of sending the `alter` commands to spice.

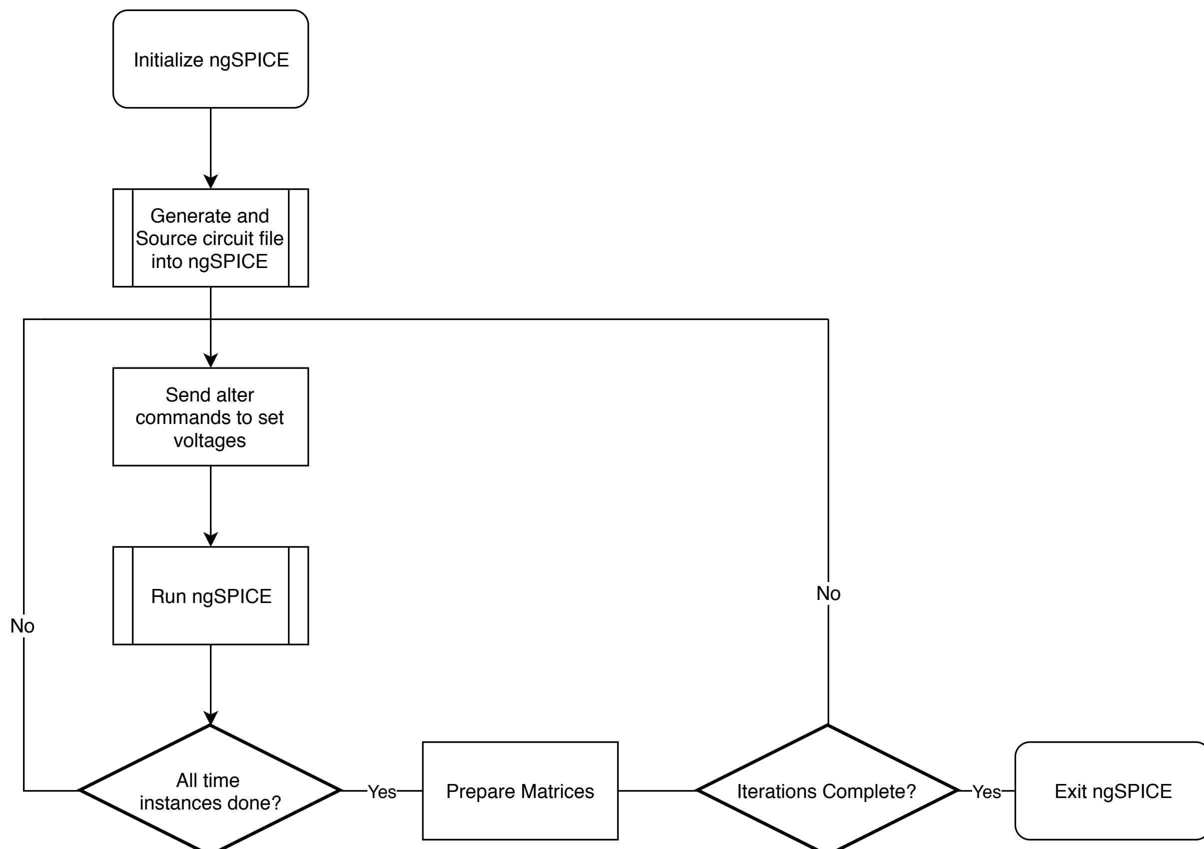

Figure 4.2: Flowchart showing usage of ngSPICE shared library

From the Flowchart in Figure 4.2 it can be clearly seen that using this scheme the circuit needs to be generated and sourced only once during the entire simulation cycle and hence reduces the time required if the circuit had be altered multiple times during each iteration. The method still suffers from the flaw of passing through ngSPICE's parser when an `alter` and data collection `compose` command is sent. Shown below is an example conversion from the supplied circuit file to the file to be parsed by ngSPICE.

```
1  Operational Transconductance Amplifier
2
3  frequency 1E6
4
5  Vinp inp 0 1.2 0.5E-3
6  Vinn inn 0 1.2 -0.5E-3
7
8  IBIAS vdd vbias 0.1E-5
9  MB vbias vbias 0 0 1E-6 180E-9 CMOSN
10 MSS source vbias 0 0 10E-6 180E-9 CMOSN
11 RBIAS vbias vdd 1E5
12
13 VDD vdd 0 1.8
14
15 ML1 outn outn vdd vdd 1E-6 180E-9 CMOSP
16 ML2 outp outn vdd vdd 1E-6 180E-9 CMOSP
17 *RL1 outp vdd 3E3
18 *RL2 outn vdd 3E3
19
20 M1 outp inn source 0 1E-6 180E-9 CMOSN
21 M2 outn inp source 0 1E-6 180E-9 CMOSN
22
23 plot outp outn
24 plot inp inn
25 plot i_vdd
```

```
1  operational transconductance amplifier
2  .include 65nm_bulk.pm
3
4  mb 6 6 0 0 cmosn W=1e-06 L=1.8e-07
5  V5 6 0 dc 0
6
7  mss 7 6 0 0 cmosn W=1e-05 L=1.8e-07
8  V6 7 0 dc 0
9
10 ml1 9 9 5 5 cmosp W=1e-06 L=1.8e-07
11 V8 9 0 dc 0
12 V4 5 0 dc 0
13
14 ml2 10 9 5 5 cmosp W=1e-06 L=1.8e-07
15 V9 10 0 dc 0
16
17 m1 10 3 7 0 cmosn W=1e-06 L=1.8e-07
18 V2 3 0 dc 0
19
20 m2 9 1 7 0 cmosn W=1e-06 L=1.8e-07
21 V0 1 0 dc 0
22
23 .op
24
25 .end
```

Chapter 5

Block-Aware Sparse LU Decomposition

5.1 Left-Looking LU Decomposition

The state-of-the art Sparse LU Decomposition still remains the one proposed by Gilbert and Peierls [5] called the left-Looking LU algorithm.

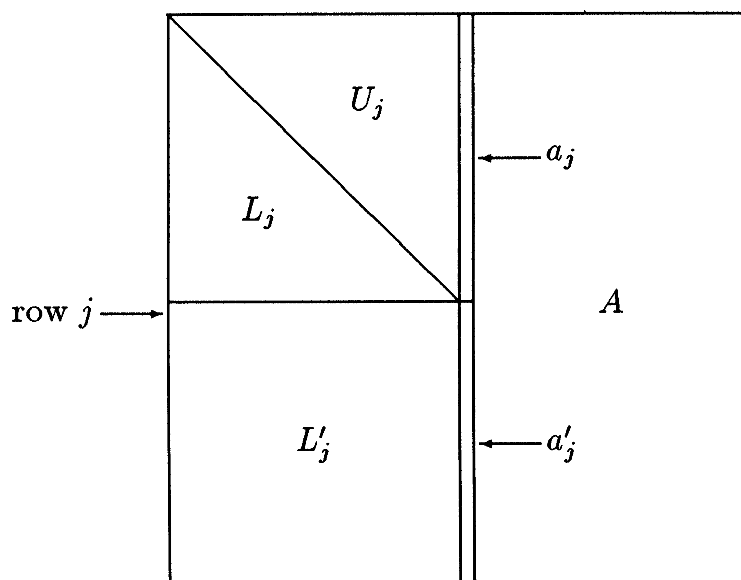

Figure 5.1: Block Matrix structure when computing Block Column j [5]

Every element in the A matrix which is to be factored into L and U matrices are considered to be block matrices themselves. From the Figure 5.1, L_j is the square L matrix of dimension $j-1$ and U_j is the corresponding U matrix. L'_j includes the jth row, but not the jth column which is yet to be factored.The algorithm is shown in Algorithm

1. Line 2 of Algorithm 1 requires a forward solve algorithm and the innovation of this algorithm lies in it. The forward solve algorithm is described in Algorithm 2.

Algorithm 1: Block LU Algorithm based on Gilbert-Peierls LU Algorithm

Input: $A \in \mathbb{R}^{(H\times N)\times(H\times N)}$

Output: $L, U \in \mathbb{R}^{(H\times N)\times(H\times N)}$

```
1 for j ← 1 to N do
      // Compute Block column j of L and U Matrix
2     Solve L_j u_j = a_j for u_j;
3     b'_j = a'_j − L'_j u_j;                     // Multiplication and Subtract
4     Pivot: Swap b_jj with the largest magnitude block of b'_j;
5     u_jj = b_jj;
6     l_jj = b'_j × inv(u_jj) ;                   // Inverse and Multiplication
7 end
```

For the purpose of Algorithm 2, it is necessary to obtain those elements in the new U Block Column that'll be non-zero and also must be sorted in some topological order. This order is necessary to ensure the dependencies across the loop iterations are not violated.

This can be achieved by creating a directed graph in the form of an adjacency list out of the L_j square matrix. The directed graph G contains vertices $V \in 1, \cdots, j-1$ and a vertex m is connected to n only if $L_j(n, m)$ is a non-zero block matrix. Because all the diagonal elements of L are identity matrices, every vertex is at least connected to itself; this information can be omitted from the graph and handled separately.

The non-zero elements of u_j are given by the vertices which are reachable from the non-zero elements of a_j block column. Ideally, the non-zero indices obtained must be sorted before going for the Forward Solve, but any topological order also works. Hence depth-first search and the reverse post-order are ideal graph traversal and topological order candidates respectively. Also, a list of visited nodes must be maintained so as to not traverse the same nodes multiple times.

Algorithm 2: Forward Solve

Input: $a_j \in \mathbb{R}^{(j\times H)\times H}$ and $L_j \in \mathbb{R}^{(j\times H)\times(j\times H)}$

Output: $u_j \in \mathbb{R}^{(j\times N)\times H}$

```
1 for k | u_kj ≠ 0 (in topological order) do
      // Revaluate u_j in a Sparse Loop
2     u_j = u_j − u_kj (l_1k, ··· , l_{j−1,k})^T ;   // Multiplication and Subtract
3 end
```

For the block matrix based LU, pivoting is not possible as choosing a largest block can be expensive if obtained using determinants. Hence pivoting is not performed.

5.2 Sparse Matrix Storage Schemes

Sparse matrices when stored as a normal dense matrix expends a large amount of memory at the cost of algorithmic and manipulation regularity which may still be ineffective when worked on as a dense matrix. Because of the very few number of elements, the sparse matrices are stored in different fashions which are optimized for the operation to be performed on them. Some of the common formats are discussed below:

Coordinate Format

The most basic format to store the sparse matrices is the coordinate format. The non-zero elements are stored in a array as tuples of `(row, column, value)` usually sorted based on rows or columns for fast access.

The format is advantageous as it allows for fast construction but it is not most optimal for random accesses and memory in most situations.

Compressed Sparse Storage

The storage format uses three arrays to store the matrix: `IA, JA, and A`. There are two versions of the storage format:

1. Row Format: `A` stores the Non-Zero element values row wise. `IA` stores the indices indicating the first entry in each row in the array `A`. And `JA` stores the column index of each non-zero element in `A`.

2. Column Format: `A` stores the Non-Zero element values column wise. `IA` stores the indices indicating the first entry in each column in the array `A`. And `JA` stores the row index of each non-zero element in `A`.

This format though more complex to construct but is a significant improvement over the coordinate format in terms of memory storage and traversing the matrix by random access for rows/columns depending on the format.

Block Compressed Sparse Storage

The only difference they share with the Compressed format is that the A matrix now consists of blocks instead of individual elements. This is a particularly useful format in a sparse matrix which consists of sparsely located dense blocks. As this is the case for

Harmonic Balance analysis and as LU decomposition requires traversal through columns making `Block Column Compressed Storage (BCCS)` appropriate for this application.

Tri-Diagonal Banded Matrix

For matrices which have non-zero elements in almost all the diagonal, super-diagonal and sub-diagonal entries only, the tri-diagonal banded storage format is suitable. It is a special case of a general banded storage format. Example of such a matrix is Figure 5.2.

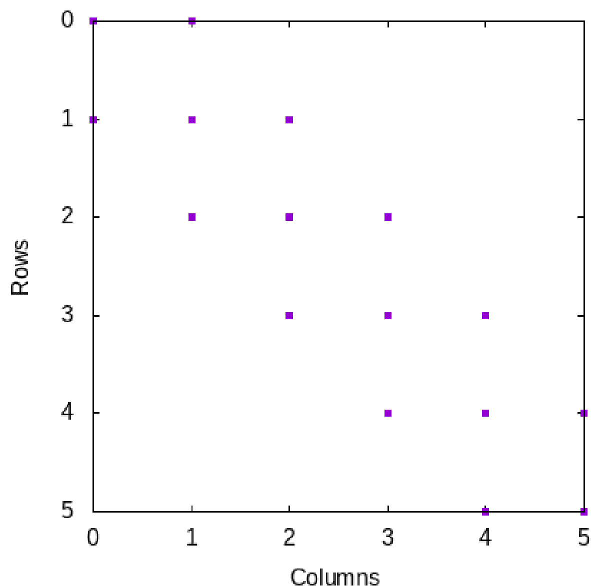

Figure 5.2: Tri-Diagonal Banded Matrix

The matrix is stored in a $3 \times N$ matrix with the diagonal occupying the second column and the sub and super diagonal occupying the 1st and 3rd columns. This storage format is used for the `Y` matrix as the blocks are only of linear circuit elements, they all conform to this pattern.

5.3 Symbolic LU Decomposition

The algorithm of left-looking LU decomposition described in the previous section is an on-line algorithm and obtains the structure of the output L and U factors as the algorithm proceeds along with the exact values. This is often suitable for matrices which are do be decomposed only once or a few times as pre-processing in such cases becomes ineffective and reduces performance.

But analysis of circuit simulation algorithms show that, though the values in the Jacobian matrix change at every iteration and with values of the circuit element, given the

fixed network topology across them, the sparsity-pattern[1] does not change. And as the sparsity-pattern of the L and U matrices only depend on that of the original matrix, they also remain constant. Hence, the sparsity pattern of the L and U matrices can be pre-computed and be used as a fixed constant in all iterations of the circuit simulator. This is also symbolic analysis which is LU decomposition without values. An example is shown in Figure 5.3.

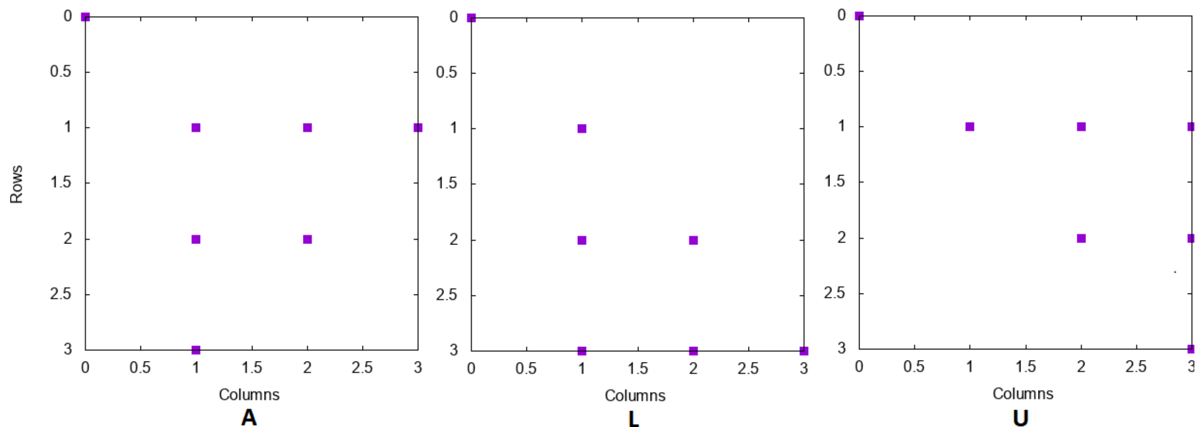

Figure 5.3: Example Symbolic Decomposition

Symbolic analysis can speed up the decomposition significantly by avoiding the need to perform graph traversals and sparse matrix structure updates during run-time. It also allows for the entire schedule of the algorithm to be prepared before-hand to allow for better tuning with respect to the hardware. This has been exploited by many to implement these algorithms on FPGAs[15]. But it suffers from a disadvantage of not being able to perform value dependent partial-pivoting which brings stability to the algorithm.

5.3.1 Fill - In

LU Decomposition for sparse matrices suffers from one major problem which is even aggravated for circuit matrices: fill-in. The L and U matrices put together have a lot more non-zero elements that the original matrix as shown in an example in Figure 5.4. This increases memory required to store them and also the number of operations to be performed to evaluate the decomposition and to use the L and U matrix to solve a system of linear equations.

The fill-in can be reduced by re-ordering the row and columns of the original matrix.

$$PAQ = LU \tag{5.1}$$

[1]the rows and columns in the matrix which contain the non-zero elements

As this is a purely symbolic problem, many heuristics have been proposed to find a suitable ordering based on graph algorithms such as AMD [16] and COLAMD [17]. In this work, COLAMD has been used for its effectiveness on circuit matrices and as it is simple and only provides column permutations.

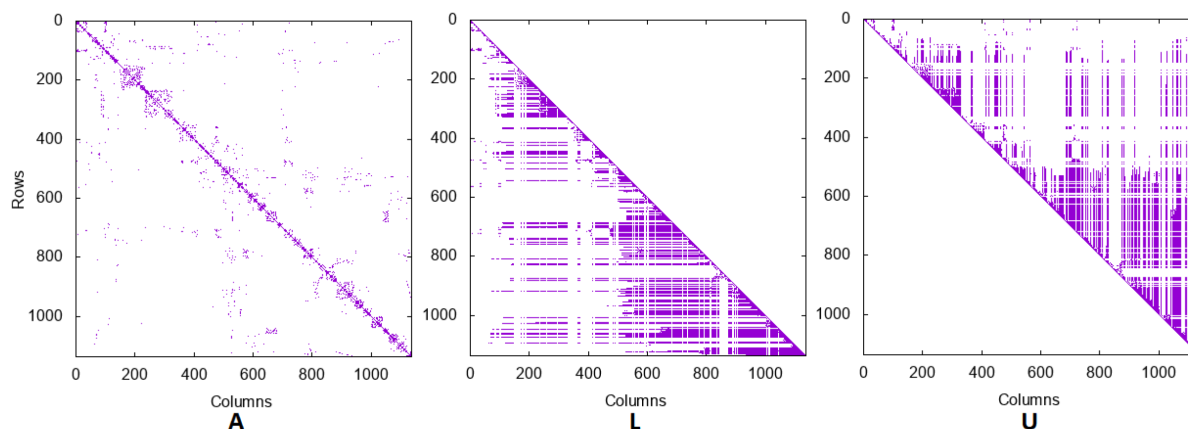

Figure 5.4: Example Circuit matrix showing extensive Fill-In after LU Decomposition

5.3.2 Row Pivoting

As it was mentioned earlier, during symbolic analysis it is not possible to perform partial pivoting based on the values in the matrix. But the COLAMD algorithm permutes the column in such a fashion that zero entries are brought into the diagonals which prevents LU decomposition without pivoting to succeed.

To circumvent this issue pivoting becomes necessary. The pivoting used for this purpose is static and is only based on whether an element is non-zero or not. The row permutation is performed by choosing the first non-zero element found in the column and permuting it with the diagonal element. An example of this application is shown in Figure 5.5.

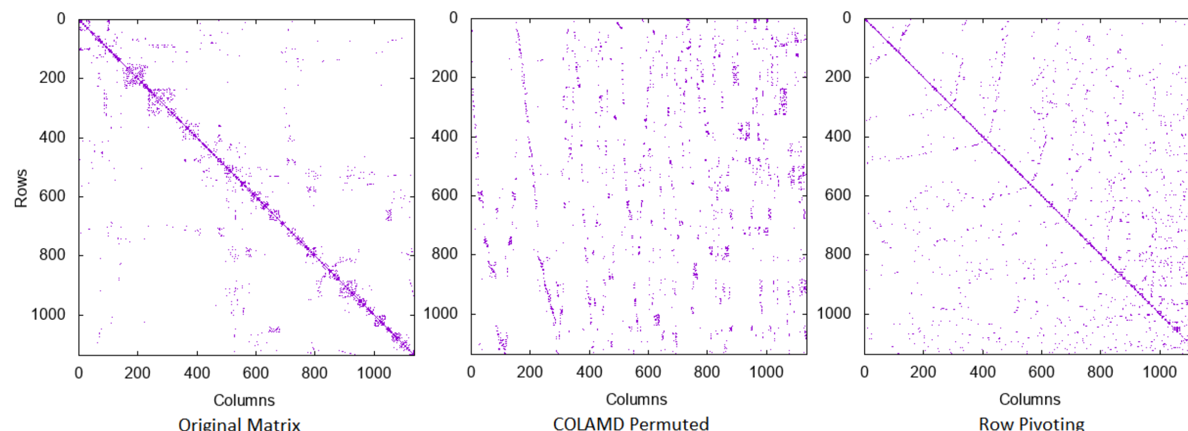

Figure 5.5: Matrices showing effect of COLAMD and Pivoting on diagonal elements

5.3.3 Column Dependency Analysis

To improve performance of the algorithm, it is necessary to analyze the extent to which the LU decomposition can be parallelized. During the forward solve of the Left-Looking LU Decomposition the solution on one column depends usually on one or more of the previous computed columns which is used for the forward solve. This dependency can be obtained by analyzing the sparsity structure of the U matrix after symbolic LU decomposition. Column i is dependent on Column j iff U(j, i) is non-zero.

The dependencies are depicted as a Strict Directed Graph where the nodes represent the columns while the edges represent the dependencies. That is to say an edge from i to j means that the column j is dependent on i. An example matrix and its U factor is shown in Figure 5.6 and its dependency graph is shown in Figure 5.7.

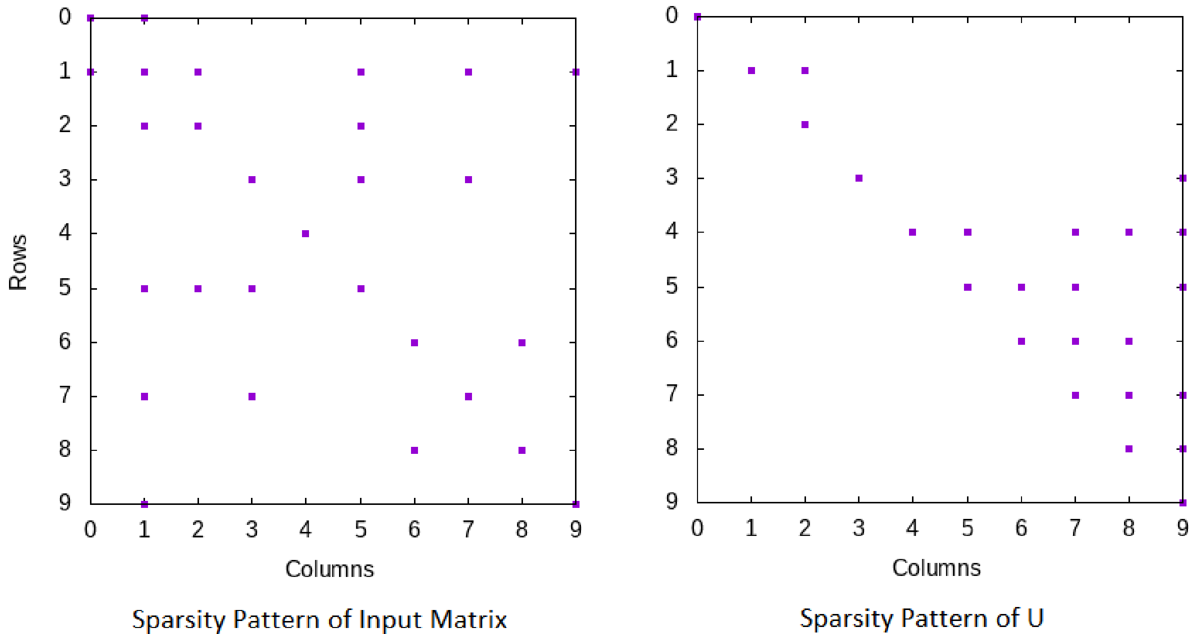

Figure 5.6: Example Matrix and its U factor

The column dependency graph is stored in the program as an adjacency list. The tool graphviz [18] is used to plot the graph in the `dot` format. The number of columns each column is dependent on is computed using a breadth first search of the graph starting from those nodes which are not dependent on any others. A queue is maintaining during runtime which stores initially the root nodes. When a thread demands for a column, the top element from the queue is popped and when it returns, the child nodes dependency count is update. As soon as a node reached zero dependency count, it is deemed ready and is pushed into the queue.

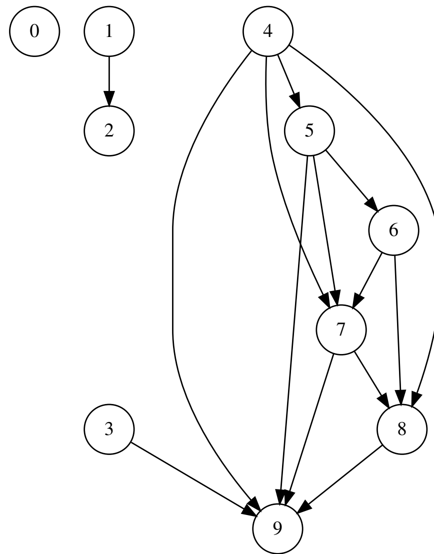

Figure 5.7: Column Dependency Graph for Matrix in Figure 5.6

When the queue is empty, the thread waits for a column to become ready for computation. If before this happens all the columns are accounted for the thread exits. This ensures that if during computation more than one node becomes ready, threads are available otherwise they would have to be terminated prematurely. The blocking is implemented using semaphores. It could also be implemented using a polling loop which is inefficient, but more importantly not possible using STL queues in C++ because they are not thread-safe and cannot be declared volatile. This means that a while loop checking the queue empty status will be optimized away at a higher optimization level as there is no queue operation being performed within the loop.

Another possible approach is to levelize the graph which is shown in Table 5.1. Each level of the graph then can be scheduled in parallel. This is more appropriate in case a static schedule needs to be generated.

Table 5.1: Levelized List of nodes based on the Example Matrix

Level	Columns
1	0, 1, 3, 4
2	2, 5
3	6
4	7
5	8
6	9

5.4 Dense LU Decomposition

When the number of harmonics become too large, the block correspondingly become very much large in size. A tiled dense LU algorithm was implemented by further dividing the blocks into smaller dense blocks to harness the locality of reference. For the operations on the tiles, Intel MKL is itself used, which implies that if any performance benefit is observed then the algorithm works faster than Intel MKL.

Algorithm 3: In-place tiled dense LU Decomposition [19]

Input: $A \in \mathbb{R}^{(H\times N)\times(H\times N)}$

Output: $L, U \in \mathbb{R}^{(H\times N)\times(H\times N)}$

```
1  for i ← 1 to N do
       // Compute Block column and row i of L and U Matrix
2      L_ii, U_ii =LU Decomposition(A_ii);                  // Matrix LU
       // Update perimeter blocks
3      for j ← (i + 1) to N do
4          Solve L_ij U_ii = A_ij for L_ij;                 // Backward Solve
5          Solve L_ii U_ji = A_ji for U_ji;                 // Forward Solve
6      end
       // Update interior blocks
7      for j ← (i + 1) to N do
8          for k ← (i + 1) to N do
9              A_kj = A_kj − L_ki × U_ij;       // Multiplication and Subtract
10         end
11     end
12 end
```

Chapter 6

Automatic Circuit Parser

For the purpose of completing the circuit simulator and automate the process of the Modified Nodal Analysis matrix generation a basic circuit parser was also implemented. It takes as input a circuit netlist with syntax similar to that used in SPICE to create the MNA matrices and structures to allow for evaluation and implementation of the intermediate steps of the simulator.

6.1 Matrix Construction

Once the entire circuit has been read into a structure which stores all the node information and mapping of the node and device names to internal structures, the Y and J matrix which form a part of the simulator iterations are constructed in a symbolic.

To keep the construction as structured as possible each column corresponds to a particular node voltage and the corresponding row represents the Kirchoff's Current Law equation written for that specific node. For situations when the column represents a current as in case of voltage sources or inductors, the corresponding voltage-related expression is written in the corresponding row. This often leads to zeros on the diagonal because of the construction but it is handled by the LU Decomposition algorithm by use of static pivoting.

Each device structure also stores the pointers to their corresponding blocks as they are responsible for filling in values in the blocks when the appropriate function is called upon. This is necessary because once the column and row permutations are applied permanently to the Jacobian matrix the blocks are moved and hence the original structured mapping no longer exists. Hence the onus of retaining which block corresponds to which device and which port is on the device structure.

6.2 Syntax

As mentioned earlier, the syntax very closely follows that of SPICE but in its current form it is quite restrictive. A brief description of the syntax is given below:

- The first *non-empty* line of the circuit is taken to be the name of the circuit and ignored for simulation purposes.
- Any line beginning with a `'*'` is considered to be comment. Inline comments are not supported.
- All lines are converted to lowercase before parsing and hence uppercase and lowercase characters are mapped together internally. This is important to note, to avoid clashing names and unwanted shorts between nodes.
- In the current state Resistors, Capcitors, Inductors, Voltage and Current sources and MOSFETs are supported by the program. The device line goes as follows: <type_character><device_name> <ports_list> <parameter_list> [model]

Table 6.1: Device description and synatx for Circuit Parser

Device Type	Type Character	Ports	Parameters	Models
Resistor	r	2 Ports	Resistance (Ω)	–
Capacitor	c	2 Ports	Capacitance (F)	–
Inductor	l	2 Ports	Inductance (H)	–
Voltage Source	v	+ and - Port	Voltage (V) Fourier Series	–
Current Source	i	Current Direction: Port 1 to 2	Current (A) Fourier Series	–
MOSFET	m	Drain, Gate, Source and Body	W and L	CMOSN/ CMOSP

- Plot Commands: `plot` command is also available and any node voltage can be plotted by following `plot` with the node names. Multiple nodes will be plotted in the same window when using a single `plot` line for all of them. `GNUPLOT` is used for the purpose of plotting, while the raw data is stored in a binary format always.
- Frequency: It is compulsory to mention the fundamental frequency. The keyword `freq[uency]` followed by the frequency value can be used to set the simulation frequency.

6.3 Example

Below is the code of an Operational Transconductance amplifier written the syntax described in the previous chapter. Figure 6.1 shows the non-zero pattern of the Jacobian matrix formed using the Circuit Parser.

```
OTA with Parasitics

frequency 1E6
Vinp inp 0 1.2 5E-3
Vinn inn 0 1.2 -5E-3
VDD vdd 0 1.8

IBIAS vdd vbias 1E-6
MB bdrain bgate bsource 0 1E-6 180E-9 CMOSN
MSS ssdrain ssgate sssource 0 10E-6 180E-9 CMOSN
RBIAS vbias vdd 1E5

ML1 l1drain l1drain l1source vdd 1E-6 180E-9 CMOSP
ML2 l2drain l1drain l2source vdd 1E-6 180E-9 CMOSP

M1 outn gate1 source1 0 1E-6 180E-9 CMOSN
M2 outp gate2 source2 0 1E-6 180E-9 CMOSN

*Parasitics
Rparinp inp gate1 2
Rparinn inn gate2 2
Rparsrc1 source1 ssdrain 2
Rparsrc2 source2 ssdrain 2

Rparbias vbias bdrain 2
Rpardiode bdrain bgate 2
Rparvbias bgate ssgate 2

Rparl1 l1drain outn 2
Rparl2 l2drain outp 2

Rparvdd1 l1source vdd 2
Rparvdd2 l2source vdd 2
Rparssgnd sssource 0 2
Rparbgnd bsource 0 2

plot outp outn
plot inp inn
plot i_vdd
```

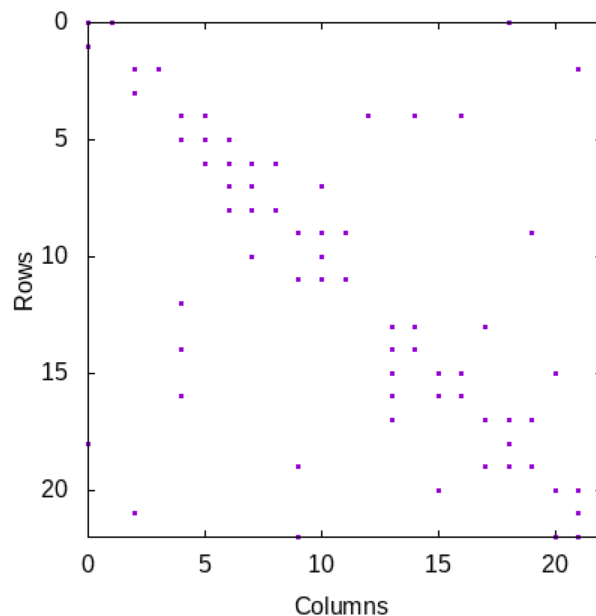

Figure 6.1: J Matrix of the OTA with parasitics generated from the Circuit Parser

Chapter 7

Custom Hardware Implementation

The secondary purpose to this work was to investigate the possibility of implementation of the Block Sparse LU decomposition on an FPGA to accelerate the dense computations and make use of the parallelism in the algorithm by implementing multiple computational blocks.

From the algorithm described in the previous Chapter 5 and the Harmonic Balance algorithm, there are six main dense matrix operations to be performed. The list below enumerates the operations, where they are used in the Harmonic Balance flow and their corresponding BLAS/LAPACK function.

1. **Matrix-Matrix Multiplication and Add** - `cblas_sgemm`: The most common operation used in the column update operation of LU decomposition.
2. **Matrix-Vector Multiplication and Add** - `cblas_sgemv`: Used in Forward and Backward solve algorithms
3. **Band Matrix-Vector Multiplication and Add** - `cblas_sgbmv`: Used to multiply the Y matrix with the x vector in Harmonic Balance
4. **Dense Matrix Factorization** - `LAPACKE_sgetrf`: Used once per column in LU decomposition and Backward Solve as a precursor to invert the pivot block.
5. **Dense Matrix Inverse** - `LAPACKE_sgetri`: Used once per column in LU decomposition to invert the pivot block.
6. **Dense Matrix Linear System Solve** - `LAPACKE_sgetrs`: Used once per column in Backward Solve to solve using the pivot block.

These algorithms are where the parallelism of the FPGA can be utilized and hence using Vivado HLS, these functions are optimized for hardware using the various available

pragma. The following section contains a discussion on some of the pragmas used and their utilities.

7.1 HLS Hardware Optimization

7.1.1 Function Instantiate

Consider the matrix multiply and add algorithm. The operation performed by this algorithm can be summarized as:

$$C = A \times B + \beta C$$

where β is a scaling factor. This algorithm is very diverse and works for all values of the floating point variable β, but it is clear that for special values of the variable the hardware can be differently optimized. Consider the two cases below:

- Case $\beta = 0$: In this case, there is no need for the addition operation to be performed and the MAC operations can be immediately be replaced by simple and faster multiplication operations. Another optimization possible is that the C matrix need not be loaded into the memory but only needs to be written back to.
- Case $\beta = \pm 1$: In this case, though MAC operations would be required and the C Matrix needs to be read from the memory, the multiplication β need not be performed. This might not yield significant performance benefits, but does help conserve resources.

The pragma function instantiate helps in taking advantage of such situations by optimizing the function specifically for different calls to the function with fixed values of the specified variable which in this case would be β. The only disadvantage in terms of the implementation is the multiple copies of the same computation block created for even multiple serial calls.

This optimization can be applied to the Matrix-Vector Multiplication and Add operations and the Band Matrix-Vector Multiplication and Add operations.

7.1.2 Array Partitioning

Array partitioning alters how arrays are stored in the local memory/BRAMs instantiated on the FPGAs. They affect how and in what fashion the array data can be accessed. There are three ways in which this can be achieved, which is shown in Figure 7.1. The advantage of array partitioning is that storing the matrix elements in different BRAMs allows for parallel access of the data.

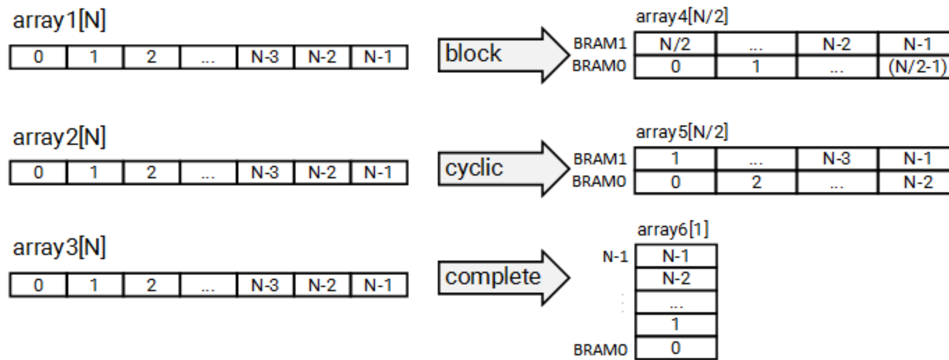

Figure 7.1: Types of Array Partitioning

Consider the situation of Matrix-Matrix Multiplication. For every element a row from matrix A and a column from matrix B are multiplied with each other, which is depicted in Figure 7.2. This operation can be parallelized with the rows of A and the columns of B are split into multiple memories to allow for parallel multiply and addition of multiple sections of the matrices.

This is specified to the compiler as follows:

```
#pragma HLS array_partition variable=A block factor=m dim=2
#pragma HLS array_partition variable=B block factor=m dim=1
```

where the matrix A is split about its second dimension which is across columns while B is split about it's first dimension.

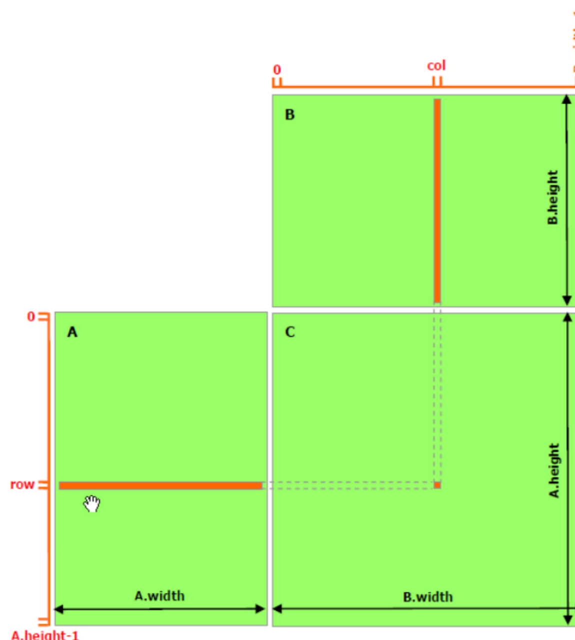

Figure 7.2: Matrix Multiplication showing computation of a single element of Matrix C

7.1.3 Pipelining

This is most commonly used optimization which provides the maximum performance benefit. Pipelining works by hiding the latecy of long duration operations by reusing empty resources by splitting the operation into multiple stages. This allows for operations to be scheduled every cycle and one operation can then be scheduled and retired in every cycle. The process is very well explained in the Figure 7.3.

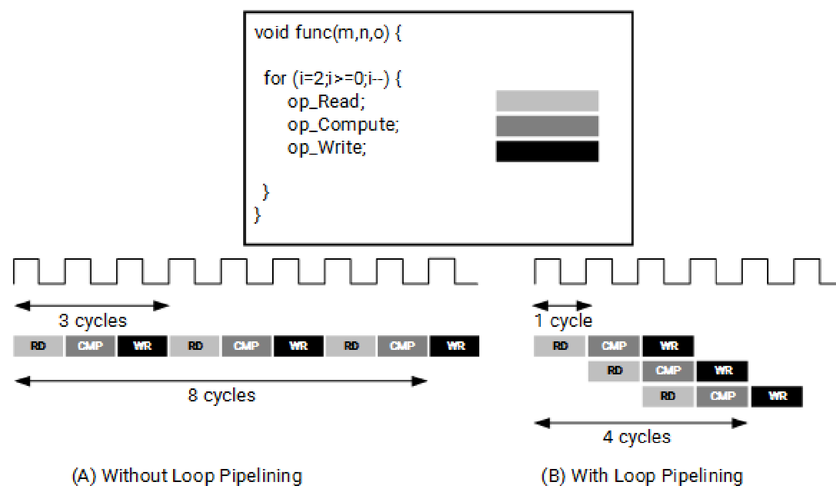

Figure 7.3: Example program showing effects and benefits of pipelining the program [6]

Addition of the pipeline pragma within a loop which contains a nested loop forces the inner loop to be unrolled and hence pipelined in the process which is the main advantage when used in program such as Matrix Multiplication. This allows for the RAM read, RAM write and the MAX operation to be pipelined and hence obtain performance benefit. It can also be used when streaming data from and to a shared processor through a bus into the local memory, speeding up the data communication latency.

Occurrence Directive

It often occurs that there are conditional blocks within loops which occur a fraction of the times the loop is executed. For example a loop with tripcount of $N \times N$ might have a conditional block execute only N times. This allows for the conditional block to be pipelined infrequently and even in a parallel fashion with the main pipeline, a few iterations earlier allowing the data to be readily accessible when required later.

This is useful when reading a matrix and a vector which have the same dimensions as the matrix has $N \times N$ elements while the vector has only N.

7.2 Data Movement Optimizations

7.2.1 Data Zero Copy

Data zero copy avoid explicitly moving the data from the processor memory to the local device memory but allows for the hardware to directly access the processor's shared memory through a bus interface. This allows for memory access to be parallelized with computation being performed in the hardware, but random access of data is not suggested. Parts of the data can be accessed in short burst reads and writes and computation performed on them while more data is read/written.

7.2.2 Data Access Pattern - Sequential

For arrays which are to be accessed in a sequential pattern, instead of creating a Random access interface, a FIFO can be created which allows for data to stream in gradually from the main memory as required. This also allows for the computation and memory access to be overlapped and hence hiding the data movement latency.

This is especially useful in reading the matrices and vectors for Matrix multiplication operations and the right-hand side matrices are read only once and in a streaming regular fashion.

Chapter 8

Results and Discussions

8.1 Fill - In Reduction

The results of reduction in Fill-In when COLAMD is applied on circuit matrices which are obtained from the Florida Suite-Sparse Matrix Collection [20] are shown in Table 8.1

Table 8.1: Fill-In w/ and w/o COLAMD column permutations for Sparse Matrices

Matrix	Dimension	NNZ	Inital Fill-In	COLAMD Fill-In
494_bus	494	1666	11202	1700
662_bus	662	2474	14524	5373
add_20	2395	17319	3996552	59596
add_32	4960	23884	5301064	17598
Hamrle1	32	98	99	84
rajat11	135	812	3541	742

8.2 Numerical Instability

Because we are performing only static pivoting, there are many situations when the determinant of the diagonal block comes close to 0 which causes numerical instability producing very high intermediate values. The problem is further aggravated as the number of harmonics are further increased. Some of the problems, their causes and possible solutions are discussed below:

- **Perfect Numerical Cancellation** This often occurs in the first iteration when starting with an initial guess of all zero. This can be circumvented by detecting the singular nature of the matrix and adding a small value to the diagonal elements. This can be seen as equivalent to adding a large resistor to ground from that diagonal node.

- **Ideal MOSFETs**: Ideal MOSFETs do not have any drain voltage dependence in the Saturation region which causes Perfect Numerical Cancellation at every iteration. This cannot be circumvented using the previous method but requires pivoting to be necessarily performed.

- **Ideal Current Sources**: Ideal current sources though do not cause singular diagonal matrices, they cause close to perfect cancellation which produces a very small diagonal block which when inverted produces incorrect results due to loss of numerical precision. It can be attributed to the absence of a relation between the two node voltages of the current source and hence can be corrected by adding a large resistor in parallel to it which provides such an equation.

- **SPICE related instability**: When using a simple square law, large intermediate values not going beyond the numerical precision are still tolerable and get corrected iteratively. But when SPICE calls are made, large values prevent SPICE from converging onto a stable operating point and hence preventing simulation.

All of the issues can be easily solved by partial pivoting. But as that cannot be performed online, a possible solution is to preprocess the circuit by performing DC simulation with partial pivoting. This pivoting can then be used as static guide for the Harmonic Balance iterations. This will also reduce the number of iterations in HB as the DC solution has been precomputed.

8.3 Simulation Setup

8.3.1 Intel Xeon Phi: Knights Landing Edition

Intel Xeon Phi is a series of manycore processors using the x86 ISA. It was originally based on a GPU design and hence shares many applications with GPU. The main advantage of this platform over a GPU is its capability to run even single-threaded applications or those targeted for other x86 architectures without much alteration. Hence this processor can run both sections of the code: the serial and the parallelizable portion, hence minimising the switching costs [7].

Knights Landing is the code name for the second generation many integrated core (MIC) architecture. The processor contains 64 Atom Cores with upto 4 threads per core using Intel Hyperthreading technology. Every core in the setup has 512-Bit vector instruction support through Intels AVX512 SIMD instruction set. The processor has a base clock speed of 1300 MHz and a L2 cache of 32 MB. It also includes a near memory called the MCDRAM of 16GB which acts as an extended cache. The maximum performance

achievable is around 2.7 TFLOPs [21].

The previous version of the Xeon Phi code-named Knights Corner, was only utilizable as a co-processor and sported a ring-interconnect network on chip. But this suffers from scaling issues especially in terms of bandwidth and latency. This version implements a mesh networking topology which significantly improves the latency and communication, making it a direct competitor to commercial GPUs.

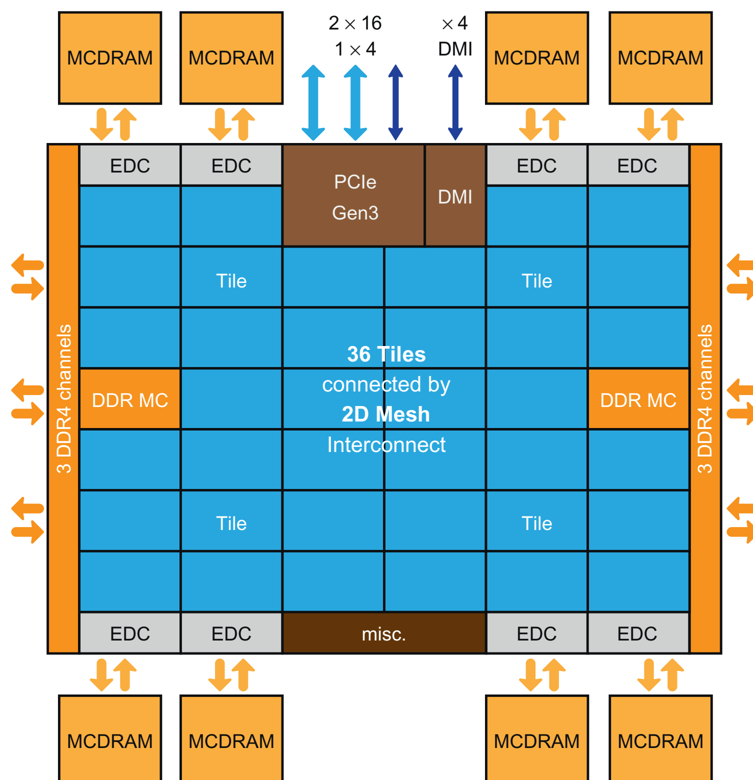

Figure 8.1: Intel Xeon Phi: Knights Landing Architecture [7]

AVX - 512

AVX 512 are Advanced Vector Extensions supporting single instruction operation of upto 8 double-precision floating point operations. These engines are similar to the GPU's ability to process SIMD instructions. Utilization of these instructions is of utmost importance to fully exploit the parallelism capability of the Xeon Phi Processors.

Intel MKL

Though the best performance benefit can be obtained by using the AVX intrinsic instructions directly in the program, that would require programming at a very low-level. The Intel MKL library conveniently offers a higher level of abstraction by implementing

the BLAS [22] and LAPACK [23] functions optimized for the MIC architecture of KNL. Some of the important functions of the library are mentioned below:

1. `cblas_?gemm`: Matrix Multiplication BLAS function which performs the following operation `C := alpha*(A)*(B) + beta*C` where `alpha` and `beta` are scalars
2. `LAPACKE_?getrf`: Matrix LU Decomposition which performs the following operation `A = P*L*U,` where P is the permutation matrix
3. `LAPACKE_?getri`: Matrix Inverse which performs the following operation `inv(A)` given the LU decomposition of the matrix

The library is multi-threaded using OpenMP from within and hence further parallellisation is not necessary when working with blocks of large enough sizes and these kernels take significant amount time and maximum parallelism is favourable.

8.3.2 Xilinx ZedBoard: Zynq-7000 SoC

For testing the implementation on the FPGA, Xilinx Zedboard was used which contains the Zynq-7000 SoC. The system-on-chip contains a dedicated ARM processor and user configurable programmable logic. The processor is a dual-core ARM Cortex A9 processor integrated with the programmable logic implemented in 28nm technology.

Table 8.2: Xilinx ZedBoard Frequency Specifications

	Frequency
Processor System	666.67 MHz
Data Motion Network	142.86 MHz
FPGA	142.86 MHz

8.4 Dense LU Decomposition

The dense LU implementation was compared for different matrix sizes and block sizes with the implementation of the MKL library.

From the Figure 8.2, it can clearly be observed that only for really large matrices like 8192×8192, the dense tiled LU algorithm provides significant benefit mainly because of the division of the blocks into sub-blocks which increase the locality of memory and hence providing the speedup.

Hence it can be concluded that for number harmonics of the order 2000 and below, using MKL functions directly on the block matrix is faster than going for further subdivision. This is because MKL utilises AVX instrinsics and hence is better tuned for Intel

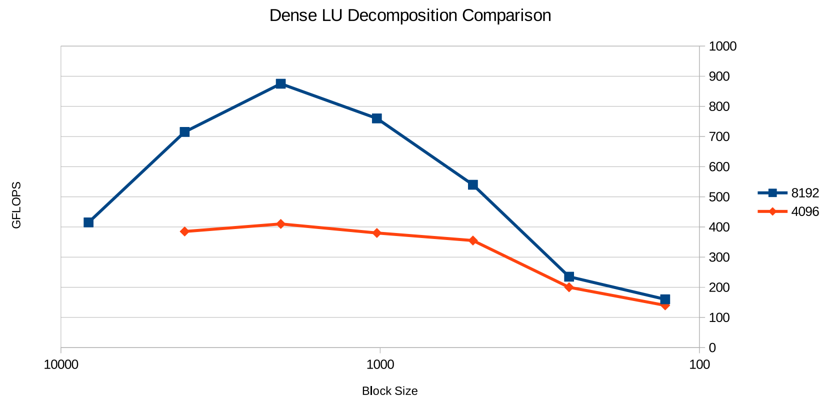

Figure 8.2: Dense Tiled LU Decomposition Comparison

hardware. For all future results the custom dense tiled LU decomposition is not used as such high number of harmonics are not required for simple circuit analyses.

8.5 FPGA Implementation

8.5.1 Performance Comparison

In this sub-section the performance of the functions implemented on hardware are compared with similar implementations on the ARM processor on the Zynq7000 SoC.

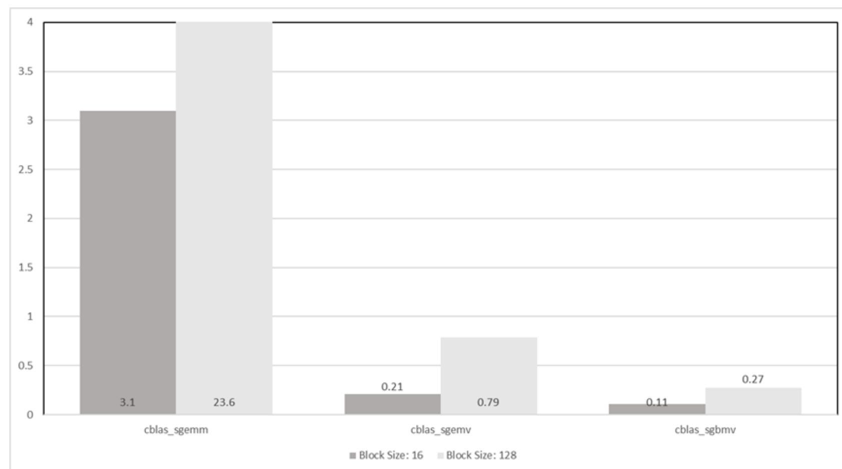

Figure 8.3: Speedup of the BLAS algorithms on the FPGA over CPU

From Figure 8.3, we can observe that the maximum speedup greater than 1 is observed only in the case of matrix multiplication. This is because the maximum parallelism is available in matrix multiplication and also the larger number of floating point operations are able to better mask the communication latency. Continuing this argument it is clear why the banded matrix-vector multiplication has the least performance: it has the least

number of operations that be performed in parallel as, in each row, only 3 MAC operations need to be performed.

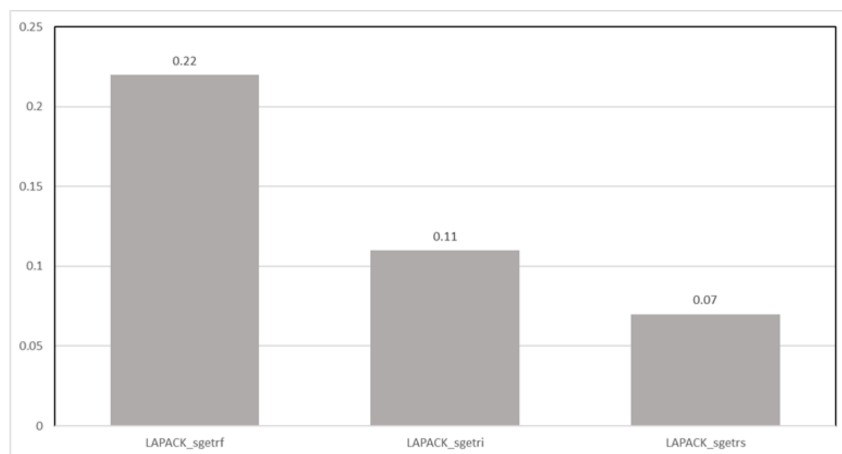

Figure 8.4: Speedup of the LAPACK algorithms on the FPGA with block size of 64

From Figure 8.4, we can observe that the performance degradation is severe in case of the matrix inverse, factorization and linear solve operations. This is because the loops in the algorithms are not of fixed tripcount and they change with every outer loop iteration. This makes pipelining inefficient when minimizing the utilization of resources as the number of operations keep on changing.

Matrix inverse is faster than linear solve because matrix inverse can be seen as a special case of linear solve where the other matrix is the identity matrix which reduces data movement as well as computation. Factorization is faster than both the other operations because though internally it does perform forward solve internally, it also has a matrix-vector product stage which is regular and parallelizable.

8.5.2 Timing Closure

Issues with timing closure were observed when implementing pivoting in Matrix Factorization function. The original code which caused the timing closure issue and the altered code to avoid it are shown below:

```
58 //This Code Segment causes Timing Closure Issues
59 for (int j = i; j < H; j++){
60 #pragma HLS PIPELINE
61     absA = ((_A[i][j] >= 0) ? _A[i][j] : -_A[i][j]);
62     if (absA > maxA) {
63         maxA = absA;
64         jmax = j;
65     }
66 }
```

```
68 // Modified Pivot Detection to meet Timing Constraints
69 for (int j = i; j < H; j++){
70 #pragma HLS PIPELINE
71     absA = ((_A[i][j] >= 0) ? _A[i][j] : -_A[i][j]);
72     if ((_A[i][j] > maxA) || (_A[i][j] < -maxA)) {
73         maxA = absA;
74         jmax = j;
75     }
76 }
```

In the first code segment, the if condition evaluation is dependent on the completion of the previous statement which is also a conditional evaluation followed by a negation in the worst case. This cause a series of sequential operations which causes a timing closure problem because of the requirement of pipelining. This problem is corrected by making the evaluation of the variable `absA` independent of the if statement evaluation. Now there are two independent conditions which can clearly be executed in parallel and hence avoiding the timing closure problem.

8.6 Simulation Results

8.6.1 Example: Operational Transconductance Amplifier

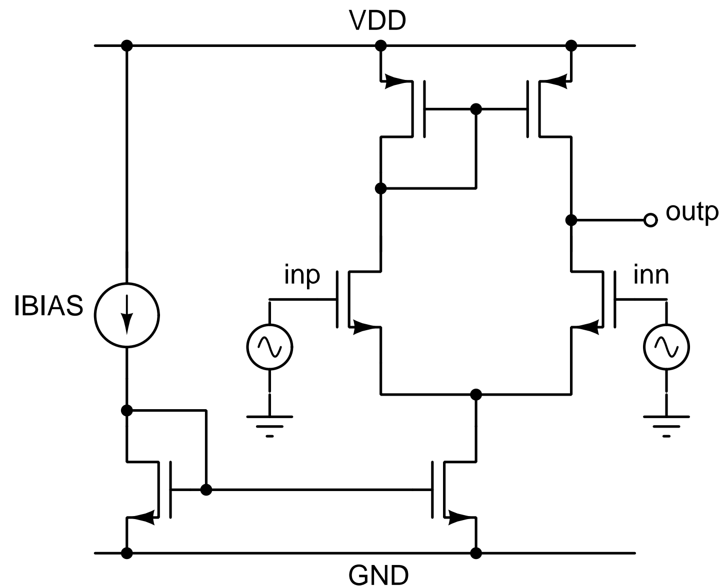

Figure 8.5: Operational Transconductance Amplifier Schematic

The Figure 8.6 shows the output waveform for multiple input voltage amplitudes using 20 harmonics of the circuit whose schematic is shown in Figure 8.5. We can see clearly that even for a large input, it is able to predict the large signal behaviour at the output.

8.6.2 Output

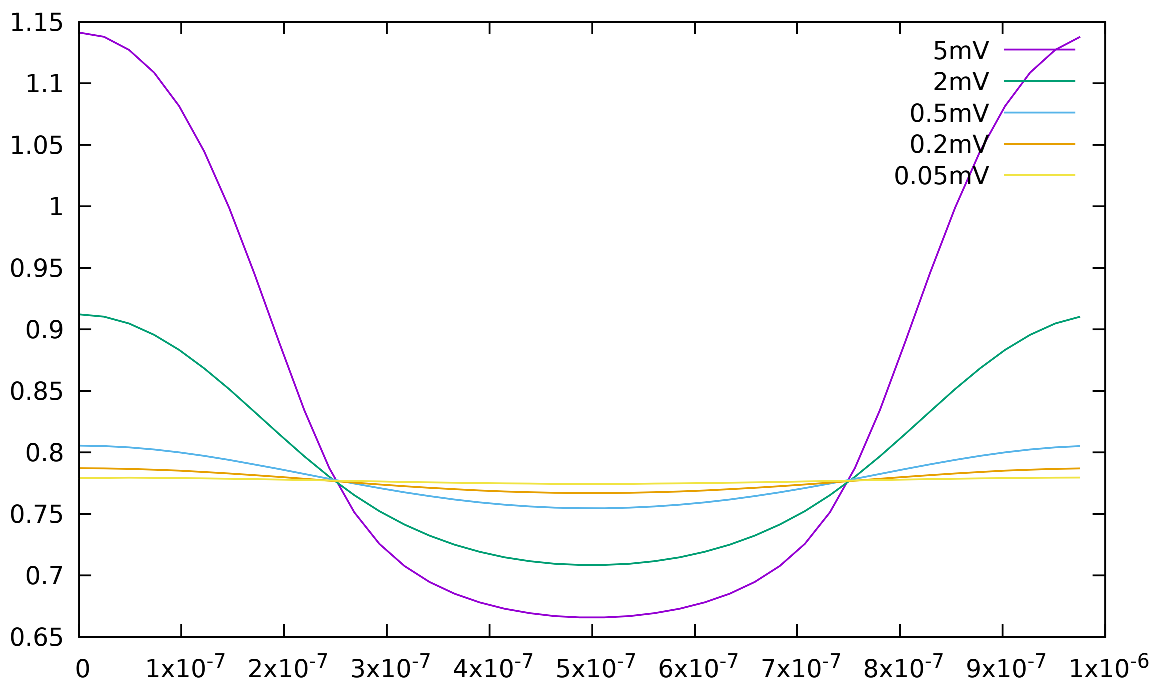

Figure 8.6: Output waveforms of the OTA for different input voltage amplitudes

8.6.3 Performance Analysis

Note: Unless otherwise mentioned, all results in the graphs are based on the program which uses the simple square law model so that the LU Decomposition's performance can be clearly seen in the overall simulation time.

Table 8.3: Matrix sizes and No. of Non-Linear Elements in circuits used to analyze performance

Circuit Name	Matrix Dimension	Non-Linear Elements
`ota_par`	23	6
`ota`	10	6
`diffamp`	9	2

The three circuits as shown in the Table 8.3 are chosen such that the variation with the number of non-linear elements as well as the size of the matrix dimension can be clearly observed. Circuits `ota_par` and `ota` have the same number of non-linear elements and hence their model-evaluation phase must take exactly the same amount of time, while the circuits `ota` and `diffamp` have similar matrix dimensions but differ in the number of non-linear elements.

Number of Threads executing Columns in Parallel

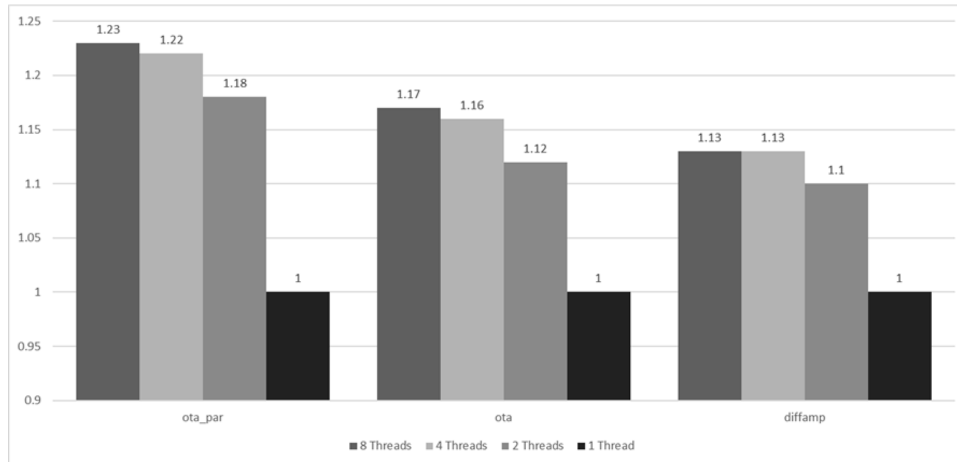

Figure 8.7: Speedup of extracting Column-Level Parallelism normalized to serial case

From the graph in Figure 8.7, we can clearly see that there is a significant speedup when parallelism is extracted at a column level. Comparing the size of the matrices with the amount of speedup achieved, it can also be noticed that for larger matrices more parallelism can be extracted.

Number of Harmonics

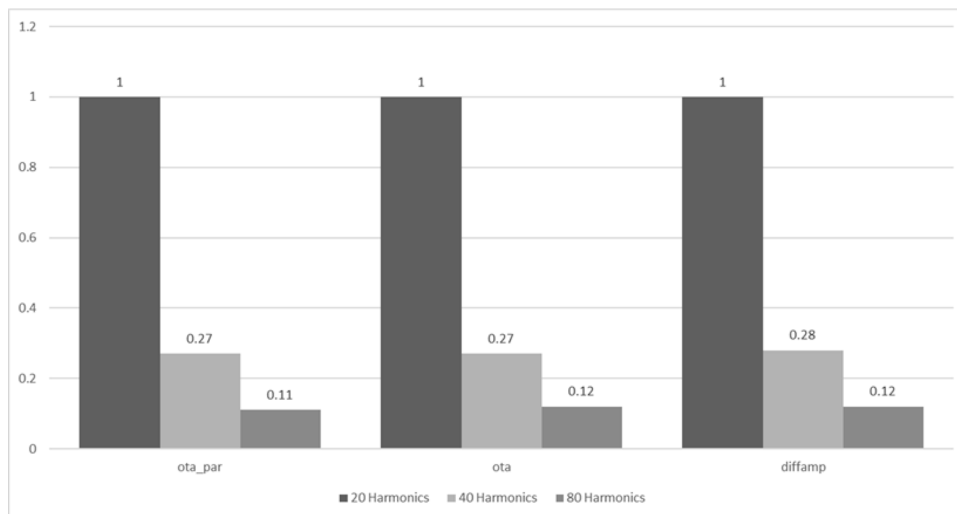

Figure 8.8: Performance degradation with increasing harmonics normalized to 20 Harmonics

From this data in Figure 8.8 we can clearly see that irrespective of the circuit, the degradation with increasing number of harmonics is the same. This shows that the performance in limited by the block operations and model evaluation which both directly depend on the size of the blocks which is given by the number of harmonics.

ngSPICE based Model Evaluation

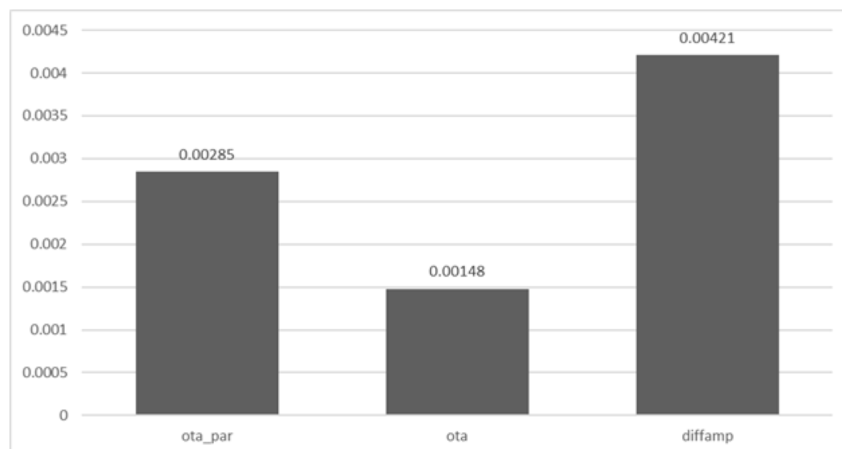

Figure 8.9: Performance degradation with using ngSPICE calls for Model Evaluation

From the data shown in Figure 8.9, we can see that there is a significant performance degradation of more than 250 times even when using only 2 non-linear devices. The degradation is severe in the `ota` circuit because compared to the dimension of the matrix, the number of non-linear elements are large. In case of `ota_par`, the larger part of the simulation time is also spent in the LU decomposition and hence the degradation seen is smaller. Because SPICE is serial, especially on the KNL the degradation seen is exaggerated. On smaller processors which have less parallelization capability while running at higher frequency, this gap is bridged to a certain extent.

Profiling Results

Function	%
main	39.1%
circuit::populate_J	34.2%
mosfet::alter_node_voltages	19.0%
ngSpice_Command	7.9%
mosfet::evaluate_nonlinear	7.3%
populate_fft	3.5%
cblas_sgemv	0.5%

(a) using ngSPICE

Function	%
main	17.1%
LU_decomposition	6.5%
circuit::populate_J	3.7%
backward_solve	3.7%
forward_solve	2.5%
populate_fft	0.3%

(b) using simple Square Law

Figure 8.10: Performance Profiling results with the two different Model Evaluations

From the two profiling results 8.10a and 8.10b, we can see that when using ngSPICE the performance-limiting step becomes the ngSPICE calls, especially the string commands which shows that the ngSPICE parser interface degrades performance.

Chapter 9

Future Work

The aim of the work is to spawn new opportunities in exploring the optimization of this algorithm. This can be done in numerous ways and some of them are listed below.

1. The implementation is still not totally complete. More devices need to be supported in the simulator portion such as inductors, BJTs, etc. Also ngspice component of the program has not yet been integrated into the program.

2. Preprocessing of Jacobian: Circuit matrices still sometimes cause cancellation producing singular diagonal blocks which can be solved using a DC Analysis based static permutation which would also make the solution and convergence more stable.

3. Explore Krylov subspace techniques, iterative solvers and preconditioners as an alternate to the direct LU factorization [4, 24]. Post-Layout extracted circuits tend to be large and hence a direct approach may not be suitable.

4. Using a static schedule and block matrix operation kernels implemented on an FPGA this algorithm can be well suited to be implemented on FPGAs and for larger circuits, on networks of FPGAs as well.

5. As shown in the Results and Discussions section, a large part of the time is spent in model evaluation especially when using SPICE like models. This can be improved by using a LUT based approach to model evaluation. Instead of implementing the complex functions with multiple model parameters, the device characteristics are evaluated at discrete points and tri-cubic polynomial coefficients stored. At evaluation time in the simulator, a simple tri-cubic spline interpolation is performed and the result obtained quickly. This allows for immense parallelization and is very well suited to implementation even on FPGAs.

Part II

Circuit Partitioning for Distributed Logic Simulation

Chapter 10

Introduction

Digital logic circuits are getting ever larger with the transistor scaling and cheaper technology and testing and verification of these circuits is getting even harder. Logic simulation is a very time consuming process and is very much slower than the actual hardware. A fast way to implement this is to emulate the circuit in an FPGA, but increasing sizes of design, it is becoming increasingly difficult to find a large enough FPGA to fit the entire design.

This brings about the problem of partitioning the circuit to allow for distributed simulations across multiple FPGA's, processor cores or different processors altogether over a communication network. Circuit networks and logic netlists are easily amenable to the use of graph partitioning problems for netlist partitioning. There are several levels where the partitioning can be performed such as the RTL level and the gate level. RTL level partitioning can yield better partitions because of the higher level view of the functions and blocks, but describing RTL in a generic format and making a general purpose partitioning tool can be cumbersome and hence a gate level partitioning is performed where the netlist can be readily represented in a graph structure.

Chapter 11

Previous Work and Motivation

This work directly takes off from the yet unpublished work of Vinay B.Y. Kumar, which includes a toolset for custom Network-On-Chip (NOC) generation. The project aims at compiled code simulation of a logic circuit on a custom generated NOC. For the purpose of simulation of logic circuits on Intel Xeon Phi platform, the logic netlists are first clubbed into 9 input LUT which fits into the 512 Bit SIMD vector units allowing bit wise operations and access to be performed quickly.

First, the verilog netlist is parsed into the BLIF format using the yosys [25] tool followed by the LUT mapping by ABC [26]. With the LUT Mapping complete, a compressed netlist can be generated. The goal now is to split the circuit at its pseudo-primary and primary outputs, that is clustering of the outputs such that the overlap between the input cones of the clusters (Refer Figure 11.1) is minimised. The algorithm implemented is a greedy splitting algorithm which is not optimized and not tested against any other possible approach.

There have been several previous explorations on techniques to partition circuits since a decade ago when logic circuit sizes began to exceed single FPGA dies and had to be split into multiple units for emulation. Some of the works [27, 28] describe Ratio-Cut and set based techniques to perform clustering on circuits. And some well studied algorithms such as the Kernighan–Lin algorithm and its heuristic improvement the Fiduccia-Mattheyses algorithm have been applied in several CAD tools for layout and routing of digital circuits.

In this work, we are exploring an approach to speed up the process by only looking at the output nodes and not at the entire circuit. This approach will not only limit communication to flip-flop outputs but also reduce the exploration set of possible partitions. We have tested algorithms based on sub-modular functions, and spectral partitioning, to compare the performance with the simple greedy splitting algorithm based on various

metrics of replication, communication (Figure 11.2 shows how partitioning may require communication across time steps) and balance.

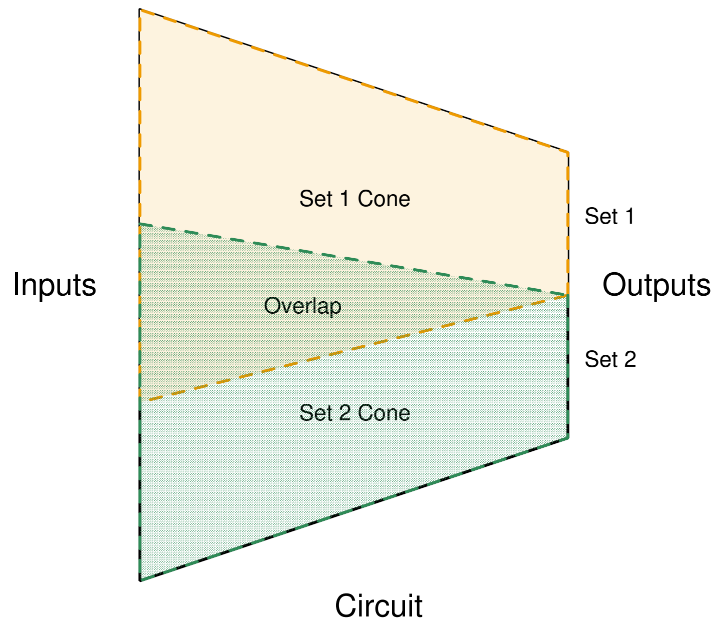

Figure 11.1: Example of Partitioning and Minimization Problem

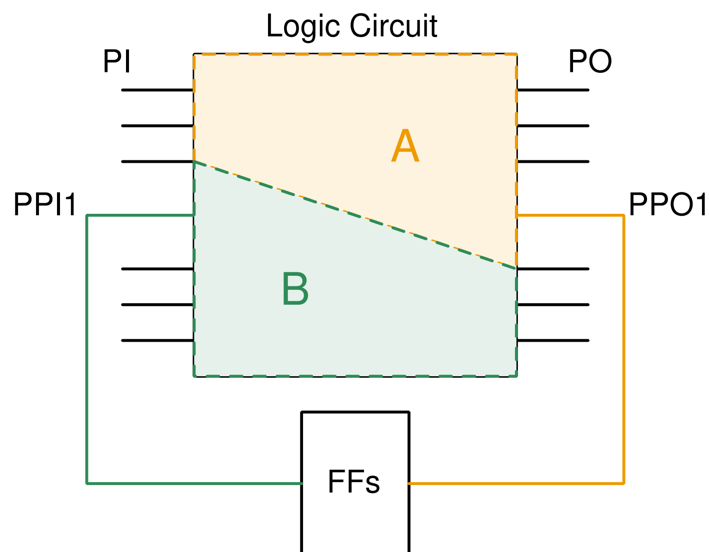

Figure 11.2: Example of Partitioning and Communication Problem

Chapter 12

Sub-Modular Function based Minimization

12.1 Sub-Modular Set Functions

Loosely, sub-modular set functions are functions on sets such that the incremental difference addition of an element to the input set makes, reduces as the size of the set increases. In other words, they satisfy the law of diminishing returns which makes them applicable to many scenarios such as electrical networks and graph theory. With the advent of the age of machine learning and AI, the sub-modular set functions are also gaining significant popularity as they are being modified and applied to learning problems such as feature selection, and even unsupervised learning.

Formally, given a finite set S, a submodular function is a set function $f : 2^S \rightarrow \mathbb{R}$ where 2_S is the power set such that it satisfies the following condition [29]:
For every $A, B \subseteq S$ with $A \subseteq B$ and every $x \in S \backslash B$ we have that

$$f(A \cup \{x\}) - f(A) \geqslant f(B \cup \{x\}) - f(B)$$

Sub-Modular functions share properties like convex functions and the simplest minimization problem of finding a minimizing set is computable in polynomial time [30].

12.2 Matlab SFO Toolbox [2]

For the purpose of performing the sub-modular minimization, the SFO Toolbox [2] available for MATLAB was used. The following function by Zhao, et.al [31] for greedy splitting for approximating multiway partitions is used:

```
function P = sfo_greedy_splitting(E,V,k)
```

where k is the number of partitions, V in the vertex set of the graph and E is called the oracle function which is supposed be the sub-modular set function which is optimized.

12.2.1 Queyranne's Algorithm

The greedy splitting algorithm at it's core performs a bi-partition based on Queyranne's Algorithm [32] which is shown in Algorithm 4.

Algorithm 4: Queyranne's Algorithm [32]

Input: V, f; `// f: Symmetric Oracle function`
Output: A^i

1 **for** $h \leftarrow 1$ *to* $|V| - 1$ **do**
2 $\quad (S_t, S_u) \leftarrow$ Pendent-Pair$[V', f]$;
3 $\quad$ store $A^h \leftarrow S_u$ and $s^h \leftarrow f(S_u)$;
4 $\quad$ update $S_t \leftarrow S_t \cup S_u$ and $V' \leftarrow V' \backslash S_u$;
5 **end**
6 Choose $i \in \arg\min s^h : h = 1, \cdots, |V| - 1$;

A pendant-pair is an ordered pair (S_t, S_u) of the vertices V of a graph, such that u is a minimizer for the Oracle function among all groups of nodes containing S_u but not S_t.

12.2.2 The Oracle Function

The oracle function here has been chosen simply to be the number of nodes in the input cone of the cluster of outputs supplied as input.

This makes sense because within the greedy splitting function, the oracle function is symmetrized as follows:

```
Fsym = E(A) + E(V\A) - E(V)
```

which is basically the overlap between the cluster and its complement which is what was sought to be optimized.

Chapter 13

Ratio-Cut using Spectral Partitioning

13.1 Spectral Graph Theory

The original algorithm and the proof was proposed by Fiedler [33, 34]. It has been used to partition graphs for many purposes such as even fill-in reduction ordering for sparse matrices [35], as an alternate to minimum degree ordering other than the usual ratio-cut applications in CAD.

Given an undirected graph $G(V, E)$ with weighted edges, A the symmetric adjacency matrix such that $A(i, j)$ is the weight of the edge (i, j) or 0 if the edge does not exist in the graph, and D the diagonal degree matrix which contains the sum of wights of all edges connected to a vertex i, at the ith diagonal element. The Laplacian Matrix L is defined as $L = D - A$. The ratio cut partitions the graph (say into X and Y) using the second eigenvalue of the Laplacian matrix such that the ratio shown in below is minimized.

$$\frac{|E(G) \cap (X \times Y)|}{|X| \cdot |Y|} \tag{13.1}$$

From Equation 13.1 we can observe that along with minimizing the cut between the partitions, it also takes into consideration the balance between the partitions represented by the product of the size of the partitions in the denominator, which is to be maximized.

The Laplacian matrix has the properties [8]:

- It is symmetric which implies that all eigenvalues are real and the corresponding eigenvectors, real and orthogonal.
- The sum of rows of the matrix is $[0, 0, \cdots, 0]$

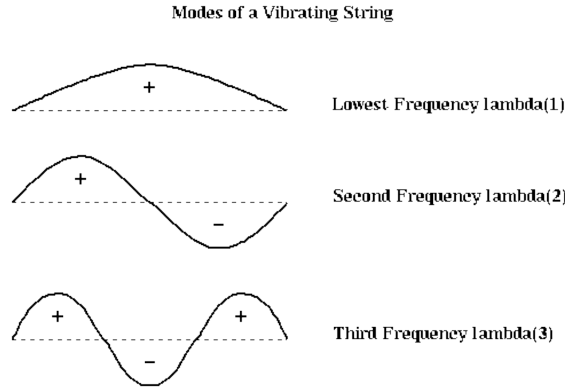

Figure 13.1: Modes of a Vibrating String [8]

- The number of connected components in the graph is given by the number of eigenvalues which are 0.
- The smallest eigenvalue is 0, which is indicative of the fact that the entire graph has at least one connected component which is itself.

Without going into the details of the proofs and derivations which are well explained in the referenced texts, the second smallest eigenvalue provides the lower bound on the ratio-cut metric as defined in Equation 13.1. The corresponding eigenvector is used to bipartition the graph based on the signs of the entry as outlined in Algorithm 5. Further partitions can be obtained by using more eigenvalues and their corresponding eigenvectors to create multi-way partitions using clustering algorithms [36, 37].

Algorithm 5: Ratio-Cut based on Spectral Bisection

Input: $G(V, E)$
Output: X, Y

1 $L = D - A$; `// Compute the Laplacian`
2 Compute the eigenvector v_2 ; `// Corresponding to` λ_2
3 **for** *each node n in G* **do**
4   **if** $v2[n] < 0$ **then**
5   Add node n to Partition X
6   **else**
7   Add node n to Partition Y
8   **end**
9 **end**

This kind of partitioning can be motivated using an analogy with a vibrating string as shown in Figure 13.1.

13.2 Circuit Conditioning

As discussed in the introduction, in this work, we are not looking to partition the circuit based on the individual gates which is easily represented as a graph. There is no unique way to convert the set of outputs and flip-flops into a graph which takes into consideration the metrics such as replication and communication which we are looking to optimize for.

13.2.1 Replication

A heuristic approach is used to encode the replication metric into the graph where the nodes are the outputs of the logic circuit. An exact overlap cannot be put in terms of the graph as the overlap does not simply sum when considering a set of outputs in a partition. A pair wise overlap is computed between the outputs and is assigned to be the weight of the edge between the two nodes.

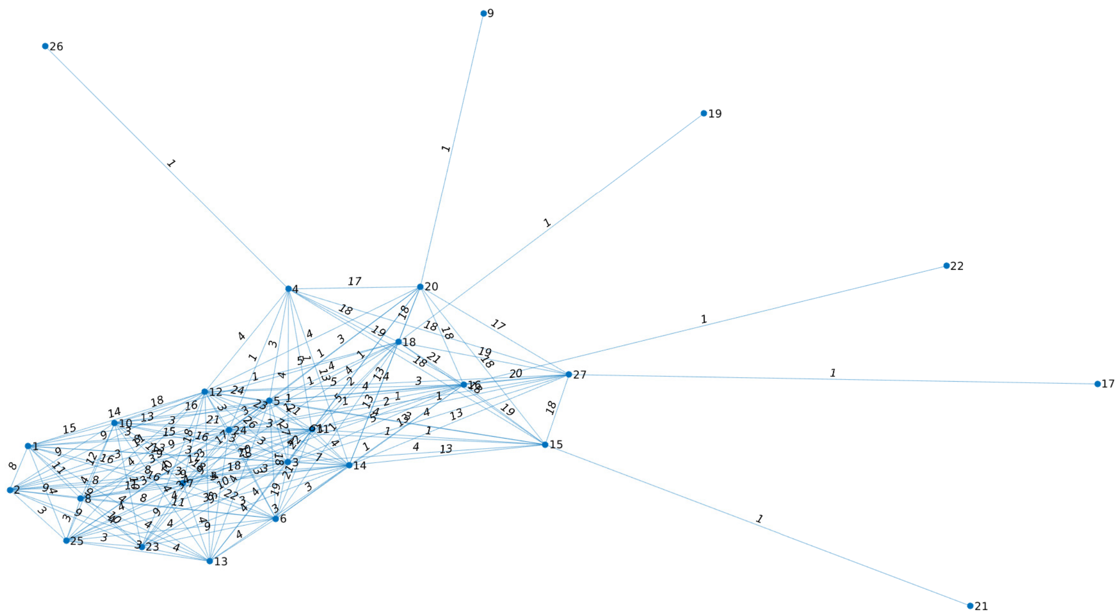

Figure 13.2: Example Graph of s526 circuit from ISCAS'85 benchmarks representing input to Spectral Partitioning embedding Replication metric information

13.2.2 Communication

Unlike replication, communication can be readily embedded into a directed graph structure. The fan-in of each pseudo-primary output is computed and an edge is introduced between the output and any pseudo-primary input contained in its fan-in.

But this would produce a directed graph, which cannot be used in spectral partitioning as the Laplacian of a directed graph might have complex eigenvalues as well. Hence to

symmetrize the matrix, each communication edge is added both ways and the weights summed up.

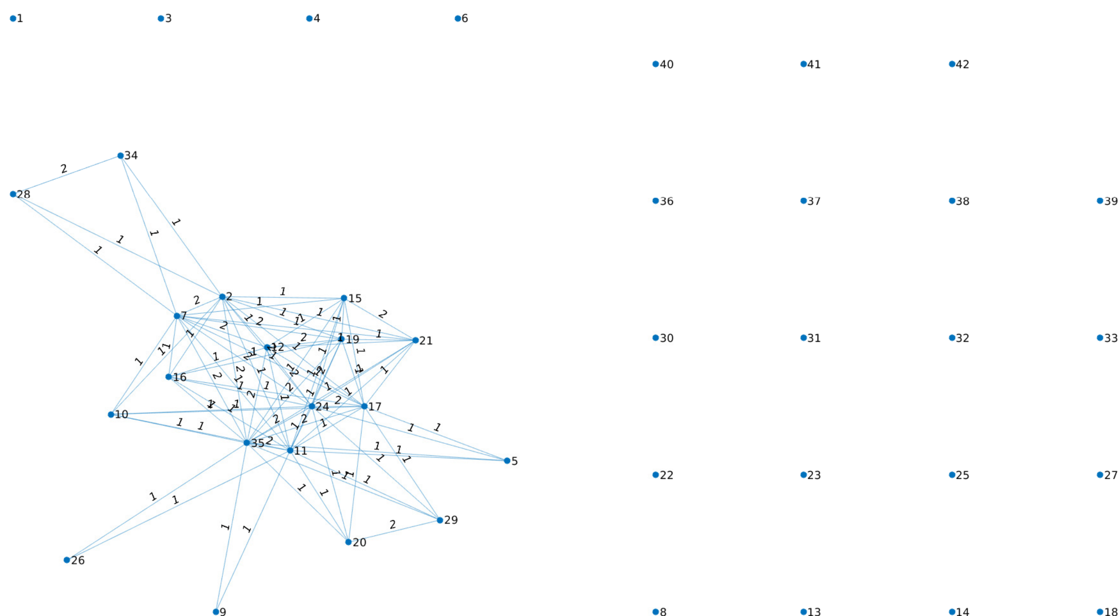

Figure 13.3: Example Graph of s713 circuit from ISCAS'85 benchmarks representing input to Spectral Partitioning embedding Communication metric information

13.2.3 Combining the Costs

Balancing comes naturally in spectral graph partitioning but replication and communication have been modeled as described in the preceding sub-sections. But for the goal of logic partitioning is necessary to take both into consideration as the same time. Also it might be the case that we may have some extra resources but cannot afford the extra latency of communication which occurs in every cycle of the logic simulation.

One way of doing this would be to normalize the Adjacency matrix obtained during the replication analysis and making the maximum element to 1 and simply adding the Adjacency matrix obtained by analysis of the communication to it. The weightages can now be introduced as multiplicative factors. But this method means that for a circuit which has maximum overlap much larger than another circuit, the overlap weightage is significantly scaled down to be brought to the same level. A more meaningful method would be to define a value of replication (r_C) which is equivalent to once unit of communication at each time step, based on the available resource and time constraints. We can write the new adjacency matrix as in Equation 13.2.

$$A = A_R/r_C + A_C \tag{13.2}$$

Chapter 14

Results and Discussion

To compare different algorithms and techniques, the metrics used are first defined here:

- Replication Metric: Sum of all gates/nodes in all the partitions. This includes the gates/nodes that are replicated among the partitions. This metric must be minimized.
- Communication Metric: Sum of all bits that must be sent from one partition to any other at the end of each clock cycle. A 'bit' broadcasted to multiple other partitions (for more than a bisection) are counted as many times as they are received. This metric also must be minimized.
- Balance Metric: Product of sizes of the partitions of the output nodes divided by the maximum possible product which is $(|V|/2)^2$. This metric must be maximized as it is a super-modular set function.

14.1 MATLAB interface with C++

The netlist was decoupled from the tool flow for the purpose of optimization, and was read into a C++ structure. But because the toolbox is present in MATLAB, interfacing between the two languages was necessary which was accomplished by using the MEX files or MATLAB executable. This allows C++ functions to be compiled in MATLAB and called as if they were MATLAB functions.

Because the MEX retains the memory allocation of the C++ code across calls to the function, the netlist only needs to read once followed by faster access to the already generated and stored netlist.

14.2 Greedy Splitting using SFO Toolbox

Here the basic greedy algorithm as implemented in the previous work in compared with the sub-modular function based optimization. Only the replication metric is provided as the cost function and hence only that metric is compared in Table 14.1.

Although from looking at the table, it might appear that the SFO toolbox based partitioning algorithm beats the other algorithm in most of the cases, there are some considerations not captured above.

1. Time taken by the toolbox is very large and increases tremendously with the number of output nodes as it is of the order $O(N^3)$.
2. The Oracle function provided suffers from a flaw, that there is no cost for lack of balance in the partitions.

Circuit	**Original Algorithm**			**SFO Toolbox Algorithm**		
Partitions →	2	3	4	2	3	4
c432	435	594	-	262	332	491
c499	292	390	488	292	390	488
c880	390	512	541	297	297	299
c1355	292	390	488	292	390	488
c1908	313	396	477	297	360	423
c2670	1139	1227	1302	984	984	984
c3540	628	791	1007	388	400	416
c5315	1411	1676	2015	1142	1142	1142

Table 14.1: Comparison of replication metric of Partitioning Algorithms on ISCAS Combinational Benchmarks

14.3 Spectral Partitioning

Here the sub-modular function based optimization is compared with the Spectral bipartitioning approach. Only the replication metric is provided as the cost function for the SFO toolbox, while the spectral approach incorporates balance as well which is compared in Table 14.2.

Circuit	**SFO Toolbox**		**Spectral Bipartition**	
Metric →	Replication	Balance	Replication	Balance
c432	262	0.49	262	0.49
c499	292	0.12	292	1
c880	297	0.15	297	0.41
c1355	292	0.12	292	1
c1908	297	0.15	311	0.92
c2670	984	0.03	984	0.9
c3540	388	0.17	388	0.17
c5315	1142	0.1	1142	0.45

Table 14.2: Comparison of Replication and Balance metric of Partitioning Algorithms on ISCAS Combinational Benchmarks

From the comparison we can clearly see that the spectral algorithms is able to achieve better balancing as compared to the greedy splitting algorithm from the SFO Toolbox in most cases at minimal increase in the replication metric. Circuits such as `c432` show no increase in balance, because the overlaps between the output nodes are such and can be observed in Figure 14.1. Node 4 is removed into a separate partition as it has the least weights while all other nodes are all to all connected with large weigths and hence splitting them to improve balance would come at a large cost of replication of nodes.

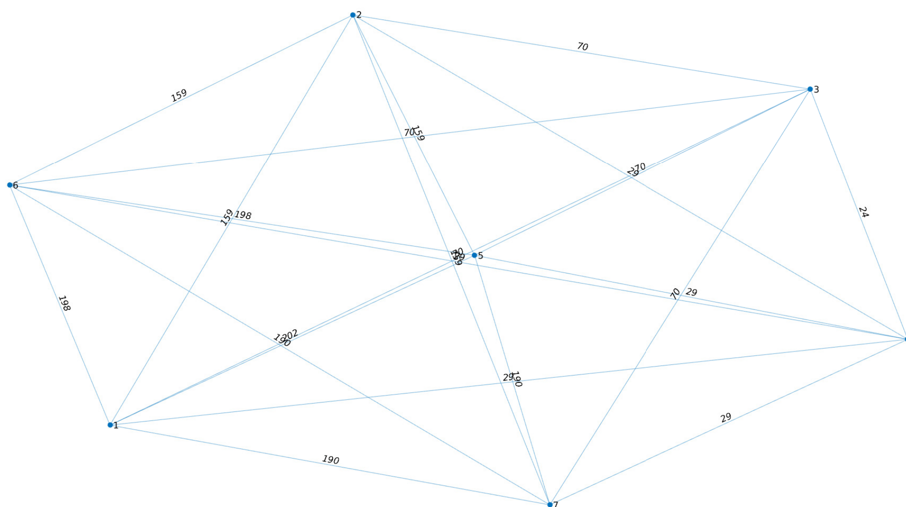

Figure 14.1: Example Circuit c432 showing lack of good Balanced Partitions

14.4 Spectral Partitioning with Communication

Here the Spectral bi-partitioning approach considering only replication is compared with the Spectral bi-partitioning approach when taking replication and communication into consideration. As mentioned earlier, for all circuits, the adjacency matrix obtained for replication is normalized before adding to the communication embedded adjacency matrix.

As defining communication requires FFs to be present and mapped between the inputs and outputs, the sequential circuits from ISCAS'85 benchmarks are used here. All the three metrics are compared in Table 14.3.

Circuit	**w/o Communication**			**w/ Communication**		
Metric →	Replication	Balance	Comm.	Replication	Balance	Comm.
s344	224	0.71	5	224	0.62	5
s349	225	0.71	5	225	0.71	5
s420	271	0.22	1	286	0.22	0
s526	265	0.4	6	262	0.27	6
s838	547	0.12	1	578	0.12	0
s9234	6094	0.14	4	6094	0.18	0
s13207	9441	0.49	69	9441	0.15	0
s15850	11067	0.17	16	11067	0.15	0
s35932	21017	0.93	178	21008	0.78	165
s38417	25585	0.26	214	25585	0.005	0
s38584	22447	0.014	1	2247	0.012	0

Table 14.3: Comparison of All metrics of the Spectral Partitioning Algorithm on ISCAS Sequential Benchmarks

For the comparison we can clearly see that after introducing the communication metric, the communication has reduced compared to when only replication was being considered. In many cases, the number of nodes have taken a significant hit at the cost of obtaining less communication. This shows that the weightages still need to be fine tuned and an appropriate method which guarantees close to expected results.

In some circuits such as `s420` we can see that introducing communication also improves replication. This brings a certain instability in the spectral bipartition algorithms specifically because the edges are not exactly representative of the overlap when considering a group of nodes.

Chapter 15

Future Work

In this work, only a small subset of algorithms have been tests to implement circuit partitioning. Some other algorithms are also being worked out and tested such as the sub-super-modular optimization algorithm which optimized the difference of sub-modular and super-modular function. This can be used to introduce the balance metric which is a super-modular function into the SFO toolbox based approach to obtain results comparable to the spectral partitioning method.

Different other graph algorithms such as those provided by the METIS which contains a set of graph partitioning programs which have been tried and tested. Clustering algorithms such as k-means can also be tested using multiple eigen-values in the spectral partitioning algorithm.

Apart from the improvements to the partitioning algorithm, the partitions produced need to be integrated into the toolflow to produce and simulate the distributed logic created. Automatic generation of the partitioned logic and simulation on a network using multiple FPGAs must be implemented to put these partitioning algorithms' results to use and get results from the actual implementation.

Part III

Appendix

Appendix I

Harmonic Balance C++ Codes

I.1 Sparse Matrix Storage

I.1.1 sparseCOO.h

```
1  #ifndef SPARSECOO_H
2  #define SPARSECOO_H
3
4  #include "stdafx.h"
5
6  class sparseCOO{
7  public:
8      uint64_t rowCount, columnCount;
9      vector<pair<pair<uint64_t, uint64_t>, float_t> > COO;
10     static bool rowCompare(const pair<pair<uint64_t, uint64_t>, float_t>& a, const
       ↪  pair<pair<uint64_t, uint64_t>, float_t>& b);
11     static bool columnCompare(const pair<pair<uint64_t, uint64_t>, float_t>& a, const
       ↪  pair<pair<uint64_t, uint64_t>, float_t>& b);
12
13 public:
14     //Generate the Sparse COO represetation using the CSV file generated by the
15     //MATLAB exe file. The source of the data is a mat file supplied by the
16     //UFlorida Sparse Matrix Java Application
17     sparseCOO(string fileName);
18     void SpMV(float_t *x, float_t **y);
19     void prettyPrint();
20     void rowSort();
21     void columnSort();
22 };
23
24 #endif //SPARSECOO_H
```

I.1.2 sparseCSC.h

```
#ifndef SPARSECSC_H
#define SPARSECSC_H

#include "stdafx.h"
#include "sparseCOO.h"

class sparseCSC {
public:
    uint64_t rowCount = 0, columnCount = 0;
    int *col_permutation, *row_permutation;
    uint64_t nnz = 0, nnz_max = 0;
    bool symbolic = false, permuted = false;
    float_t *A;          //A - Non-Zero Elements
    uint64_t *IA, *JA;   //IA - Row Indices; JA - Column start indices
public:
    sparseCSC(uint64_t rowCount, uint64_t columnCount, bool symbolic = true, uint64_t
    ↪  nnz_guess = 0);
    sparseCSC(float_t* sparseMatrix, uint64_t x, uint64_t  y);
    sparseCSC(sparseCOO* input, bool symbolic = false);

    void insert_element(uint64_t row);
    bool nextElement(uint32_t *row, uint32_t *column, uint32_t *offset, float_t *num);
    void do_colamd();
    void prettyPrint();
    void spy(string header, bool clean = true);
    void clear_permutation();
    void symbolic_lu(sparseCSC *L, sparseCSC *U);
};

#endif //SPARSECSC_H
```

I.1.3 sparseBCCS.h

```
#ifndef SPARSEBCCS_H
#define SPARSEBCCS_H

#include <stdafx.h>
#include "sparseCOO.h"

class sparseBCCS{
public:
    uint64_t rowCount, columnCount, numBlocks, rowBlocks;
    pair<uint64_t, uint64_t> blockSize;
```

```
    vector<pair<uint64_t, float_t*> > blocks;
    vector<uint64_t> blockColPointer;

    bool banded = false, permuted = false;

    // Column Permutation: New to Old
    // Row Permutation:
    //  Column 1: Old to New Mapping
    //  Column 2: New to Old Mapping
    int *col_permutation, *row_permutation;

    sparseBCCS();
    sparseBCCS(uint64_t rowCount, uint64_t columnCount, pair<uint64_t, uint64_t>
    ↪  blockSize);
    void initialize(uint64_t rowCount, uint64_t columnCount, pair<uint64_t, uint64_t>
    ↪  blockSize);

    sparseBCCS(sparseCOO* input, pair<uint64_t, uint64_t> blockSize);

    // Makes the Row Permutation permanent by altering the sparse structure
    void applyPermutation();

    static uint64_t getNumBlocks(sparseCOO* input, pair<uint64_t, uint64_t>
    ↪  blockSize);
    void SpMV(float_t *x, float_t *y);
    void prettyPrint(bool symbolic = true);
};

bool compareBlocks(pair<uint64_t, float_t *> a, pair<uint64_t, float_t *> b);

#endif //SPARSEBCCS_H
```

I.2 Circuit Parser and Evaluation

I.2.1 ngspice.h

```
#pragma once

#ifndef _NGSPICE_H_
#define _NGSPICE_H_

#include <stdafx.h>

#define PRINT_OUT 0
#define PRINT_STATUS 0
```

```
10  #define FILE_OUT 0
11  #define FILE_STATUS 0
12
13  namespace ngspice{
14      extern std::ofstream ofile_out, ofile_err, ofile_stat;
15
16      int controlledExit(int exit_status, bool unload, bool exit_type, int id, void
        ↪  *caller);
17      int sendChar(char *out, int id, void *caller);
18      int sendStat(char *status, int id, void *caller);
19      int sendData(vecvaluesall *data_vec, int count, int id, void *caller);
20      int sendInitData(vecinfoall *data_addr, int id, void *caller);
21      int bgThreadRunning(bool running, int id, void *caller);
22  }
23
24  #endif //_NGSPICE_H_
```

I.2.2 device.h

```
1   #pragma once
2
3   #ifndef DEVICE_H
4   #define DEVICE_H
5
6   #include <stdafx.h>
7
8   class device {
9
10  protected:
11      static float_t fund;
12
13      uint8_t num_ports, num_parameters;
14      uint64_t *ports;
15      bool type;
16      string d_name;
17
18      // Changes meaning based on type of Device
19      float_t *parameters;
20
21      // Y and J Block Indices which need to be updated
22      // Inherent ordering of blocks depending on the type of device
23      vector<float_t *> Y_blocks, J_blocks;
24
25      // Function defined as virtual as the devices of all
26      // types are going to be stored as generic device pointers
```

```
27
28      // Default Base Class function, initializes to Identity Band Block
29      virtual void populate_band (float_t *band_block, bool pos_term);
30
31      // Default Base Class function, initializes to Identity Block
32      virtual void populate_block (float_t *block, bool pos_term);
33      device (string name, uint8_t num_ports, uint64_t *ports, float_t *parameters,
34          int num_parameters = 1, bool type = false);
35      ~device ();
36
37  public:
38      virtual void setBlockPointers (float_t *Y_block, float_t *J_block, uint64_t row,
        ↪  uint8_t port_index);
39
40      static void set_frequency (float_t frequency);
41      static float_t get_frequency ();
42
43      virtual void populate_B (float_t *B);
44
45      // Store the contribution of the device as a function of the "node" potential
46      // to KCLs at various other nodes in the circuit in the preallocated "rows" array
47      // Port Index supplies which port of the device connected to the node
48      // Return value is the number of such rows
49      virtual uint8_t which_rows_Y(uint64_t *rows, uint8_t port_index = 0);
50      virtual uint8_t which_rows_J(uint64_t *rows, uint8_t port_index = 0);
51
52      virtual void populate_Y ();
53      virtual void populate_J ();
54
55      // Doubles as populate_J function for non-linear elements
56      virtual void alter_node_voltages(float_t *x = NULL, uint64_t time_index = 0);
57      virtual void evaluate_nonlinear(float_t *x = NULL, float_t *f_x = NULL, uint64_t
        ↪  time_index = 0);
58  };
59
60
61  /*
62   *  num_ports is set to 2 by default;
63   *  Extra Functions to set the B vector on the RHS
64  */
65  class source : public device {
66
67      // type variable
68      // 0 - Voltage Source; 1 - Current Source
69  public:
70
71      // Default only 1 paramater for DC Source
```

```
72      // Further Parameters of AC Source - Frequency decided for the entire circuit
73      // Ports: 3 for voltage source including the current through voltage
74      //  source variable as the 3rd "virtual" port;
75      //  2 for current source
76      source (string name, uint64_t *ports, float_t *parameters,
77          int num_parameters = 1, bool type = false);
78
79      uint8_t which_rows_Y(uint64_t *rows, uint8_t port_index = 0);
80      void setBlockPointers (float_t *Y_block, float_t *J_block, uint64_t row, uint8_t
        ↪  port_index);
81
82      // Populates the relevant H-length port of the B vector
83      void populate_B (float_t *B);
84  };
85
86  /*
87   *  num_ports is set to 2 by default
88  */
89  class resistor : public device {
90
91  private:
92      void populate_band (float_t *band_block, bool pos_term = true);
93      void populate_block (float_t *block, bool pos_term = true);
94
95  public:
96      // Only one resistance parameter
97      resistor (string name, uint64_t ports[2], float_t parameter);
98  };
99
100 /*
101  *  num_ports is set to 2 by default
102 */
103 class capacitor : public device {
104
105 private:
106     void populate_band (float_t *band_block, bool pos_term = true);
107     void populate_block (float_t *block, bool pos_term = true);
108
109 public:
110     // Only one resistance parameter
111     capacitor (string name, uint64_t ports[3], float_t parameter);
112 };
113
114 /*
115  *  num_ports is set to 2 by default
116 */
117 class inductor : public device {
```

```
118
119 private:
120     void populate_band (float_t *band_block, bool pos_term = true, bool unit = false);
121     void populate_block (float_t *block, bool pos_term = true, bool unit = false);
122
123 public:
124     // Only one resistance parameter
125     inductor (string name, uint64_t ports[3], float_t parameter);
126
127     uint8_t which_rows_Y(uint64_t *rows, uint8_t port_index = 0);
128     void setBlockPointers (float_t *Y_block, float_t *J_block, uint64_t row, uint8_t
        ↪  port_index);
129
130     // Needs its own function because of the uniqueness
131     void populate_Y ();
132     void populate_J ();
133 };
134
135 /*
136  *  num_ports is set to 4 by default
137 */
138 class mosfet : public device {
139
140 private:
141     // Technology Parameters
142     static const float_t mu_n;
143     static const float_t mu_p;
144
145     static const float_t C_ox;
146     static const float_t lambda;
147
148     static const float_t vthn0;
149     static const float_t vthp0;
150
151     // Port Names
152     static const uint8_t DRAIN, GATE, SOURCE, BODY;
153
154     float_t gate[H * H], drain[H * H], source[H * H];
155
156     char command[512];
157     // parameters[0] - W
158     // parameters[1] - L
159
160     // type = false : NMOS; true  : PMOS
161
162     // ports[0] - D; ports[1] - G
163     // ports[2] - S; ports[3] - B
```

```
164
165     //public: mosfet::MOS IV(float_t D, float_t G, float_t S, float_t B = 0);
166
167 public:
168     // Two parameters only for now, with different ways of initialization as suitable
169     mosfet (string name, uint64_t ports[4], float_t W, float_t L, bool type = false);
170     mosfet (string name, uint64_t ports[4], float_t *parameters, bool type = false);
171
172     // Return 0 as no Y entry for non-linear devices
173     uint8_t which_rows_Y (uint64_t *rows, uint8_t port_index = 0);
174
175     // Store the contribution of the device as a function of the "node" potential
176     // to KCLs at various other nodes in the circuit in the preallocated "rows" array
177     // Port Index supplies which port of the device connected to the node
178     // Return value is the number of such rows
179     uint8_t which_rows_J (uint64_t *rows, uint8_t port_index = 0);
180
181     void setBlockPointers (float_t *Y_block, float_t *J_block, uint64_t row, uint8_t
        ↪  port_index);
182
183     void alter_node_voltages(float_t *x = NULL, uint64_t time_index = 0);
184     void evaluate_nonlinear( float_t *x, float_t *f_x, uint64_t time_index);
185 };
186 #endif // DEVICE_H
```

I.2.3 circuit.h

```
1  #pragma once
2
3  #ifndef CIRCUIT_H
4  #define CIRCUIT_H
5
6  #include <stdafx.h>
7  #include "device.h"
8  #include "ngspice.h"
9  #include "sparse/sparseBCCS.h"
10
11 #define DELIMITERS " ,()\t\r\n"
12
13 class circuit {
14
15 private:
16     string name, spfile_name;
17     static bool ngspice_running;
18
```

```
19      unordered_map <string, uint64_t> node_map;
20      vector <string> node_names;
21
22      vector <device *> sources, linear, nonlinear;
23
24      vector<vector<string> > nodes_to_plot;
25
26      // Holds the devices at each node and
27      // the port of the device connected to that node
28      vector<vector <pair<device *, uint8_t> > > node_devices;
29
30      void clear_J();
31      void clear_Y();
32
33  public:
34      static void clear_block(float_t *block, bool band = false);
35
36      // Linear Circuit Sparse-Blocked Matrix
37      sparseBCCS Y;
38      // Jacobian Matrix combining Y and Non-Linear Elements
39      sparseBCCS J;
40      // Constant Sources (LHS of MNA Equation)
41      float_t *B;
42
43      circuit(const char* cirfile, const char *file_name);
44      void structure_YJ();
45
46      void populate_Y();
47      void populate_J(float_t *x, float_t *f_x);
48      void populate_B();
49
50      uint64_t get_num_nodes();
51      string get_node_name(uint64_t node);
52      bool store_plot_command(const char* file_name);
53
54      ~circuit();
55  };
56
57  #endif // CIRCUIT_H
```

I.3 Extracting Column Parallelism

I.3.1 Plotting Column Dependency Graph

```
410 #ifndef NOGRAPHVIZ
411 // Plot the Dependency Graph using graphviz in the dot format
412 int plot_dependency_graph(const char *fname, uint64_t num_cols) {
413 
414     char file_name[256];
415     strcpy(file_name, fname);
416 
417     Agraph_t *graph;
418     Agnode_t *nodes[num_cols];
419     Agedge_t *edge;
420 
421     static GVC_t *gvc;
422     if (!gvc)
423         gvc = gvContext();
424 
425     // Create a Strict Directed Graph
426     graph = agopen(file_name, Agstrictdirected, NULL);
427 
428     // Create all the nodes
429     for (uint64_t col = 0; col < num_cols; col++) {
430         nodes[col] = agnode(graph, &(to_string(col + 1)[0]), TRUE);
431         agsafeset(nodes[col], "width", ".5", "");
432     }
433 
434     // Create the edges
435     for (uint64_t node_start = 0; node_start < num_cols; node_start++) {
436         for (auto node_end : adj_list[node_start]) {
437             edge = agedge(graph, nodes[node_start], nodes[node_end],
438                 &((to_string(node_start) + "_" + to_string(node_end))[0]), TRUE);
439         }
440     }
441 
442     // Use the directed graph layout engine
443     gvLayout(gvc, graph, "dot");
444 
445     FILE *fp;
446     fp = fopen((string(file_name) + ".dot").c_str(), "w");
447 
448     // Store the plot output
449     gvRender(gvc, graph, "dot", fp);
450     fclose(fp);
451     gvFreeLayout(gvc, graph);
```

```
452     agclose(graph);
453
454     system(("dot -Tps " + (string(file_name) + ".dot") + " -o " + string(file_name) +
      ↪  ".eps").c_str());
455
456     return (gvFreeContext(gvc));
457 }
458 #endif
```

I.3.2 Get Next Column

```
170 uint64_t get_next_column(uint64_t col_done) {
171
172     // Define a critical section to avoid corruption
173     #pragma omp critical (ready_cols)
174     if (col_done != -1) {
175         // Code implemented by the last returning thread
176         if (++num_cols_done == N) {
177             int waiting_threads = omp_get_num_threads();
178             for (int i = 0; i < waiting_threads; i++)
179                 sem_post(&ready_sem);
180         }
181         for (auto col : adj_list[col_done]) {
182             if ((--dependency_count[col]) == 0) {
183                 ready_cols.push(col);
184                 sem_post(&ready_sem);
185             }
186         }
187     }
188
189     uint64_t next_col = -1;
190     // No ready column available
191     // Find a better solution than polling for ready columns
192     // Semaphores would be the best option - Having the size of the queue
193     sem_wait(&ready_sem);
194     if (num_cols_done == N) return -1;
195
196     #pragma omp critical (ready_cols)
197     if (num_cols_done != N) {
198         next_col = ready_cols.front();
199         ready_cols.pop();
200     }
201     return next_col;
202 }
```

Appendix II

Circuit Partitioning C++ Codes

II.1 dcircuit.h

```
#ifndef DCIRCUIT_H
#define DCIRCUIT_H

#include <stdlib.h>
#include <vector>
#include <unordered_map>
#include <queue>
#include <string>
#include <stdlib.h>
#include <string.h>
#include <algorithm>
#include <sstream>
#include <fstream>
#include <omp.h>

//#include <unistd.h>

/*
 * node: Single gate in a digital circuit
 * Elements:
 *  id               : Node id in string format (As in the original netlist)
 *  type             : Type of node as defined in "logic" namespace
 *  input_count      : Number of inputs to the node
 *  output_count     : Number of outputs of the node (1 unless a FAN node)
 *  inverting        : Boolean describing the inverting nature of the node
 *  input_nodes      : Vector of input nodes, along with their position in the
 *                       input nodes' output node vector
 *  output_nodes     : Vector of output nodes, along with their positic in the
 *                       output nodes' input node vector
```

```
30  *  value           : 5-Valued Logic value of the outputs of the node (in same
    ↪  ordering)
31 */
32 struct node{
33     std::string id;
34     unsigned short type;
35     unsigned int input_count, output_count;
36     bool inverting;
37     std::vector<std::pair<node *, unsigned short> > input_nodes, output_nodes;
38     long int visited = 0;
39 
40     node();
41     ~node();
42 };
43 
44 /*
45  * dcircuit: Collection of nodes forming a circuit
46  * Elements:
47  *  circuit     : Vector of all nodes
48  *  inputs      : Vector of primary inputs of the circuit
49  *  outputs     : Vector of primary outputs of the circuit
50  *  id2node     : Reverse mapping of id to node (node also contains information about
    ↪  the id)
51  *
52  * Methods:
53  *  Constructor : Takes as input path of the intermediate ".ov" file to read the
    ↪  netlist from
54  */
55 class dcircuit{
56 public:
57     std::vector<node *> circuit;
58     std::vector<node *> inputs, outputs;
59     std::unordered_map<std::string, node*> id2node;
60     std::unordered_map<std::string, int> id2index;
61 
62     std::unordered_map<std::string, int> id2oindex;
63 
64     dcircuit(const char *);
65     ~dcircuit();
66 
67     // Obtain count of nodes in the cone above the output set
68     long times_traversed = 0;
69     int count_cone(node *output_node);
70     int count_cone(std::vector<int> output_set);
71     int count_cone(std::vector<node*> output_set);
72     int count_cone(std::vector<std::string> output_set);
73     void find_source_ff(node *gate, std::vector<int> *pin);
```

```
74      void reset_traversed();
75
76      void store_adjacency_csv(const char* file_name);
77      void store_outputindex_csv(const char* file_name);
78
79      std::queue<node *> assign_inputs(std::string* , unsigned short* , int);
80      std::queue<node *> assign_inputs(unsigned short*);
81      std::vector<std::string> faults();
82  };
83
```

Appendix III

BSIM4 Model

Fortunately, the latest BSIM4 models provide just what is required to perform the Harmonic Balance simulation. The list of available operating point parameters which are of importance to the method are listed in Table III.1

Table III.1: Operating Point Parameters available in BSIM4 Models

Parameter	**Parameter (in Model)**	**Expression**
Transconductance	gm	dI_D/dV_{GS}
Body Effect Transconductance	gmbs	dI_D/dV_{BS}
Drain-Source Conductance	gds	dI_D/dV_{DS}
Gate - Gate Capacitance	cgg	dQ_G/dV_G
Gate - Source Capacitance	cgs	dQ_G/dV_S
Gate - Drain Capacitance	cgd	dQ_G/dV_D
Gate - Body Capacitance	cgb	dQ_G/dV_B
Source - Gate Capacitance	csg	dQ_S/dV_G
Source - Source Capacitance	css	dQ_S/dV_S
Source - Drain Capacitance	csd	dQ_S/dV_D
Source - Body Capacitance	csb	dQ_S/dV_B
Drain - Gate Capacitance	cdg	dQ_D/dV_G
Drain - Source Capacitance	cds	dQ_D/dV_S
Drain - Drain Capacitance	cdd	dQ_D/dV_D
Drain - Body Capacitance	cdb	dQ_D/dV_B
Body - Gate Capacitance	cbg	dQ_B/dV_G
Body - Source Capacitance	cbs	dQ_B/dV_S
Body - Drain Capacitance	cbd	dQ_B/dV_D
Body - Body Capacitance	cbb	dQ_B/dV_B
Gate Charge	qg	Q_G
Source Charge	qs	Q_S
Drain Charge	qd	Q_D
Body Charge	qb	Q_B

Bibliography

[1] H. Vogt, H. Hendrix, and P. Nenzi, "Ngspice users manual, version 27 (describes ngspice-27 release version)," tech. rep., ngspice Project, September 2017.

[2] A. Krause, "Sfo: A toolbox for submodular function optimization," *J. Mach. Learn. Res.*, vol. 11, pp. 1141–1144, Mar. 2010.

[3] B. Bandali, "Steady-state analysis of nonlinear circuits using the harmonic balance on gpu," Master's thesis, University of Ottawa, 2014.

[4] P. Feldmann, B. Melville, and D. Long, "Efficient frequency domain analysis of large nonlinear analog circuits," in *Proceedings of Custom Integrated Circuits Conference*, pp. 461–464, May 1996.

[5] J. R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," *SIAM J. Sci. Stat. Comput.*, vol. 9, pp. 862–874, Sept. 1988.

[6] Xilinx Inc., "pragma HLS pipeline, SDAccel Development Environment Help," 2018. [Online].

[7] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition 2Nd Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2nd ed., 2016.

[8] J. Demmel, "CS267: Notes for Lecture 23 on Graph Partitioning, Part 2," April 1999.

[9] K. S. Kundert and A. Sangiovanni-Vincentelli, "Simulation of nonlinear circuits in the frequency domain," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 5, pp. 521–535, October 1986.

[10] G. W. Somers, "Acceleration of block-aware matrix factorization on heterogeneous platforms," Master's thesis, University of Ottawa, 2016.

[11] M. E. Brinson and S. Jahn, "Qucs: A gpl software package for circuit simulation, compact device modelling and circuit macromodelling from dc to rf and beyond,"

International Journal of Numerical Modelling: Electronic Networks, Devices and Fields, vol. 22, no. 4, pp. 297–319.

[12] M. Günther, U. Feldmann, and J. T. Maten, *Modelling and discretization of circuit problems*, vol. 0537 of *CASA-report*. Eindhoven: Technische Universiteit Eindhoven, 2005.

[13] G. Q. Zhang, F. van Roosmalen, and M. Graef, "The paradigm of "more than moore"," in *2005 6th International Conference on Electronic Packaging Technology*, pp. 17–24, Aug 2005.

[14] B. J. Sheu, D. L. Scharfetter, P.-K. Ko, and M.-C. Jeng, "Bsim: Berkeley short-channel igfet model for mos transistors," *IEEE Journal of Solid-State Circuits*, vol. 22, no. 4, pp. 558–566, 1987.

[15] T. Nechma and M. Zwolinski, "Parallel sparse matrix solution for circuit simulation on fpgas," *IEEE Transactions on Computers*, vol. 64, pp. 1090–1103, April 2015.

[16] P. R. Amestoy, T. A. Davis, and I. S. Duff, "An approximate minimum degree ordering algorithm," *SIAM Journal on Matrix Analysis and Applications*, vol. 17, no. 4, pp. 886–905, 1996.

[17] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng, "A column approximate minimum degree ordering algorithm," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 3, pp. 353–376, 2004.

[18] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull, "Graphviz — open source graph drawing tools," in *Lecture Notes in Computer Science*, pp. 483–484, Springer-Verlag, 2001.

[19] S. C. Woo, J. P. Singh, and J. L. Hennessy, *The performance advantages of integrating block data transfer in cache-coherent multiprocessors*, vol. 29. ACM, 1994.

[20] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[21] Wikipedia contributors, "Xeon phi — Wikipedia, the free encyclopedia," 2018. [Online].

[22] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 1–17, 1990.

[23] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, and A. Greenbaum, "Linear algebra package," 1999.

[24] H. G. Brachtendorf, G. Welsch, and R. Laur, "Fast simulation of the steady-state of circuits by the harmonic balance technique," in *Proceedings of ISCAS'95 - International Symposium on Circuits and Systems*, vol. 2, pp. 1388–1391 vol.2, April 1995.

[25] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[26] A. Mishchenko *et al.*, "Abc: A system for sequential synthesis and verification," *URL http://www. eecs. berkeley. edu/alanmi/abc*, p. 17, 2007.

[27] Nan-Chi Chou, Lung-Tien Liu, Chung-Kuan Cheng, Wei-Jin Dai, and R. Lindelof, "Circuit partitioning for huge logic emulation systems," in *31st Design Automation Conference*, pp. 244–249, June 1994.

[28] C. J. Alpert and A. B. Kahng, "Recent directions in netlist partitioning: a survey," *Integration*, vol. 19, no. 1, pp. 1 – 81, 1995.

[29] A. Schrijver, *Combinatorial Optimization*, vol. 24 of *Algorithms and Combinatorics*. Springer-Verlag Berlin Heidelberg, 1 ed., 2003.

[30] M. Grötschel, L. Lovász, and Schrijver, *A. Combinatorica*. Springer, 1981.

[31] L. Zhao, H. Nagamochi, and T. Ibaraki, "Greedy splitting algorithms for approximating multiway partition problems," *Mathematical Programming*, vol. 102, no. 1, pp. 167–183, 2005.

[32] M. Queyranne, "Minimizing symmetric submodular functions," *Mathematical Programming*, vol. 82, no. 1-2, pp. 3–12, 1998.

[33] M. Fiedler, "A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory," *Czechoslovak Mathematical Journal*, vol. 25, no. 4, pp. 619–633, 1975.

[34] M. Fiedler, "Laplacian of graphs and algebraic connectivity," *Banach Center Publications*, vol. 25, no. 1, pp. 57–70, 1989.

[35] A. Pothen, H. D. Simon, and K.-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM J. Matrix Anal. Appl.*, vol. 11, pp. 430–452, May 1990.

[36] U. Von Luxburg, "A tutorial on spectral clustering," *Statistics and computing*, vol. 17, no. 4, pp. 395–416, 2007.

[37] M. Naumov and T. Moon, "Parallel spectral graph partitioning," tech. rep., NVIDIA Technical Report, NVR-2016-001, 2016.