Steady-State Analysis of Nonlinear Circuits using the Harmonic Balance on GPU

Thesis Submitted to the Faculty of Graduate and Postdoctoral Studies

In partial fulfillment of the requirements for the degree of

Masters of Applied Science in

Electrical and Computer Engineering

© Bardia Bandali, Ottawa, Canada, 2013

uOttawa

School of Electrical Engineering and Computer Science

Faculty of Engineering

University of Ottawa

In the name of God, the Most Gracious, the Most Merciful

To my wife ***Roya****, for her vital love & support*
To our sons ***Amin*** *&* ***Hossein****, for the happiness they brought to our life*

Contents

List of Figures

List of Algorithms

List of Tables

List of Symbols

2D	Two Dimension
3D	Three Dimension
AFM	Artificial Frequency Mapping
API	Application Programming Interface
BE	Backward Euler
BKLU	Block-KLU
BWD	Backward substitutions
CAD	Computer-Aided Design
CCS	Compressed Column Sparse
CU	Compute Units
DAE	Differential Algebraic Equations
DFT	Discrete Fourier Transform
DFT	Discrete Fourier Transform
DRAM	Dynamic RAM
DSP	digital signal processors
FIR	Finite Impulse Response
FMT	Frequency Mapping Techniques

FWD	Forward substitution
GCN	Graphics Core Next
GEMM	GEneral Matrix Multiplication
Gflops	Giga FLoating point OPerations
GPGPU	General Purpose computing on GPU
GPRs	General Purpose Register
GPU	Graphics Processing Unit
HB	Harmonic Balance
Hybrid-BKLU	Hybrid Block KLU
IDFT	Inverse Discrete Fourier Transform
IR	Intermediate Representation
LFR	local register file
LO	Local Oscillator
MNA	Modified Nodal Analysis
NDRange	N-dimensional work space
NR	Newton-Raphson
OpenCL	Open Computing Language
PE	Processing Element
RF	Radio Frequency
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data

SPU	stream processing unit
SRF	stream register file
SSE	streaming SIMD extensions
TRSM	TRiangular Solve Matrix
UDP	Ultra Dispatch Processor
VLIW	Very Large Instruction Word

Abstract

This thesis describes a new approach to accelerate the simulation of the steady-state response of nonlinear circuits using the Harmonic Balance (HB) technique. The approach presented in this work focuses on direct factorization of the sparse Jacobian matrix of the HB nonlinear equations using a Graphics Processing Unit (GPU) platform. This approach exploits the heterogeneous structure of the Jacobian matrix. The computational core of the proposed approach is based on developing a block-wise version of the KLU factorization algorithm, where scalar arithmetic operations are replaced by block-aware matrix operations. For a large number of harmonics, or excitation tones, or both the Block-KLU (BKLU) approach effectively raises the ratio of floating-point operations to other operations and, therefore, becomes an ideal vehicle for implementation on a GPU-based platform.

Motivated by this fact, a GPU-based Hybrid Block KLU framework is developed to implement the BKLU. The proposed approach in this thesis is named *Hybrid-BKLU*. The Hybrid-BKLU is implemented in two parts, on the host CPU and on the graphic card's GPU, using the OpenCL heterogeneous parallel programming language. To show the efficiency of the Hybrid-BKLU approach, its performance is compared with BKLU approach performing HB analysis on several test circuits. The Hybrid-BKLU approach yields speedup by up to 89 times over conventional BKLU on CPU.

Acknowledgments

It is almost impossible to appreciate properly of gentle people who are helpful during a journey, specifically using a non-native language.

My deep appreciation is presented to ***Dr. Emad Gad****, my supervisor, for his comprehensive and continuous support during the work. I would like to thank* ***Dr. Miodrag Bolic*** *for his trust that provided the opportunity to prove myself.*

Also, my wife whose endless help and support was the greatest motive to complete this thesis.

1. Introduction

1.1. Problem Overview and Motivation

Circuit simulators are integral parts of most modern Computer-Aided Design (CAD) tools. The computationally dominant component in modern circuit simulators are often the intensive numerical tasks such as transient time-domain simulation, DC operating point calculation and steady-state analysis of a circuit with periodic stimulus. The main focus in these tasks is often a large matrix that needs to be factorized.

One of the main tasks that CAD tools perform on these circuits is computing the steady-state response. For circuits with linear sub-networks characterized by frequency-domain measurements, the HB approach [1], [2] is typically the method of choice for computing the steady state response, provided that the nonlinear elements have weak nonlinear behavior.

The HB approach formulates the steady-state problem as a system of nonlinear equations, in which the circuit waveforms are represented by Fourier series and the unknowns are their Fourier coefficients. Therefore, the main computational tasks are calculation and factorization of the HB Jacobian matrix at each iteration. Typically, the HB Jacobian matrix is much denser in structure than the typical matrices encountered in other circuit simulations, and also much larger in size, especially when the input contains multiple frequencies that cannot be treated as commensurate tones. However, the HB approach becomes computationally cumbersome when the nonlinear elements behave strongly in a nonlinear manner, since this produces waveforms with sharp transitions that need a large number of Fourier coefficients to be captured accurately [3].

Krylov subspace iterative methods [7] have addressed the problem of the large size of Jacobian matrix by avoiding direct factorization and even the explicit computation

of this matrix. Instead, it uses the notion of matrix-vector product and applies it iteratively until reaching convergence [8]. Nonetheless, convergence in this approach requires forming a computationally inexpensive pre-conditioner; a matrix that approximates the inverse of the HB Jacobian matrix. Unfortunately, the presence of strong nonlinear behavior makes finding a suitable pre-conditioner very difficult and the whole approach becomes ineffective with a prohibitively slow convergence rate [9]. Indeed, in those situations, as has been reported in [9], the direct factorization of the Jacobian matrix becomes more efficient.

On another independent front, developments on the front GPU with their massively parallel architecture have presented new opportunities to accelerate various parts of the circuit simulation requirements. The key advantage offered by the GPU architectures, compared with modern CPUs, is the large amounts of Giga FLoating point OPerations that can be performed in one second (Gflops). Unfortunately, translating this advantage into real performance gains is fraught with significant challenges. It is the objective of this thesis to take advantage of the GPU in the direct factorization of the HB Jacobian matrix.

One of the main challenges that impedes harvesting significant gains in performance from GPUs is the presence of dependency among the processed data items, which do arise naturally in many of the algorithms and numerical techniques used in circuit simulation. Such dependency, if not handled carefully, entails using conditional statements which degrades the performance greatly. Another bottleneck that can also reduce the the performance significantly is the frequent access to the PCIe bus to transfer data between the main CPU and the GPU board. This bus can only transfer 8 giga transfers per second in each direction [18], thereby making the communication time between the main CPU and the GPU device a dominant part of the simulation process.

While the latency arising from the PCIe bus can be hidden by careful pipe-lining approaches, the problem of data dependency presents challenges of greater magnitudes. As a result of those challenges, efficient utilization of GPUs in general circuit simulation task can, be carried out in one of two possible ways:

1. In applications where there is minimal data dependency. Examples of this approach to GPU-based circuit simulation can be found in [19], [20], where the GPU is mainly used in device model evaluation.

2. By restructuring current simulation algorithms to minimize the level of data dependencies, and make them more GPU-oriented, through ensuring more uniform memory access patterns. An example of this approach has been reported in [21] for the analysis of Power Distribution Networks using the multi-grid analysis techniques.

The work presented in this thesis falls under the second category. More precisely, the problem tackled in this work is the problem of finding the HB multi-tone steady-state solution through direct factorization of the Jacobian matrix. The objective of the proposed approach is to accelerate this factorization using a GPU. To the best of the author's knowledge, this is the first work to propose using a GPU to accelerate factorizing the HB Jacobian matrix.

1.2. Existing Solutions

This section reviews briefly the existing approaches that attempted to utilize GPU to accelerate the LU factorization of general matrices. The emergence of modern computer architecture paradigms, such as the multi-core CPU, the many core GPUs, and the Field Programmable Gate Array (FPGA) devices, and integrating them into conventional computing platforms has fueled research interest in exploiting the inherent parallelism for accelerating several tasks in circuit simulations. In particular, the problem of direct matrix factorization, which is one of the common tasks in conventional circuit simulation, has become the focus of several recent research works.

First research efforts that focused on utilizing the many-core GPU devices addressed the problem of linear system solutions for systems with dense matrices [10],[11],[12]. However, matrices arising in circuit simulation problems are characterized by very sparse structures. In addition, the dependency among the columns makes utilizing the GPU architectures very challenging. Hardware implementations of sparse LU solver are presented by [13], [14], [15] on FPGA, some of which are suitable for specific matrix types; though all are not scalable to large circuits due to FPGA on chip resource limitations (e.g. Processing Elements, Look-up Tables, Clock frequency). Handling the sparse structures (for matrices arising in circuit simulation) on FPGA was proposed in [15],[19]. However, to handle the large matrices that arise in circuit simulations,

and circumvent the limited resources on FPGAs, recent research thrust has shifted to utilizing the GPU in the matrix factorization [24]. *Li and Li* introduced a categorized sparse matrix storage format for GPU, and developed a right hand looking method for LU factorization based on that format [16]. *Chen et al.* proposed a two-fold approach for sparse systems [17], in which depending on the matrix sparsity a serial or parallel execution path is chosen. The serial path is a complete solution on the CPU, and the parallel path is based on a blocked right hand looking factorization. They also introduced a task-parallelism to improve the parallel path. Since right hand looking LU factorization methods are block based, both [16] and [17] depend on Jacobian sparsity pattern and suffer from optimum coalesced memory accesses on GPU.

It is important to stress here the different nature and structure of the Jacobian matrices arising from the HB problem, compared to the matrices encountered in regular circuit simulations, which is the main focus of the previous works in literature. Those matrices, on the one hand, inherit some of the sparse structure of underlying circuits, in the sense that coupling between the circuit nodes resembles the coupling in typical circuits. However, it is the coupling between the harmonic coefficients, used in representing the node voltages, that produces dense spots with increasing sizes for larger number of tones or larger number of Fourier coefficients. This peculiar nature presents opportunities and challenges for efficient utilization of the GPU.

1.3. Contributions

The fundamental contribution of the thesis is the demonstration of how to utilize a heterogeneous platform that includes CPU and a GPU to accelerate the direct LU factorization of the Jacobian matrix that arises in the HB analysis technique.

To achieve this contribution, the work presented in this thesis developed a block form of the LU factorization algorithm based on the well known KLU factorization algorithm [22]. This factorization technique is referred to as Hybrid-BKLU. The main advantage in the Hybrid-BKLU is that insulates the floating-point operations on contiguous blocks of memory. Such a feature becomes a key performance booster when a GPU is involved in the computations to construct the heterogeneous Hybrid-BKLU approach, since it enables the main execution kernels to have uniform memory access to totally independent data items.

1.4. Thesis Organization

The thesis is organized as follows. Chapter two presents a brief review on circuit simulations and data parallelization, and chapter three provides the necessary background theory on HB steady state analysis. Chapter four introduces the idea of the Block-Aware BKLU algorithm and shows its execution profile on a CPU platform. Chapter five presents the proposed Hybrid-BKLU approach based on GPU platform, and chapter six shows the simulation results and their descriptions. Finally, chapter seven presents concluding remarks and future work opportunities.

2. A Brief Review of Circuit Simulations and Data Parallelism

This chapter presents a brief review of the two main topics addressed by this thesis, namely, circuit simulation and the task of data parallelization. Section 2.1 provides the necessary background on the topic of circuit simulation, where it reviews the common approaches of circuit formulation and briefly discusses the various circuit simulation tasks available in commercial simulators with a particular emphasis on the task of steady state circuit simulation. Section 2.3 discusses data parallelism and its realization on hardware platforms in section 2.4. Finally, a modern programming language for implementation of parallelizable algorithms on heterogeneous platforms is described in section 2.5.

2.1. Circuit Simulation

Circuit analysis is a mathematical method used to simulate the behavior of electrical and electronic circuits (e.g. VLSI, analog/digital circuits, Radio Frequency (RF) circuits, power distribution networks) in different conditions and/or with different input excitations. In general, electric circuits are composed of simple two terminal elements, e.g. resistor, capacitor, voltage source and multi-terminal elements like transistors and transformers.

The first step common to almost all circuit simulation tasks is to describe the circuit in the mathematical domain. The circuit formulation in the mathematical domain is generally based on the Kirchoff's current and voltage laws and the constitutive equation describing the relation between the voltages and/or the currents in the different circuit elements, e.g. Ohm's law. Section 2.1.1 outlines the Modified Nodal Analysis (MNA)

[23], [39] for mathematical circuit formulation, which is the most commonly used in all commercial circuit simulators.

2.1.1. MNA Formulation of Circuit Equations

The MNA approach to representing a general circuit in the mathematical domain describes the circuit as a mixed system of differential and algebraic equations that takes the following form:

$$\mathbf{G}\mathbf{x}(t) + \mathbf{C}\frac{d\mathbf{x}(t)}{dt} + \mathbf{f}(\mathbf{x}(t)) + \frac{d}{dt}\mathbf{g}(\mathbf{x}(t)) = \mathbf{b}(t), \tag{2.1}$$

where:

- $\mathbf{x}(t)$ is a vector of the circuit unknown variables,
- $\mathbf{G}$ is a matrix describing the linear memory-less elements of the circuit, e.g. resistors, conductors,
- $\mathbf{C}$ is a matrix describing the memory elements of the circuit. e.g. capacitors,
- $\mathbf{f}$ and $\mathbf{g}$ are vectors of nonlinear functions that describe the nonlinear memory-less and memory elements, respectively, and,
- $\mathbf{b}(t)$ is a vector of independent sources.

The automatic construction of the matrices and vectors in (2.1) is carried out using the notion of "stamps", where each circuit element is associated with a specific stamp that describes its contribution to a matrix or vector in the formulation. Therefore, a particular matrix or a vector is obtained by accumulating all the stamps of all circuit elements. Figure. 2.1 shows a sample of stamps for some of the commonly used circuit elements.

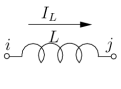
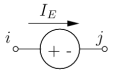
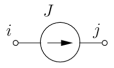
Element	Circuit Schematic	Stamp
Resistor		$\begin{array}{c} \\ i \\ j \end{array}\begin{array}{c} \begin{array}{cc} i & j \end{array} \\ \begin{bmatrix} g & -g \\ -g & g \end{bmatrix} \end{array}$
Capacitor		$\begin{array}{c} \\ i \\ j \end{array}\begin{array}{c} \begin{array}{cc} i & j \end{array} \\ \begin{bmatrix} C & -C \\ -C & C \end{bmatrix} \end{array}$
Inductor		$\begin{array}{c} \\ i \\ j \end{array}\begin{array}{c} \begin{array}{cccc} i & \cdots & j & \end{array} \\ \begin{bmatrix} 0 & \cdots & 0 & 1 \\ \vdots & \cdots & \vdots & 0 \\ 0 & \cdots & 0 & -1 \\ 1 & 0 & -1 & -sL \end{bmatrix} \end{array}$
Voltage Source		$\begin{array}{c} \\ i \\ j \end{array}\begin{array}{c} \begin{array}{cccc} i & \cdots & j & \end{array} \\ \begin{bmatrix} 0 & \cdots & 0 & 1 \\ \vdots & \cdots & \vdots & 0 \\ 0 & \cdots & 0 & -1 \\ 1 & 0 & -1 & 0 \end{bmatrix} \end{array} \begin{bmatrix} \\ \\ \\ E \end{bmatrix}$
Current Source		$\begin{array}{c} \\ i \\ j \end{array}\begin{array}{c} \\ \begin{bmatrix} J \\ \vdots \\ -J \end{bmatrix} \end{array}$

Figure 2.1.: Sample stamps

To give an example on the usage of stamps in MNA formulation, Figure. 2.2 depicts a circuit containing a nonlinear capacitor, a nonlinear resistor, an inductor and voltage and current sources. The MNA formulation for this circuit takes the following form:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & \frac{1}{R} & 1 & -1 \\ 0 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ I_E \\ I_L \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & C & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -L \end{bmatrix} \begin{bmatrix} dv_1/dt \\ dv_2/dt \\ dI_E/dt \\ dI_L/dt \end{bmatrix} + \begin{bmatrix} 0 \\ \mathbf{f}(v_2) \\ 0 \\ 0 \end{bmatrix} + \frac{d}{dt} \begin{bmatrix} g(v_1 - v_2) \\ -g(v_1 - v_2) \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} I_s \\ 0 \\ E \\ 0 \end{bmatrix} \tag{2.2}$$

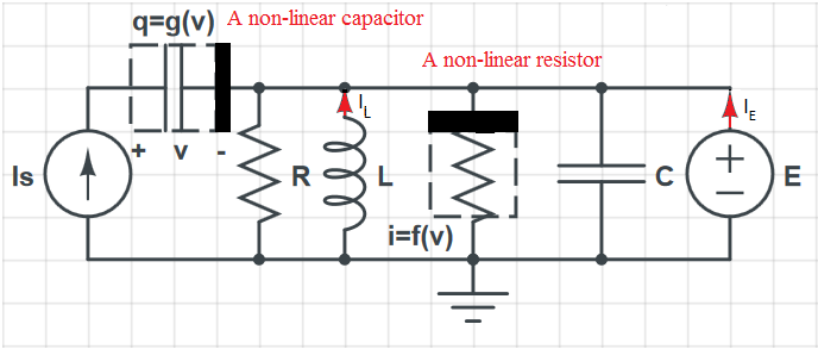


Figure 2.2.: A circuit with nonlinear capacitor and nonlinear resistor

A slightly modified version of the MNA known as the charged oriented MNA formulation, includes only algebraic nonlinear terms. In other words, this modified form of the MNA does not include the differential nonlinearity that appears as the fourth term on the left side of (2.1), thereby, leading to a system of Differential Algebraic Equations (DAE) described by :

$$\mathbf{G}\mathbf{x}(t) + \mathbf{C}\frac{d\mathbf{x}(t)}{dt} + \mathbf{f}(\mathbf{x}(t)) = \mathbf{b}(t) \tag{2.3}$$

This MNA formulation is known as the charge-oriented MNA. The representation of the nonlinear capacitors or inductors, which appears as the differential nonlinearity in the classical MNA (2.1), is done in an alternative way in the charge oriented MNA formulation by including the charge or flux of the nonlinear capacitor or inductor as an additional unknown waveform. The circuit in Figure. 2.2 could provide an example of the alternative formulation, by showing how the nonlinear capacitor is treated differently. The charge oriented formulation for this circuit produces the following system:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & \frac{1}{R} & 1 & -1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ I_E \\ I_L \\ q \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & C & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -L & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} dv_1/dt \\ dv_2/dt \\ dI_E/dt \\ dI_L/dt \\ dq/dt \end{bmatrix} + \begin{bmatrix} 0 \\ \mathbf{f}(v_2) \\ 0 \\ 0 \\ -g(v_1 - v_2) \end{bmatrix} = \begin{bmatrix} I_s \\ 0 \\ E \\ 0 \\ 0 \end{bmatrix} \tag{2.4}$$

The usage of the charge oriented MNA formulation is preferable to the classical MNA formulation (2.1) in certain situations such as the simulation of the circuit steady state periodic response to a periodic input, which is the main focus of this thesis. The preference given to this modified formulation in this situation is due to the fact that it enables using the technique of Artificial Frequency Mapping (AFM) which will be explained in details in sec. 3.2.3.

2.2. Common Circuit Simulation Tasks

This section reviews some of the frequently used circuit simulation tasks.

2.2.1. DC Analysis

Among different analysis, the DC analysis is a prerequisite for most of the other analysis and is automatically done by all computer simulation programs. DC analysis finds the operating points, i.e. the node voltages and the branch currents.

To find the operating or quiescent point of a given circuit, the circuit is left only under the influence of the DC sources. Thus all AC and transient sources in the circuit are deactivated, and all capacitors and inductors are replaced by open and short circuits respectively. This procedure typically transforms the MNA formulation into the much simpler form:

$$\mathbf{G}\mathbf{x} + \mathbf{f}(\mathbf{x}) = \mathbf{b} \tag{2.5}$$

(2.5) is an algebraic system of nonlinear equations that can be solved by iterative methods. One of the simplest iterative techniques, which is widely used in circuit simulation programs, is Newton-Raphson (NR) method. The solution obtained is a vector that defines the DC operating points for each variable in the circuit.

2.2.2. Transient Simulation

Transient analysis finds instantaneous values of the desired variables, after a power up of the input excitation sources or any sudden changes in the inputs or the environmental conditions. The initial conditions of the transient analysis is provided by the DC analysis. In the transient analysis, the derivative is approximated by one of the discretization techniques such as the Backward Euler (BE):

$$\dot{\mathbf{x}}_{n+1} = \frac{1}{h}(\mathbf{x}_{n+1} - \mathbf{x}_n), \tag{2.6}$$

where $\mathbf{x}_{n+1}$ and $\mathbf{x}_n$ are the vector of unknowns, $\mathbf{x}(t)$, calculated at $t = t_{n+1}$ and $t = t_n$ respectively, and h is given by $h = t_n - t_{n+1}$. Substituting (2.6) into the charge oriented MNA formulation transforms the differential system into following nonlinear algebraic system:

$$(\mathbf{G} + \frac{\mathbf{C}}{h})\mathbf{x}_{n+1} + \mathbf{f}(\mathbf{x}_{n+1}) = \mathbf{b}_{n+1} + \frac{\mathbf{C}}{h}\mathbf{x}_n \tag{2.7}$$

Again the system in (2.7) is solved using NR method, for $\mathbf{x}_{n+1}$ given $\mathbf{x}_n$ for values of $n = 0, 1, 2, ...$ to obtain approximate values for $\mathbf{x}(t)$ at the the discrete time instants $t_0, t_1, t_2, ...$ where $\mathbf{x}_0$ is obtained from the DC operating point analysis. As an example, Figure. 2.3 shows transient response of a sample diode circuit after power up in the time domain.

2.2.3. AC Analysis

AC analysis calculates the output response of a circuit to a small-signal sinusoidal input in the steady-state condition. This type of simulation assumes that response contains only the same frequency of the input signal. Therefore, the circuit should contain linear elements or be linearized using proper models in advance. The linearized MNA equations are transformed to Laplace-domain to perform AC analysis in the frequency domain. The resulting system is represented by:

$$\mathbf{G}\mathbf{X}(s) + s\mathbf{C}\mathbf{X}(s) = \mathbf{B} \tag{2.8}$$

The circuit is analyzed over a range of frequencies by substituting s with $2\pi jf$ where f is swept over the desired frequencies in Hz. The system (2.8) being linear is solved by simply factorizing the matrix $\mathbf{G} + \mathbf{C} + 2\pi jf$ for each value of f in the desired frequency range.

2.2.4. Steady State Analysis

This type of simulation is the main focus of the work done in this thesis. Steady state analysis, in this thesis, refers to to the task of simulating the periodic circuit steady-state response as a result of applying or stimulating the circuit with periodic sources, i.e. sources that varies periodically with time. In general, the response of a circuit includes a transient phase that starts at $t = 0$, and a steady state response that is reached as t approaches to ∞. The goal of the steady state analysis is to compute the steady state phase of the response without having to follow the circuit waveforms right from the beginning at $t = 0$. Figure. 2.3 illustrates the main idea of the steady state analysis by showing the response of a diode circuit stimulated by a periodic input source, where the figure highlights the initial transient phase that evolves, as $t \to \infty$, into a periodic response. The goal of steady state analysis is to compute the steady state phase directly.

Steady state is a stable and balanced situation of a circuit that begins after the transient behavior of the circuit is passed. Steady state analysis methods are classified into time domain (e.g. shooting method) and frequency domain (e.g. harmonic balance). Time domain methods solve differential system of equations using ordinary differential equation methods.

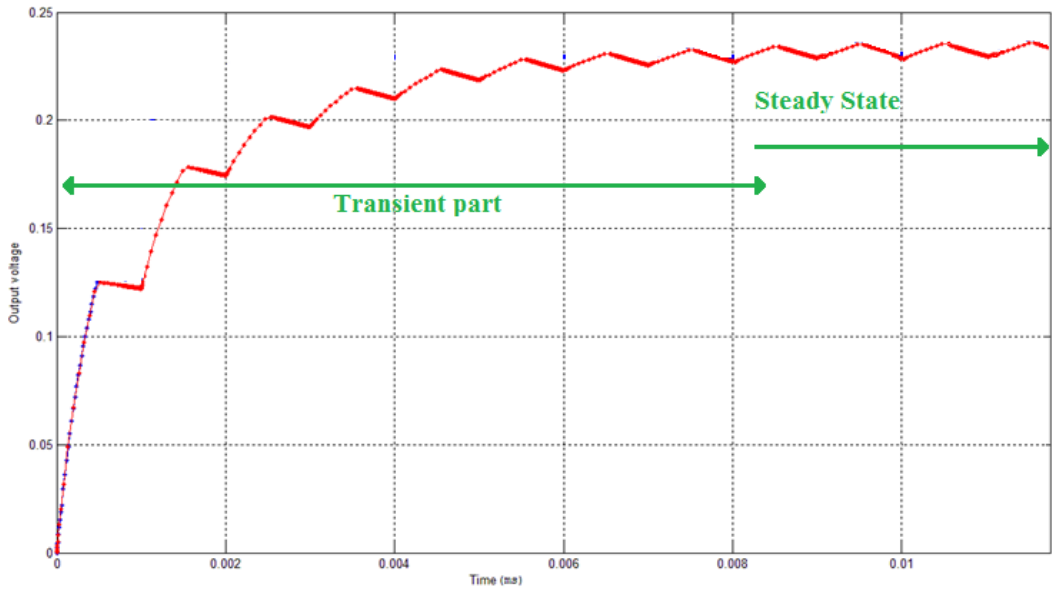


Figure 2.3.: The transient and steady-state response

The main thrust of this thesis is on the steady state analysis using the HB technique. In particular, the HB approach is an integral part of the modern CAD tools of RF and analog microwave circuits which are common to all front-end wireless communication systems.

For circuits with linear sub-networks characterized by frequency-domain measurements, the HB approach [1],[2] is typically the method of choice for computing the steady state, provided that the nonlinear elements have weak nonlinear behavior. The HB approach formulates the steady-state problem as a system of nonlinear equations, in which the circuit waveforms are represented by Fourier series, and the unknowns are its Fourier coefficients. The main computational task therefore becomes the computation and factorization of the HB Jacobian matrix at each iteration. Typically the Jacobian matrix is much denser in structure than the circuit-like matrices and also much larger in size, especially when the input contains multiple frequencies that cannot be treated as commensurate tones. However, the HB approach becomes computationally cumbersome when the nonlinear elements behave strongly in a nonlinear manner, since this produces waveforms with sharp transitions that need a large number of Fourier coefficients to be captured accurately [3]-[7].

The goal of this thesis is to employ a GPU to speedup the direct factorization of the Jacobian matrix. Chapter 3 presents a detailed derivation of the problem that arises from the HB technique.

2.3. Data Parallelism

The goal of data parallelism is to decrease the execution time of a program or algorithm using parallel processing elements. The processing element could be a single processor, a core in a multi-core CPU or in a many-core GPU. In contrast to task parallelism, in which the execution of different processes are distributed between computing devices, data parallelism is a type of parallelization of algorithms where data are distributed between computing devices that process them by the same task. In the next sections, a review of speedup in data parallelism, methods, models and architecture required to handle it are presented.

2.3.1. Speedup

Amdahl's law [25] defines the available speedup for fixed size work by dividing it via processing elements. There is always a part of a work, declared as a percentage a, that cannot be parallelized and limits the speedup. If T_1 is the required time for serial execution of the work and T_n is the execution time of the work on n threads executing in parallel, $S(n)$ is the achieved speedup using n execution threads relative to serial execution:

$$S(n) = \frac{T_1}{T_n} = \frac{1}{a + \frac{1-a}{n}} \tag{2.9}$$
$$a \in [0, 1]$$
$$n \in \mathbb{N}$$

On the other hand, if the work size is increased in a fixed time, the Gustafson's law [26] would be beneficial. If t_s and t_p are the execution time of the serial and parallel part of the work respectively, $t_s + t_p$ is the execution time on a parallel computer with n processing elements and $t_s + nt_p$ is the execution time on a single processing element. Thus, $S(n)$ the achieved speedup using n processing elements is:

$$S(n) = \frac{T_1}{T_n} = \frac{t_s + nt_p}{t_s + t_p} = n - \frac{t_s}{t_s + t_p}(n - 1) \tag{2.10}$$

A good example of incorporating the Gustafson's law is processing stream of packets in a network.

Based on the classification for computer architectures defined by Flynn [27], four groups are characterized by the number of concurrent instructions (or programs) and data streams Table. 2.1.

Data parallelism or loop level parallelism is the method to achieve speedup by simultaneous operation on large group or stream of data on parallel processing elements, rather than dividing the operation between parallel tasks or threads. In fact, in data parallelism the processing elements do the same task on different parts of data, while

Table 2.1.: Flynn's taxonomy

	Single Instruction	**Multiple Instruction**
Single Data Stream	SISD	MISD
Multiple Data Stream	SIMD	MIMD

in task parallelism the processing elements do different tasks on different data. The hardware implementation of data parallelism is done by Single Instruction Multiple Data (SIMD) architecture. Three main systems which use SIMD architecture are: Vector Processors, Multimedia Processors and GPUs. It is worth mentioning that GPUs also benefit from another architecture which is a subcategory of MIMD called Single Program Multiple Data (SPMD). In SPMD devices, multiple processors can independently execute a copy of a single program. Vector processors and multimedia processors are explained in the following two subsections, and GPU's basics and architecture are explained in more details in the next section.

2.3.2. Vector Processing

The first applications of SIMD architecture in vector processing were in CDC STAR-100 [29] and TI-ASC [28] supercomputers in early 1970s, and it was popularized by Cray supercomputers in the later decade. In a vector processor, large amount of data are read from memory to register files, then a single instruction operation is executed on these registers and finally the results are written back to the memory.

Memory access is a great issue in vector processing. In order to benefit from parallel data processing, data vectors must be supplied as fast as possible. Therefore, the vector processor must access multiple memory banks instead of sequential or interleaved access. For instance, Cray T932 needs 1344 memory banks to operate on full memory bandwidth [30].

2.3.3. Multimedia SIMD

Today, SIMD module and instruction set architecture are widely used in mobile devices and almost all desktop computers. There are many applications that operate on smaller data sizes than 32 bit, e.g. graphics pixels use 8 bits for each color and

audio samples are 8 bits or 16 bits. Also, signal processing algorithms in multimedia applications usually operate on blocks of data. Therefore, utilization and performance of such systems could be improved by using small SIMD modules. In SIMD module packing 16 data bytes or 8 data words needs a 128 bit vector register. This simplifies the overall architecture of SIMD module because it requires less logic circuits and area for implementing ports, crossbars and register banks. As a result, the instruction set could be increased and designed to be more powerful than conventional CPU instruction set [30]. Although streaming SIMD extensions (SSE) instruction set is improved by performing double precision floating point operations, it is almost impossible for a compiler to benefit from the complete power of SSE instruction set to automatically generate optimal code for an arbitrary multimedia application. Instead, it is more useful to provide optimized hand-written libraries, e.g. a Finite Impulse Response (FIR) filter, to be used in multimedia applications such as a speech compression algorithm.

2.3.4. Stream Processing Model

Stream processing model is based on arranging a program or algorithm into groups of small computational kernels that operate on streams of data. Let us define:

- Streams are sets of data of the same type, which may be single numbers or complex structures.
- Kernels are sets of operations, simple or complex computations that perform on input stream elements and provide at least one output stream.

Considering the structure of this model for communication and concurrency between kernels and streams, efficient use of memory and processing resources is provided by steam processing model [31]. First, for communication, local access is used for transferring temporary results within each kernel, streaming is used for transferring data streams between kernels, and global movement is used for I/O devices. Second, concurrency is realized through instruction level parallelism on a part of a kernel, data parallelism by simultaneously applying the same group of operations on different set of elements of streams, and kernel parallelism via executing multiple independent tasks in parallel. Stream processors work on streams and kernels, and are specially designed according to the stream processing model, which means that they are efficient candidates for applications with large amount of parallel computations, coarse granularity

memory usage and clearly defined control structures [31].

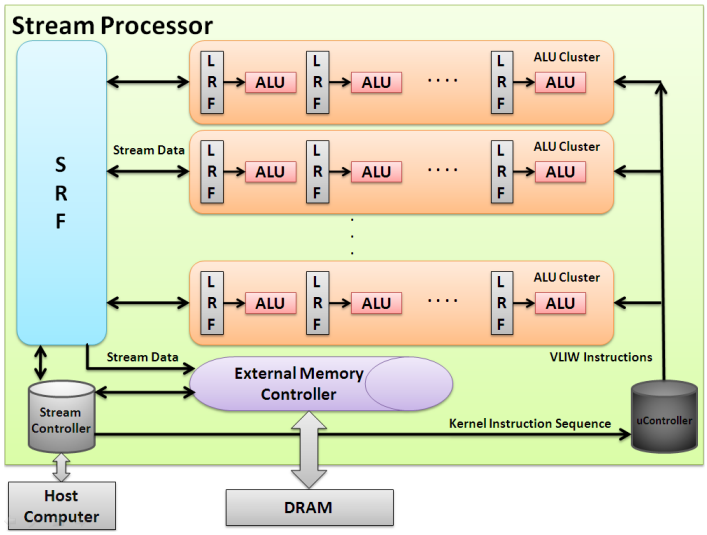


Figure 2.4.: Stream processor architecture [31]

The most important blocks of a stream processor, as shown in Figure. 2.4, are:

- ALUs: ALU is the smallest and the main functional-computational unit. ALU clusters consist of the same type and number of ALUs.
- LRFs: LRF is a local register file used for fast and temporary data access by each computational unit (ALU).
- SRF: SRF is the stream register file used to transfer streams of data between LRFs and/or off chip host or memory.
- DRAM: Dynamic RAM is the off-chip memory used for storing mass global streams of data.

The kernel instructions and stream transfers are sent by the host processor to the stream controller. Also, the stream controller is responsible for communicating with the host processor. The uController receives SIMD instructions from the stream controller and sends them to ALU clusters. If the compiler has the instruction scheduling capability, it would be able to incorporate Very Large Instruction Word (VLIW) instructions. Finally, the external memory controller module handles stream transfer between the external memory and the internal SRF.

2.4. Graphics Processing Unit, GPU

Over the past ten years, there have been important improvements in graphic processing hardware and software. The demands for better resolution, more colors and higher image resolution and video quality provide daily increase in performance and capabilities of GPUs. While GPU was designed for a special purpose, its processing power and memory bandwidth made it an ideal platform for solving complex algorithms. By discovering the capabilities of Graphic Processors to solve complicated mathematical problems and parallel execution of thousands of threads, GPU became an interesting area to research. Rapid growth of the game market, the performance improvements of GPU chips and their programmability during the last decade boosted research activities in this field, and revealed more potential applications for GPU programming.

In the following subsections, first a brief review of General Purpose computing on GPU (GPGPU), which was the motive for new GPU architecture design, is presented and then architecture of the last two modern GPUs, VLIW and Graphics Core Next (GCN), from AMD Company are described.

2.4.1. GPGPU

One of the most important reasons of improving graphic cards was to realize Three Dimensional (3D) virtual scenes into Two Dimensional (2D) images. Rendering an image is performed by doing various processes on four graphic basic elements, i.e. vertices, primitives, fragments, and pixels. This could be done by passing data through several mathematical computation stages called Graphics Pipeline. At the beginning, these stages were implemented by fixed-hardware designs. The emergence of new graphic algorithms transformed some of these stages to be programmable a few years afterward (early 2000) [35]. The final evolution pace toward modern pipeline was done by ATI Company in 2005. They introduced a unified shader (also called unified pipeline) in the Xenos chip for Xbox360 game console. Unified shader incorporates a large block of parallel floating processors for all shader stages, instead of serializing several shading stages. Before introducing unified shader, GPU resources were underutilized. For example, an application or part of an application may require a large amount of vertex shading and other application or part may require a little pixel

shading. A unified shader can dynamically allocate variable amount of its parallel processors for each shading algorithm or stage. Figure. 2.5 shows a simple flow diagram of old and new pipelines. Input and output buffers are considered to provide pipelining stages as well as parallel execution of shaders. For example in this figure, each of *Shaders* $A - D$ could be under-utilized.

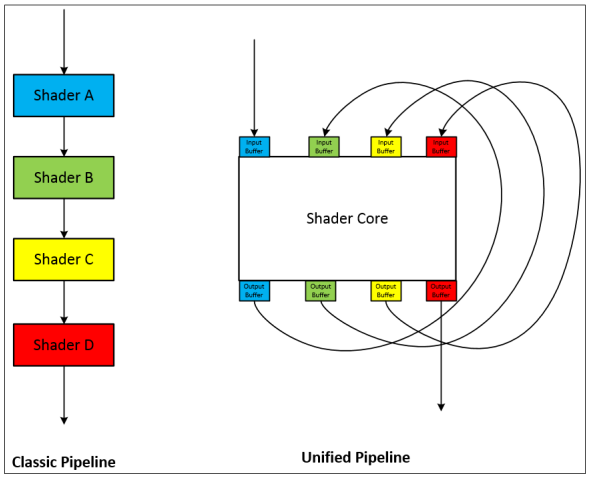


Figure 2.5.: Classic and new Graphic Pipeline [36]

After the emergence of unified pipeline, several researchers tried to speedup their non-graphics algorithm using programmable shading stages. At this stage, a programmer should introduce data as graphics elements, and adopt the algorithm like a graphics processing shader algorithm. Although this method of programming was quiet tedious task, it eventually became successful and started a new era in general purpose computing on graphics processing cards. Popularity of GPGPU and future potentials in parallel processing motivated GPU manufacturers to adapt GPU architecture based on general computation as well as graphics processing. Today the two most favorite GPGPU programming languages are NVIDIA's CUDA and OpenCL standard. Since CUDA is a proprietary language and works only on NVIDIA GPUs, OpenCL which is a platform independent open standard language is used as the programming language in this thesis and is explained in more detail in the next section.

2.4.2. VLIW Architecture GPUs

GPU is not a stream processor, in other words a GPU is not suitable for using in a video camera. GPUs are designed to render 3D images. They should work on large amounts of data with simpler computations than stream processors. Therefore, the stream processor architecture had been considered for GPU architecture design.

There are three important definitions which are closely related to hardware and need to be introduced now.

1. Work-item: each kernel instance is a work-item (also called thread).
2. Work-group: work-items are organized into clusters called work-groups. Within a work-group, work-items can share data in local memory and all work-items within a group execute on the same stream processor array.
3. Wave-front: a wavefront is a group of concurrently executing work-items on the same stream processor array.

In order to provide a reference for further comparison, first the main modules of AMD HD2900 GPU (also known as R600 family) [37], which was the first desktop computer GPU with unified pipeline, is explained. Figure. 2.6 depicts the architecture of AMD HD2900. The main modules of AMD HD2900 are:

- Command processor: It is the first module seen from host in the GPU, and is responsible for command and data stream fetching and state control. It also includes separate queues for commands, data streams and kernels, DMA for data streams and interrupt control unit.
- Ultra Dispatch Processor (UDP): The main control unit for the compute core is UDP. It distributes workloads into 64 work-items and could handle up to 100 work-items in flight to hide latency. It can inhibit work-items that are waiting for resources.
- Stream Processing Unit (SPU): Each SPU contains 5 Processing Elements (PE, also called stream processor) that works on 4 separate data elements performing 5 scalar instructions during 4 cycles. It is arranged as a 5 way processor capable of executing 5 scalar floating point multiply-add (or 5 integer operations), Figure. 2.7.

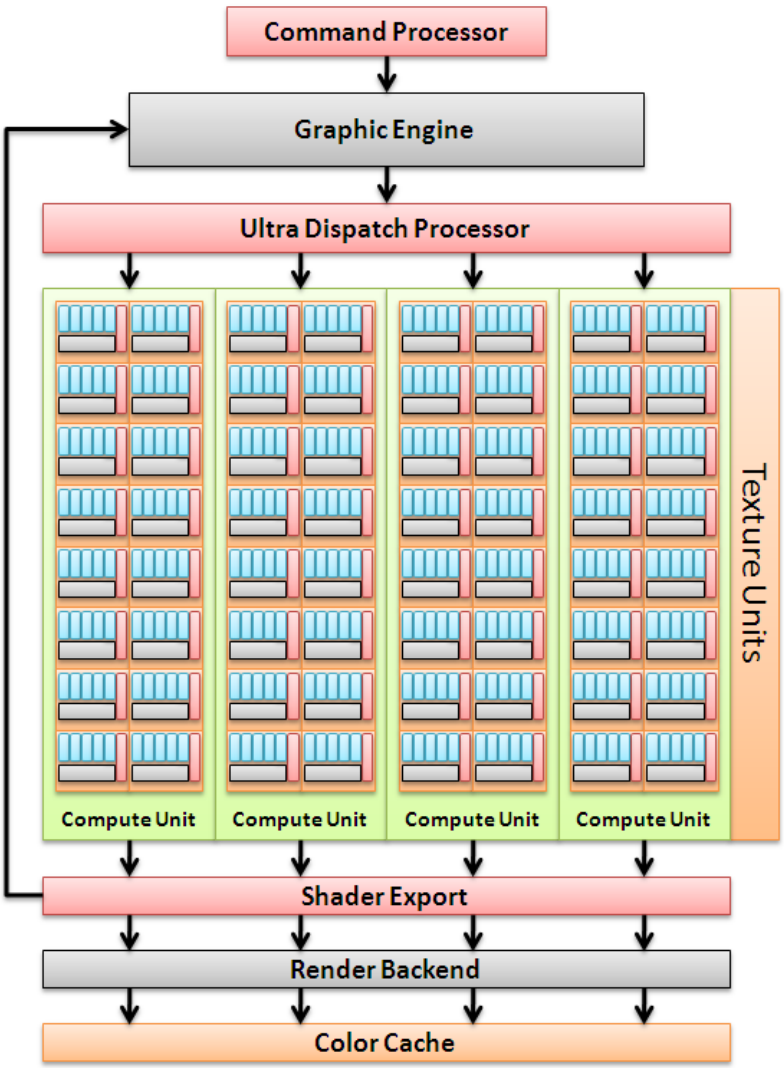


Figure 2.6.: Simplified AMD HD2900 GPU architecture [37]

- Stream Engine: Stream engine or Compute Unit (CU), includes 4 SIMD arrays of each containing 16 SPUs, totally 320 processing elements.
- Branch Execution Unit: A separate branch execution unit is considered to directly handle the flow control and branch predication in PEs.
- General Purpose Registers: GPRs are multi-ported shared register files for each SPU.

The previous 5 way SPU (VLIW5) design has been chosen by AMD to cover their

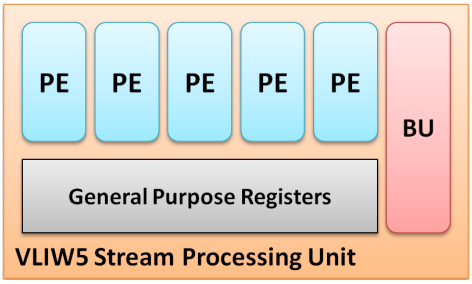


Figure 2.7.: Stream Processing Unit [37]

vertex shader algorithm which used 4 floating point components dot product (w, x, y, z) and a scalar for lighting at the same time. Recently, researchers in AMD found that new algorithms seldom use the fifth processor and it is underutilized. Therefore, they changed the SPU to VLIW4 architecture with equal functionality for processors, and used the space of the fifth processor to increase the performance and number of SPUs. As a result, AMD HD6900 (also known as Cayman family) has 24 SIMD engines (totally 1536 processing elements) and supports 64 bit floating point operations as well [38], Figure. 2.8.

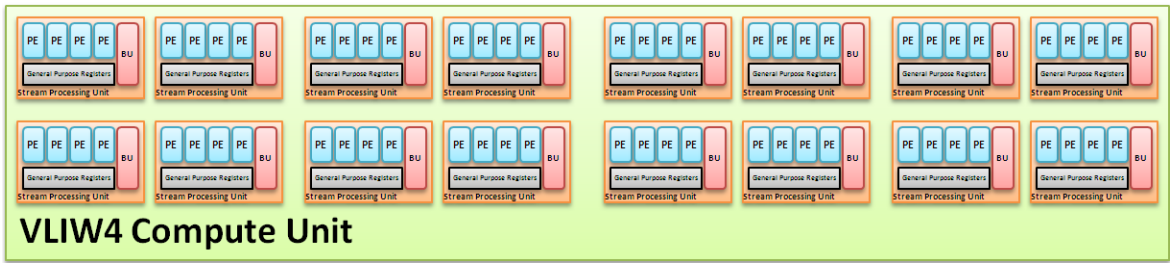


Figure 2.8.: VLIW4 Compute Unit architecture

2.4.3. GCN Architecture GPUs

The most important difference between modern GPUs and previous architectures is in their SIMD engines, which is explained here.

The compute unit or shader engine in VLIW architecture included 16 stream processors which could execute 4 or 5 instructions bundled in a VLIW5 instruction. The compiler was responsible to find and schedule those instructions. Since VLIW architecture was essentially designed to support operations on basic graphic elements properties; x,y,z,w

or colors of a dot (red, green, blue); the code should be vectorized in order to obtain the maximum utilization of GPU processing power. This architecture was well suited for graphics data and algorithms. However, the majority of computation algorithms in general purpose GPU programming could not be vectorized, and also the compiler was unable to provide the optimum parallel code. Therefore, GCN architecture has been designed to support both graphic and general purpose computations.

In this architecture, each CU has four 16 way vector SIMD units, and each SIMD unit can execute instructions with up to 16 data elements. In other words, each SIMD unit can concurrently execute a single instruction in 16 work-items, or a vector instruction in 8 or 4 work-items, and compute a separate wave-front. Thus, the responsibility of finding wave-fronts to be executed in parallel is transferred to a dedicated hardware scheduler [34]. Figure. 2.9 shows a compute unit in GCN architecture. As can be seen, this compute unit contains four stream processing unit (SIMD) each containing 16 processing elements and private general purpose registers. Although there is only one BU, it contains four separate program counters and instruction fetch units for each stream processing unit. The incorporated GPU platform in this thesis is the AMD Radeon ™ HD7950 (also called Tahiti family), which is based on the GCN architecture.

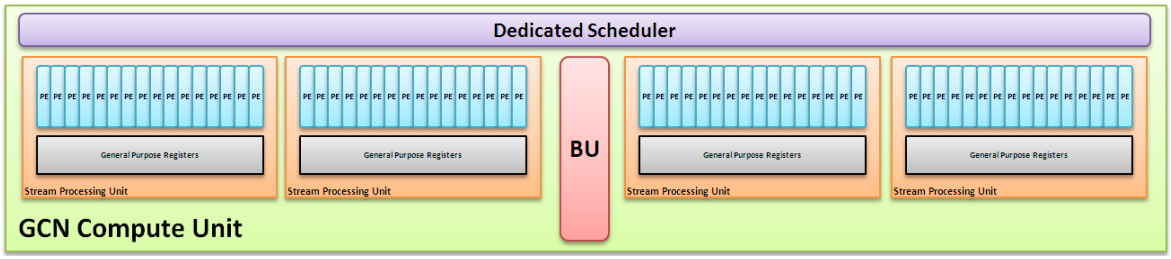


Figure 2.9.: Simplified block diagram of a compute unit in GCN architecture

2.5. Open Computing Language, OpenCL

OpenCL is a platform-independent open standard and royalty free language which is developed to exploit remarkable parallelism power in multi-computing device systems. These systems may include multi-core central processing units (CPU), and or many-core GPU, and/or digital signal processors (DSP), and/or reconfigurable hardware such

as FPGAs. The run-time compilation characteristic, as one of the main characteristics of OpenCL, enables the programmers to recognize available devices, select optimum devices for tasks, assign tasks to devices and control their execution and transfer data between them. OpenCL is a super-set of the C99 standard with built in functions and special types for data parallelism, like vector data types and image data types, which supports SIMD instructions even on x86 multi-cores. While OpenCL language concepts and keywords are quite simple to understand, the important challenges are first thinking parallel and then designing and writing portable parallel programs which perform well on a variety of different devices.

2.5.1. Platform Model

In the OpenCL platform model, a host device is connected to one or more compute devices, Figure. 2.10. Each compute device may has several compute units which in turn are composed of many processing elements. The host could be a computer running a well known operating system (e.g. Windows, Linux) on a single or multi-core CPU with a common bus interface like PCIe which connects all devices and main CPU. Devices can execute program instructions which could be SIMD. The common bus provides communication between the host and devices as well as sharing memory between them. The host controls program execution on devices and data transfer between device memories.

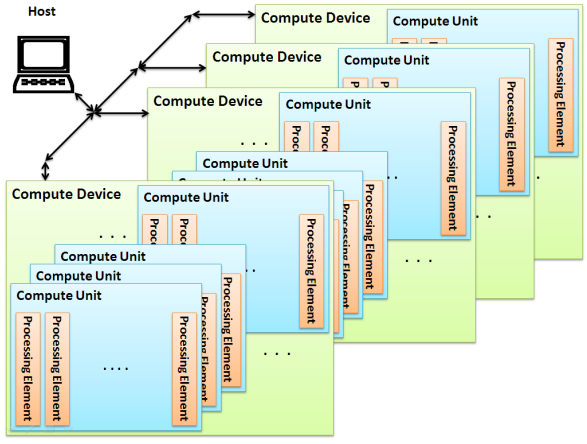


Figure 2.10.: OpenCL platform model [32]

2.5.2. Execution Model

Any OpenCL application consists of two main parts: the host program which runs on the host and the device programs which run on the associated devices. The device programs are made of simple functions named kernels. Kernel instructions are executed in the processing elements of the compute units. In order to manage the OpenCL devices, the host program creates an environment named context, which contains the desired devices including the host CPU. Context is the main object which creates and manages the other objects like command queues, memory objects, and kernel objects. Figure. 2.11 shows a simple context containing a CPU, a GPU, a DSP and three command queues.

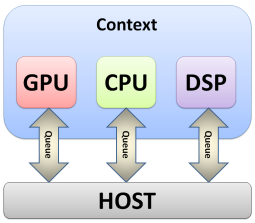


Figure 2.11.: A Simple Context [32]

OpenCL supports both data-parallelism and task-parallelism. Either the developer or the OpenCL itself can define the total number of work-items executing in parallel, and partition them into work-groups. A problem should be defined in an N-dimensional work space named NDRange, in which N is 1, 2, and 3. For example a 2D work space is used to process surface problems. Each independent element of the work space is called a work-item. During the execution, a kernel is instantiated and assigned as a work-item with appropriate work space indices. All of the work-items execute the same kernel instructions, but on different data. The work-items could be grouped into work-groups. In order to identify a work-item in a defined work-space, two types of indices are considered, Global ID and Local ID. Figure. 2.12 depicts a 16×16 image, totally 256 pixels, consisting of 16 work-groups each containing 4×4 work-items. A specific work-item's IDs and its work-group IDs are shown as well.

Figure 2.12.: Work-groups and Work-items [33]

2.5.3. Memory Model

The OpenCL memory model is composed of four memory areas, Figure. 2.13. Global memory is the largest memory (>1GB) which all work-items and work-groups can read from and write into. Furthermore, the host has access to the global memory and is the only device that can allocate this memory. A specific portion of the global memory is considered as constant memory that work-items have read-only access to it. Local memory is a relatively small memory (<256KB) which is used for shared read and write accesses of the work-items in each work-group. The local memory is several times faster than the global memory. Private memory is another small memory (<256KB) which is the closest to the work-items and is several times faster than the local memory. The private memory structure is based on register file architecture and is only accessible to one work-item. In OpenCL, memory management is explicit which means data must be transferred in sequence from the host memory to the global memory and to the local memory or back.

2.5.4. Compilation Model

Whilst the OpenCL programming language is based on the ISO C99 specification with some specific extensions and restrictions, OpenCL uses dynamic or runtime compilation for device kernels. In dynamic compilation, also known as off-line, the code is first complied to an Intermediate Representation (IR), and later during runtime the IR is compiled to machine code for execution. The IR step is usually done once, hence the next step is shorter. In the runtime compilation, also known as on-line, all kernels code (C text) are compiled during each execution time. Not all manufacturers support both of them.

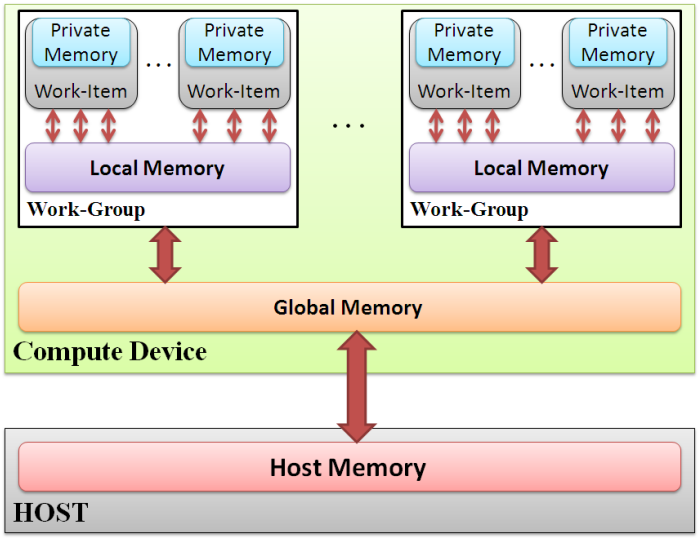


Figure 2.13.: OpenCL Memory Model [32]

An OpenCL developer must use both platform Application Programming Interface (API) and runtime API. Using platform API, developer should query for available OpenCL devices and create contexts to use them. And with runtime API, developer uses context to manage the execution of kernels on OpenCL devices and data transfer between devices and the host.

2.5.4.1. Extensions

There are new extensions to ISO C99 in OpenCL. The important additions which provide support of parallelism are:

- Work-items and work-group concepts.
- Specific vector and image data types (e.g. float4, char16, image2d).
- Synchronization commands and atomic operations (e.g. barrier).

In addition, there are address space qualifiers for Global, Local, and Private memory variables. And, built in functions such as vector mathematical functions, image access functions, and work-item/work-group identification functions.

2.5.4.2. Restrictions

Programmers should be well acquainted with the OpenCL and GPU restrictions. The OpenCL does not support any of the standard C99 headers, function pointers, function return values, recursion, variable length arrays, multidimensional arrays, pointer to pointer, pointer space conversion and bit field structures. Although manufacturers may claim to support OpenCL, there are always some features, usually not declared clearly, that are not considered. For instance, sub-device division (also called device fission) has not yet been supported on GPUs.

2.5.5. Parallel Design Considerations

In order to gain GPU's compute power more efficiently and perform more utilization of the GPU's resources, three important points should be considered during the design and implementation stages, partitioning the work, data locality and path divergence which are discussed in next subsections.

2.5.5.1. Partitioning The Work

According to GPU manufacturer recommendations [41], one of the most effective methods in exploiting GPU's compute power is to define maximum number of work-items and work-groups to keep all compute units busy. First, this is achieved by specifying larger NDRange dimension for bigger problems. Second, the optimum amount of the work for each work-item should be defined according to the available private memory and local memory. Third, the optimum number of local work-items, which results in concealing data latency, should be arranged by considering the number of stream processors and the number of wave-fronts in the fly. Unfortunately, there is no specific formula to accomplish this, and some experiments are required to obtain the best results for a specific GPU device.

2.5.5.2. Data Locality

Accessing the same memory address or adjacent memory addresses by a small portion of code for a short time is called data locality. It is desirable to provide data as

closest as possible to the executing cores. In practice, the following important GPU memory specification, which directly affects the application performance, should be considered: the cache size, the cache line width, the number of memory channels, memory channel bus width and memory allocation granularity size. The speed of access to different memory levels in GPU from private to local and global degrades by an order of magnitude. Therefore, besides the points that are declared in GPU manufacturer's documents for arranging data in the memory, the following two most important rules are considered. First, if the same data or array of data is used by a group of work-items, it should be moved by only one work-item to the local memory in order to avoid multiple access and memory channel conflict. Second, if the elements of an array are accessed by work-items in a work-group, they should be synchronized by a barrier to provide a single wide memory access.

In conclusion, to increase the performance of memory transfer data must be accessed sequentially and as close as possible to CUs. Comparing to CPUs, GPUs are weak in pointer operations, which means that there is huge performance degradation in random memory access. As a consequence, algorithms that perform too much traversing through pointer chain operations are prone to achieve poor performance.

2.5.5.3. Path Divergence

Flow control execution such as branches, switch cases and loops in GPU is quite different than in CPU, which burdens some unnecessary execution of instructions to the GPU. In fact, the stream processor hardware combines all different paths in the kernel's code and executes them sequentially. This process is called branch divergence. For example, in a kernel with this condition:

$$\begin{cases} if\ (condition) \\ \quad statement1 \\ else \\ \quad statement2 \end{cases}$$

all work-items first execute *statement*1 path and then execute *statement*2 path. The total execution time is the sum of the execution times of both paths, but only the results of the necessary path is considered. Therefore, it is desired to avoid flow control in the kernels code as much as possible.

2.6. Summary and Discussion

The OpenCL parallel computing language is used in this thesis to harvest potential data parallelism in the problem of HB in steady state circuit simulation. This is done by reconstruction of new data structure for the Jacobian matrix in HB analysis and properly allocation of memory and compute power of the CPU-GPU heterogeneous platform. Next chapter will discuss the HB analysis in detail and will be continued by first the implementation of the problem on CPU.

3. The Harmonic Balance (HB) Analysis

This chapter provides a detailed derivation of the mathematical problem that arises in the course of the HB analysis of nonlinear circuits in the steady state regime. The chapter serves to explain the computational problem at the heart of the HB approach.

Section 3.1 derives the HB analysis for circuits with single tone excitation sources. Section 3.2 extends the HB to circuits with multi-tone excitation sources using the artificial Frequency Mapping Technique (FMT).

3.1. Single Tone Excitation

A signal $u(t)$ is considered a single tone if it can be represented using a Fourier series as follows:

$$u(t_i) = U_0 + \sum_{k=1}^{K} U_k^C cos(k\omega_0 t_i) + U_k^S sin(k\omega_0 t_i), \tag{3.1}$$

where ω_0 is the angular frequency in rad/sec which is given by:

$$\omega_0 = \frac{1}{2\pi f} \tag{3.2}$$

and f is the single tone or frequency in Hz. $u(t)$ is a periodic signal, whose period T

is given by:

$$T = \frac{1}{f} \qquad sec \tag{3.3}$$

At the center of Fourier representation (3.1) is the Fourier coefficients U_0, U_k^C and U_k^S. Computing those coefficients is an essential part of the HB analysis, and is treated in the next subsection.

3.1.1. The Discrete Fourier Transform (DFT)

To determine the coefficients of Fourier expression, $u(t)$ is sampled at equally spaced $H = 2K + 1$ time instants $t_0, t_1, ..., t_{2K}$ in the interval $[0, T]$:

$$\underbrace{\begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{2K} \end{bmatrix}}_{u} = \underbrace{\begin{bmatrix} 1 & cos(\omega_0 t_0) & sin(\omega_0 t_0) & \cdots & cos(K\omega_0 t_0) & sin(K\omega_0 t_0) \\ 1 & cos(\omega_0 t_1) & sin(\omega_0 t_1) & \cdots & cos(K\omega_0 t_1) & sin(K\omega_0 t_1) \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & cos(\omega_0 t_{2K}) & sin(\omega_0 t_{2K}) & \cdots & cos(K\omega_0 t_{2K}) & sin(K\omega_0 t_{2K}) \end{bmatrix}}_{\mathbf{\Gamma^{-1}}} \underbrace{\begin{bmatrix} U_0 \\ U_1^C \\ U_1^S \\ \vdots \\ U_K^C \\ U_K^S \end{bmatrix}}_{U} \tag{3.4}$$

The Inverse Discrete Fourier Transform (IDFT) operator Γ^{-1} matrix can be used to obtain the time domain samples of $u(t)$ from the Fourier series coefficients. Similarly, the inverse of the IDFT, or DFT matrix Γ, can be used as an operator to obtain the Fourier series coefficients given the time domain samples. Both Γ^{-1} and Γ are needed to convert between the frequency and time-domain. Some of the properties of the IDFT operator are important to note. The columns of Γ^{-1} are orthogonal with the norm $\frac{H}{2}$, except for the first column that has a norm of H. Therefore, considering two

arbitrary columns γ_m and γ_n in Γ^{-1}, their inner products is given by:

$$\gamma_m^T \gamma_n = \begin{cases} 0 & m \neq n \\ \frac{H}{2} & m = n \neq 1 \\ H & m = n = 1 \end{cases} \tag{3.5}$$

Since the product of $(\Gamma^{-1})^T \Gamma^{-1}$ is a diagonal matrix, after some calculation it can be shown that rows of the Γ, ρ_m, are calculated from columns of Γ^{-1} as following:

$$\rho_m = \begin{cases} \frac{1}{H}\gamma_m^T & m = 1 \\ \frac{2}{H}\gamma_m^T & m = 2, 3, ..., H \end{cases} \tag{3.6}$$

The DFT and IDFT matrix operators are the key elements in constructing the Jacobian matrix, which is needed to solve the nonlinear system of equations.

3.1.2. The HB for Linear Circuits

To facilitate deriving the HB for general nonlinear circuits, first its implementation to the simpler type of linear circuits is presented. According to the MNA formulation discussed in previous chapter, a general circuit with only linear components can be described by the following system of equations:

$$\mathbf{G}\mathbf{x}(t) + \mathbf{C}\frac{d\mathbf{x}(t)}{dt} = \mathbf{b}(t), \tag{3.7}$$

where, $\mathbf{b}(t)$ is a vector whose entries are periodic function of the time with period T, and $\mathbf{x}(t)$ is the vector of the circuit response waveforms, that is also periodic with the same period. Using the idea that periodic waveforms can be represented by Fourier series, both $\mathbf{b}(t)$ and $\mathbf{x}(t)$ can be substituted in (3.7) to obtain:

$$\mathbf{G}\left(X_0 + \sum_{k=1}^{K} X_k^C \cos(k\omega_0 t) + X_k^S \sin(k\omega_0 t)\right) + \mathbf{C}\left(\sum_{k=1}^{K} -k\omega_0 X_k^C \sin(k\omega_0 t) + k\omega_0 X_k^S \cos(k\omega_0 t)\right) = B_0 + \sum_{k=1}^{K} B_k^C \cos(k\omega_0 t) + B_k^S \sin(k\omega_0 t) \tag{3.8}$$

Here B_0 and $B_k^{C,S}$ are the Fourier coefficients of $\mathbf{b}(t)$ which are known since they represent the independent sources, while X_0, $X_k^{C,S}$ represent the unknown Fourier coefficients of the circuit response. The goal of the HB analysis is to compute those coefficients . This goal is achieved by taking advantage of the orthonormal property of the *sin* and *cos* functions. Multiplying the above formula once by $cos(p\omega_0 t)$ and once by $sin(p\omega_0 t)$, p being an arbitrary integer, and integrating from 0 to T yields to the following two systems of equations:

$$\mathbf{G}X_p^C + p\omega_0 \mathbf{C}X_p^S = B_p^C \tag{3.9}$$

$$\mathbf{G}X_p^S - p\omega_0 \mathbf{C}X_p^C = B_p^S \tag{3.10}$$

Also, integrating (3.8) from 0 to T leads to (3.12) and provides a system for the DC coefficients vector X_0,

$$\begin{bmatrix} \mathbf{G} & p\omega_0\mathbf{C} \\ -p\omega_0\mathbf{C} & \mathbf{G} \end{bmatrix} \begin{bmatrix} X_p^C \\ X_p^S \end{bmatrix} = \begin{bmatrix} B_p^C \\ B_p^S \end{bmatrix} \tag{3.11}$$

$$\mathbf{G}X_0 = B_0 \tag{3.12}$$

Solving for the coefficients can be carried out by first solving (3.12) for X_0. Next, (3.11) is solved for X_p^C and X_p^S for all possible values of $p = 0, 1, 2, ..., k$.

3.1.3. The HB for Nonlinear Circuits

The steady-state response of a circuit with a nonlinear element will contain harmonic components at multiplies of the excitation frequency. For example, $i_s(t)$ in the circuit of Figure. 3.1, contains harmonic components up to order 3 of input angular frequency ω, is calculated as: $i_s(t) = 8 + 17cos(wt) + 8cos(2wt) + 3cos(3wt)$.

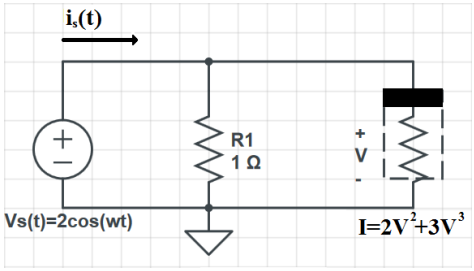


Figure 3.1.: A nonlinear circuit with cubic nonlinearity

In a general circuit with complex nonlinear behavior, the response will have infinite multiples of the excitation frequency. Thus $\mathbf{x}(t)$ is written as:

$$\mathbf{x}(t) = X_0 + \sum_{m=1}^{\infty} X_m^C \cos(m\omega_0 t) + X_m^S \sin(m\omega_0 t) \tag{3.13}$$

In order to enable implementation of the HB on a computer, the infinite series need to be truncated to a sufficiently large number of terms, M. Substituting (3.13) in the MNA formulation of the nonlinear circuit:

$$\mathbf{G}\mathbf{x}(t) + \mathbf{C}\frac{d\mathbf{x}(t)}{dt} + \mathbf{f}(\mathbf{x}(t)) = \mathbf{b}(t) \tag{3.14}$$

and expanding the third term by its Fourier representation:

$$\mathbf{f}(\mathbf{x}(t)) = F_0 + \sum_{m=1}^{M} F_m^C \cos(m\omega_0 t) + F_m^S \sin(m\omega_0 t) \tag{3.15}$$

will enable computing the Fourier coefficients of the response. To realize that, it should be noted that F_0, F_m^C and F_M^S are directly dependent on the waveforms in the $\mathbf{x}(t)$, who in turn are described by its Fourier coefficients X_0, X_m^C and X_m^S. Arranging X_0, X_m^C and X_m^S in a vector $\overline{X} \in \mathbb{R}^{N\times(2M+1)}$ where N is the MNA formulation size, yields:

$$\overline{X} = \begin{bmatrix} X_0^T & X_1^{C^T} & X_1^{S^T} & \cdots & X_K^{C^T} & X_K^{S^T} \end{bmatrix}^T \tag{3.16}$$

In general, a bar notation is used on top of a vector or matrix to indicate that this vector or matrix is composed of other sub-vectors or sub-matrices. The dependence of the coefficients of $\mathbf{f}(\mathbf{x}(t))$, F_0, F_1^C and F_1^S on the coefficients of $\mathbf{x}(t)$, X_0, X_1^C and X_1^S can be expressed more emphatically by rewriting (3.15) as:

$$\mathbf{f}(\mathbf{x}(t)) = F_0(\overline{X}) + \sum_{m=1}^{M} F_m^C(\overline{X})\cos(m\omega_0 t) + F_m^S(\overline{X})\sin(m\omega_0 t) \tag{3.17}$$

Thus, in Fourier representation will be:

$$\begin{aligned} &\mathbf{G}\left(X_0 + \sum_{m=1}^{M} X_m^C\cos(m\omega_0 t) + X_m^S\sin(m\omega_0 t)\right) + \\ &\mathbf{C}\left(\sum_{m=1}^{M} -m\omega_0 X_m^C\sin(m\omega_0 t) + m\omega_0 X_m^S\cos(m\omega_0 t)\right) + \\ &F_0(\overline{X}) + \sum_{m=1}^{M} F_m^C(\overline{X})\cos(m\omega_0 t) + F_m^S(\overline{X})\sin(m\omega_0 t) \\ &= B_0 + \sum_{k=1}^{K} B_k^C\cos(k\omega_0 t) + B_k^S\sin(k\omega_0 t) \end{aligned} \tag{3.18}$$

By using orthonormal property and similar steps expressed for (3.8), the following $2M+1$ systems of equations, each with size is N, are obtained:

$$
\begin{aligned}
\mathbf{G}X_0 + F_0(\overline{X}) &= B_0 \\
\mathbf{G}X_1^C + \omega_0\mathbf{C}X_1^S + F_1^C(\overline{X}) &= B_1^C \\
\mathbf{G}X_1^S - \omega_0\mathbf{C}X_1^C + F_1^S(\overline{X}) &= B_1^S \\
\vdots &= \vdots \\
\mathbf{G}X_K^C + K\omega_0\mathbf{C}X_K^S + F_K^C(\overline{X}) &= B_K^C \\
\mathbf{G}X_K^S - K\omega_0\mathbf{C}X_K^C + F_K^S(\overline{X}) &= B_K^S \\
\mathbf{G}X_{K+1}^C + (K+1)\omega_0\mathbf{C}X_{K+1}^S + F_{K+1}^C(\overline{X}) &= 0 \\
\mathbf{G}X_{K+1}^S - (K+1)\omega_0\mathbf{C}X_{K+1}^C + F_{K+1}^S(\overline{X}) &= 0 \\
\vdots &= \vdots \\
\mathbf{G}X_M^C + M\omega_0\mathbf{C}X_M^S + F_M^C(\overline{X}) &= 0 \\
\mathbf{G}X_M^S - M\omega_0\mathbf{C}X_M^C + F_M^S(\overline{X}) &= 0 \\
F_{M+1}^C(\overline{X}) &= 0 \\
F_{M+1}^S(\overline{X}) &= 0 \\
\vdots &= \vdots
\end{aligned}
\tag{3.19}
$$

The above system of nonlinear equation systems (3.19) is represented in the following form:

$$
\overline{Y}\,\overline{X} + \overline{F}(\overline{X}) = \overline{B}, \tag{3.20}
$$

where $\overline{Y} \in \mathbb{R}^{[N\times(2M+1)]\times[N\times(2M+1)]}$ is the following block diagonal matrix:

$$
\overline{Y} = \begin{bmatrix}
\mathbf{G} & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
0 & \mathbf{G} & \omega_0\mathbf{C} & 0 & 0 & \cdots & \cdots & \cdots & 0 \\
0 & -\omega_0\mathbf{C} & \mathbf{G} & 0 & 0 & \cdots & \cdots & \cdots & 0 \\
0 & 0 & 0 & \mathbf{G} & 2\omega_0\mathbf{C} & 0 & \cdots & \cdots & 0 \\
0 & 0 & 0 & -2\omega_0\mathbf{C} & \mathbf{G} & 0 & \cdots & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \mathbf{G} & M\omega_0\mathbf{C} \\
0 & 0 & 0 & \cdots & \cdots & \cdots & \cdots & -M\omega_0\mathbf{C} & \mathbf{G}
\end{bmatrix}
\tag{3.21}
$$

and, $\overline{F}(\overline{X})$ and $\overline{B}$ are $N \times (2M+1)$ vectors of Fourier coefficients:

$$\overline{F}(\overline{X}) = \begin{bmatrix} F_0(\overline{X}) \\ F_1^C(\overline{X}) \\ F_1^S(\overline{X}) \\ \vdots \\ F_M^C(\overline{X}) \\ F_M^S(\overline{X}) \end{bmatrix} \quad ; \quad \overline{B} = \begin{bmatrix} B_0 \\ B_1^C \\ B_1^S \\ \vdots \\ B_M^C \\ B_M^S \end{bmatrix} \tag{3.22}$$

In contrast to the case of linear circuits, the system of equations in (3.20) is nonlinear in $\overline{X}$ whose solution can only be obtained by iterative techniques. The next subsection outlines the NR method for solving a general system of nonlinear equations.

3.1.4. Solution of nonlinear System Using the NR

To illustrate the practical procedure to solve a system of nonlinear equations numerically, a system of n equations in n unknowns shown by:

$$\begin{aligned} \psi_1(x_1, x_2, \cdots, x_n) &= 0 \\ \psi_2(x_1, x_2, \cdots, x_n) &= 0 \\ &\vdots \\ \psi_n(x_1, x_2, \cdots, x_n) &= 0 \end{aligned} \tag{3.23}$$

The NR starts with an initial guess vector $x = x^{(0)}$, which might have a residual error vector Δ. The Jacobian matrix is formed by the partial derivatives of the nonlinear equations. Next, this error is multiplied by the inverse of Jacobian matrix to provide a correction term to find the next trial vector $x^{(1)}$. The above steps are represented by the following equations:

$$\Psi(x^{(0)}) = \Delta^{(0)} \tag{3.24}$$

$$\delta_x = (J\,|_{x=x^{(0)}}^{(0)})^{(-1)}\Delta^{(0)} \tag{3.25}$$

$$J = \begin{bmatrix} \frac{\partial\psi_1}{\partial x_1} & \cdots & \frac{\partial\psi_1}{\partial x_n} \\ \cdots & \ddots & \vdots \\ \frac{\partial\psi_n}{\partial x_1} & \cdots & \frac{\partial\psi_n}{\partial x_n} \end{bmatrix} \equiv \frac{\partial\Psi}{\partial x} \tag{3.26}$$

$$x^{(1)} = x^{(0)} - \delta_x \tag{3.27}$$

This process is repeated until the obtained vector converges sufficiently close to the solution. Algorithm 3.1 presents the NR method in the form of a simplified pseudo-code.

Algorithm 3.1 The pseudo code of NR method

Input : A system of nonlinear equations $\Psi(x)$
Output : A solution vector $x^{(.)}$ satisfying $\Psi(x^{(.)}) = 0$
begin
  $\epsilon \leftarrow$*A small positive number*; *Representing an acceptable error threshold.*
  $x^{(0)} \leftarrow$*A ininitial guess trial vector*; *Calculate the error of initial vector.*
  $\Delta^{(0)} \leftarrow \Psi(x^{(0)})$
  if $\| \Delta^{(0)} \| > \epsilon$ **then;** *Error is higher than an acceptable threshold.*
    $Convergence = false$
  else; *Error is lower than an acceptable threshold.*
    $Convergence = true$; $x^{(.)} \leftarrow x^{(0)}$;
  $j \leftarrow 0$; *Repeat until converging to a solution*
  while $Convergence = false$ **do;** *Compute at* $x = x^{(j)}$
    $J \leftarrow \frac{\partial\Psi}{\partial x}$ @ $x = x^{(j)}$
    $\Delta^{(j)} \leftarrow \Psi(x)$ @ $x = x^{(j)}$; *Compute the correction step for* $x^{(j)}$
    $\delta_x \leftarrow J^{-1}\Delta^{(j)}$; *Obtain the next trial vector.*
    $x^{(j+1)} \leftarrow x^j - \delta_x$; *Compute the error at the next trial vector.*
    $\Delta^{(j)} \leftarrow \Psi(x)$ @ $x = x^{(j)}$
    if $\| \Delta^{(j)} \| > \epsilon$ **then**; *Error is higher than an acceptable threshold.*
      $Convergence = false$
    else; *Error is lower than an acceptable threshold.*
      $Convergence = true$; $x^{(.)} \leftarrow x^{(j)}$;
    $j \leftarrow j + 1$
  return $x^{(.)}$
end

3.1.5. Application of the NR Iteration to HB Equations

The NR method is used to compute the solution vector of the HB equations (3.19), which can be represented by:

$$\overline{Y}\,\overline{X}^{(0)} + \overline{F}(\overline{X}^{(0)}) - \overline{B} = 0 \tag{3.28}$$

First, an initial guess vector $\overline{X}^{(0)}$ is needed. Typically the initial guess is constructed from the result of DC analysis. This is actually done by augmenting the DC vector, denoted here by $\mathbf{x}_{DC}$, by zeros as shown next:

$$\overline{X}^{(0)} = \left[\; [\mathbf{x}_{DC}] \quad \left[\; 0 \quad \cdots \quad 0 \;\right] \quad \cdots \quad \left[\; 0 \quad \cdots \quad 0 \;\right] \;\right]^T \tag{3.29}$$

The NR iteration then proceeds by updating the initial guess first computing the errors due to the initial guess, given by:

$$\Phi(\overline{X}^{(0)}) = \overline{Y}\,\overline{X}^{(0)} + \overline{F}(\overline{X}^{(0)}) - \overline{B}, \tag{3.30}$$

and then using the Jacobian matrix, obtained from:

$$\frac{\partial \Phi}{\partial X} = \overline{Y} + \frac{\partial \overline{F}}{\partial \overline{X}} \tag{3.31}$$

and then using:

$$\overline{X}^{(0)} = \left(\frac{\partial \Phi}{\partial X}\right)^{-1} \Phi(\overline{X}^{(0)}) \tag{3.32}$$

The above update rule is applied iteratively to compute successive solution vectors $\overline{X}^{(j)}$ thereby requiring computations for $\Phi(\overline{X}^{(0)})$ and $\frac{\partial \Phi}{\partial X}$ at each iteration. The process is

terminated if $\overline{X}^{(j)}$ produces a residual error below a pre-specified error tolerance. Due to the lack of direct analytic expressions for $\overline{F}$ in terms of $\overline{X}$, $\overline{Y}$, calculation of $\overline{F}(\overline{X}^{(j)})$ and $\frac{\partial \overline{F}(\overline{X}^{(j)})}{\partial \overline{X}^{(j)}}$ is done numerically as follows.

As described in (3.15) and (3.17), $\overline{F}(\overline{X}^{(j)})$ is computed from $\mathbf{f}(\mathbf{x}(t))$. The main steps involved in computing $\overline{F}(\overline{X}^{(j)})$ starting with a given vector $\overline{X}^{(j)}$ is summarized as follows:

1. $\overline{X}^{(j)}$ needs to be first ordered so that the Fourier coefficients for each variable of the MNA formulation are grouped together. This ordering is referred to as harmonic-minor node-major, since it orders the harmonics or Fourier coefficients first then the node or MNA variable second. A vector $\overline{X}^{(j)}$ that is ordered in the harmonic-minor node-major is denoted by adding a *"node"* subscript to it. The relation between a vector ordered in this mode to the initial ordering, such as the one in 3.16, can be expressed using the idea of permutation:

$$\overline{X}^{(j)}_{node} = P\overline{X}^{(j)} \tag{3.33}$$

   where P is a permutation matrix, i.e.

$$P^T P = I \tag{3.34}$$

2. The time domain samples at $t_0, t_1, ..., t_{2M}$ for all MNA variables are calculated by applying IDFT operator on each of the MNA variables. This can be expressed mathematically by the following transformation:

$$\overline{\mathbf{x}}^{(j)}_{node} = \overline{\mathbf{\Gamma}}^{-\mathbf{1}}\overline{X}^{(j)}_{node} \tag{3.35}$$

   where, $\overline{\mathbf{\Gamma}}^{-\mathbf{1}}$ is a block diagonal matrix that has the matrix $\mathbf{\Gamma}^{-\mathbf{1}}$ repeated along its diagonal N times, and N being the total number of MNA variables.

3. The time domain samples in the vector $\overline{\mathbf{x}}^{(j)}_{node}$ are used the values of the $\mathbf{f}(\mathbf{x}(t))$ at the same time instants $t_0, t_1, ..., t_{2M}$. The resulting vector of time samples is denoted $\overline{\mathbf{f}}^{(j)}_{node}$ to indicate that it is ordered in the harmonic-minor node-major mode.

4. Finally, $\overline{F}^{(j)}_{node}$ is computed by first applying the DFT for each of the MNA vari-

ables and then restoring it to the original ordering. Mathematically, this process is equivalent to the following transformation:

$$\overline{F}(\overline{X}^{(j)}) = P^T\,\overline{\mathbf{\Gamma}}\,\overline{\mathbf{f}}^{(j)}_{node}(x) \tag{3.36}$$

where, $\overline{\mathbf{\Gamma}}$ is a block diagonal matrix having $\mathbf{\Gamma}$ as its diagonal blocks. The pre-multiplication by $\overline{\mathbf{\Gamma}}$ serves to convert the time domain samples of $\mathbf{f}(\mathbf{x}(t))$ to the corresponding Fourier coefficients, while the pre-multiplication by P^T serves to restore the ordering to the initial ordering.

The above four steps show that $\overline{F}(\overline{X}^{(j)})$ can be written as:

$$\overline{F}(\overline{X}^{(j)}) = P^T\,\overline{\mathbf{\Gamma}}\,\overline{\mathbf{f}}^{(j)}_{node}(\overline{\mathbf{\Gamma}}^{\mathbf{-1}}\,P\,\overline{X}^{(j)}) \tag{3.37}$$

Notice if the initial ordering for $\overline{X}^{(j)}$ and $\overline{F}(\overline{X}^{(j)})$ has been selected to be in the node-minor harmonic-major (i.e. by grouping all the harmonics or Fourier coefficients for every single MNA variable together in contiguous sub-vectors), then P will be given by $N(2M+1)\times N(2M+1)$ identity matrix. Figure. 3.2 depicts the sequence of operations performed on $\overline{X}^{(j)}$ to compute $\overline{F}(\overline{X}^{(j)})$.

For the purpose of this thesis, the adopted initial ordering throughout all the computations will be based on node-minor harmonic-major. Hence, the permutation matrix needed for this purpose will be a simple $N(2M+1)\times N(2M+1)$ identity matrix. This choice for the ordering then enables to write $\overline{F}(\overline{X}^{(j)})$ as follows:

$$\overline{F}(\overline{X}^{(j)}) = \overline{\mathbf{\Gamma}}\,\overline{\mathbf{f}}^{(j)}_{node}(\overline{\mathbf{\Gamma}}^{\mathbf{-1}}\overline{X}^{(j)}) \tag{3.38}$$

The adoption of the harmonic-minor node-major ordering mode for the components of $\overline{X}$ will necessitate re-ordering the matrix $\overline{Y}$, since its current structure of (3.20) is based on the assumption that $\overline{X}$ is ordered in the node-minor harmonic-major ordering scheme.

Converting the ordering scheme of $\overline{Y}$ from the node-minor harmonic-major to the harmonic-minor node-major can be done by noting that $\overline{Y}$ in (3.21) can be represented

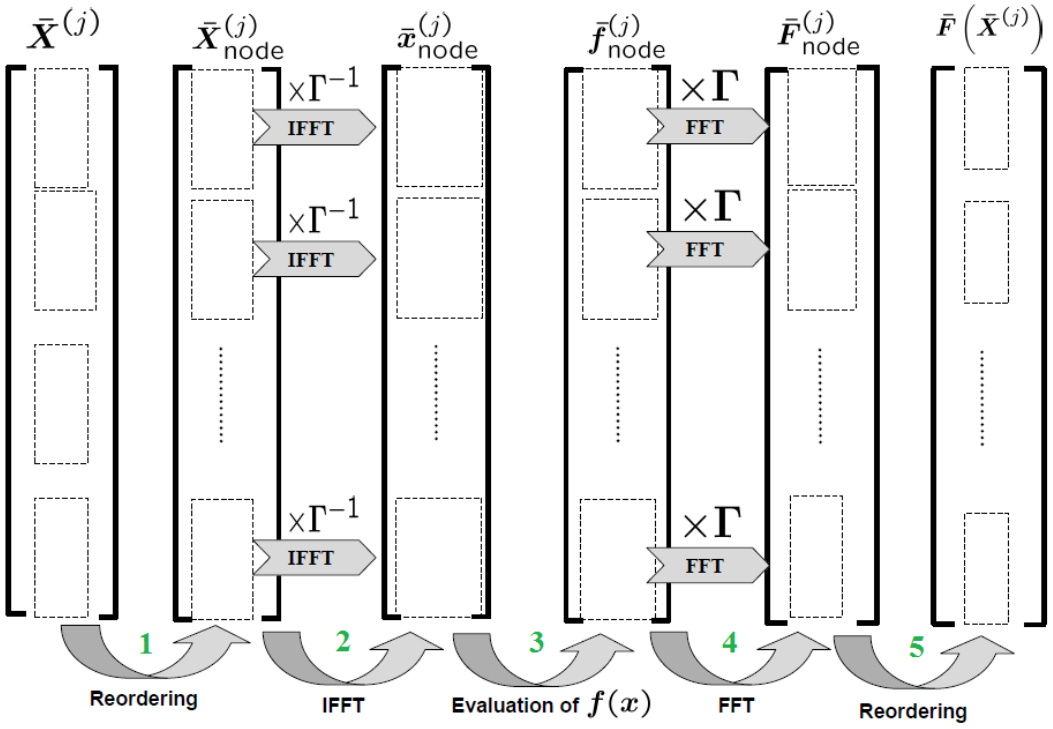


Figure 3.2.: Block diagram [40] of the required steps to compute $\overline{F}(\overline{X}^{(j)})$

as:

$$\overline{Y} = I \otimes \mathbf{G} + K_M \otimes \mathbf{C} \tag{3.39}$$

where $\otimes$ denotes the matrix Kronecker product, and K_M is defined as

$$K_M = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & T_\Lambda \otimes \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \end{bmatrix} \tag{3.40}$$

I is an identity matrix of size $N(2M+1) \times N(2M+1)$, and T_Λ is a diagonal matrix whose diagonal elements are the angular frequency components in the response, i.e. $\omega_0, 2\omega_0, 3\omega_0, ..., M\omega_0$. Rewriting the $\overline{Y}$ corresponding to the ordering of the $\overline{X}$ and $\overline{F}$ in the harmonic-minor node-major can be carried out by swapping the operands of the

Kronecker operation. In other words, if $\overline{Y}_{node}$ is used to refer to the $\overline{Y}$ matrix when its entries are permuted so that $\overline{X}$ is in harmonic-minor node-major, then $\overline{Y}_{node}$ is given by:

$$\overline{Y}_{node} = \mathbf{G} \otimes I + \mathbf{C} \otimes K_M \tag{3.41}$$

where the operands of the Kronecker have been swapped.

After having shown how to compute the $\overline{F}(\overline{X}^{(j)})$, now the focus is on computing the HB Jacobian matrix.

Assuming that harmonic-minor node-major is adopted for the HB problem, the overall HB Jacobian can be written as:

$$\frac{\partial \Phi}{\partial \overline{X}} = \overline{Y}_{node} + \frac{\partial \overline{F}(\overline{X}^{(j)})}{\partial \overline{X}^{(j)}} \tag{3.42}$$

where $\overline{F}(\overline{X}^{(j)})$ in this case is given by:

$$\overline{F}(\overline{X}^{(j)}) = \overline{\mathbf{\Gamma}} \times \overline{f}^{(j)}_{node}(\overline{\mathbf{\Gamma}}^{-\mathbf{1}} \overline{X}^{(j)}) \tag{3.43}$$

Here, the computation of the second term is the main issue. It can be shown using the chain rule of the differentiation that this term can be obtained as blocked $N \times N$ matrix, in which the blocks are of size $(2M+1) \times (2M+1)$. To illustrate this structure, $\frac{\partial \overline{F}(\overline{X}^{(j)})}{\partial \overline{X}^{(j)}}$ is written as follows:

$$\frac{\partial \overline{F}(\overline{X}^{(j)})}{\partial \overline{X}^{(j)}} = \begin{bmatrix} \Gamma \overline{\left[\frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_1}\right]} \Gamma^{-1} & \cdots & \Gamma \overline{\left[\frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_N}\right]} \Gamma^{-1} \\ \vdots & \ddots & \vdots \\ \Gamma \overline{\left[\frac{\partial \mathbf{f}_N}{\partial \mathbf{x}_1}\right]} \Gamma^{-1} & \cdots & \Gamma \overline{\left[\frac{\partial \mathbf{f}_N}{\partial \mathbf{x}_N}\right]} \Gamma^{-1} \end{bmatrix} \tag{3.44}$$

where the sub-matrices $\overline{\left[\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j}\right]}$ at an arbitrary entry (i, j) are given by:

$$\overline{\left[\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j}\right]} = \begin{bmatrix} \frac{\partial \mathbf{f}_i(\mathbf{x}_j(t_0))}{\partial \mathbf{x}_j(t_0)} & \cdots & \frac{\partial \mathbf{f}_i(\mathbf{x}_j(t_0))}{\partial \mathbf{x}_j(t_{2M})} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{f}_i(\mathbf{x}_j(t_{2M}))}{\partial \mathbf{x}_j(t_0)} & \cdots & \frac{\partial \mathbf{f}_i(\mathbf{x}_j(t_{2M}))}{\partial \mathbf{x}_j(t_{2M})} \end{bmatrix} \tag{3.45}$$

3.1.6. Structure of the HB Jacobian Matrix

This subsection takes a closer look at the structure of the Jacobian matrix, since its factorization represent the major computational efforts of the HB approach.

It should be noted that the sub-matrices $\overline{\left[\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j}\right]}$ in (3.44) are diagonal matrices. This is because the nonlinearity of the $\mathbf{f}(\mathbf{x}(t))$ is an algebraic nonlinearity, which implies that $\mathbf{f}(\mathbf{x}(t))$ depends only on $\mathbf{x}(t)$ at a specific time instant. In other words, $\mathbf{f}(\mathbf{x}(t))|_{t=t_p}$ depends only on $\mathbf{x}(t)$ at $t = t_p$ and not at any other time. Hence (3.45) is actually given by:

$$\overline{\left[\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j}\right]} = \begin{bmatrix} \frac{\partial \mathbf{f}_i(\mathbf{x}_j(t_0))}{\partial \mathbf{x}_j(t_0)} & 0 & \cdots & 0 \\ 0 & \frac{\partial \mathbf{f}_i(\mathbf{x}_j(t_1))}{\partial \mathbf{x}_j(t_1)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\partial \mathbf{f}_i(\mathbf{x}_j(t_{2M}))}{\partial \mathbf{x}_j(t_{2M})} \end{bmatrix} \tag{3.46}$$

Also, $\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j}$ at any instant will be non-zero, if, and only if, $\mathbf{f}_i$ is a function of $\mathbf{x}_j$. Given that $\mathbf{f}_i$ in a typical circuit is a function of only few entries in $\mathbf{x}$ (e.g. 2 or 3), it then follows that many of the block entries in (3.44) will be zeros.

Furthermore, those blocks that are not equal to zeros in $\frac{\partial \overline{F}(\overline{X}^{(j)})}{\partial \overline{X}^{(j)}}$ will be full blocks. This is because the IDFT and DFT operator matrices are full blocks.

Finally, by noting that the total Jacobian matrix is given by summation of $\overline{Y}_{node}$ and

$\frac{\partial \overline{F}}{\partial \overline{X}}$, i.e. :

$$\frac{\partial \Phi}{\partial \overline{X}} = \mathbf{G} \otimes I + \mathbf{C} \otimes K_M + \frac{\partial \overline{F}(\overline{X}^{(j)})}{\partial \overline{X}^{(j)}} \tag{3.47}$$

it becomes possible to see that the structure of the total Jacobian matrix can be described as a matrix of size $N(2M+1) \times N(2M+1)$ that is structured as $N \times N$ blocks whose sizes are $(2M+1) \times (2M+1)$, each. In addition, a block-entry (i, j) with $i = 1, ..., N$ and $j = 1, ..., N$ is structurally non-zero block if any of the following conditions are satisfied:

1. the (i, j) entry of the $\mathbf{G}$ matrix is non-zero,
2. the (i, j) entry of the $\mathbf{C}$ matrix is non-zero, or
3. $\mathbf{f}_i$ is a nonlinear function of $\mathbf{x}_j$,

otherwise, the block is a structural zero block. The above description indicates that there is resemblance between the sparsity pattern of the HB Jacobian matrix and the sparsity pattern of the matrix $\mathbf{G} + \mathbf{C} + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$, where $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ is the Jacobian matrix given by:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_1} & \cdots & \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_N} & \cdots & \frac{\partial \mathbf{f}_N}{\partial \mathbf{x}_N} \end{bmatrix} \tag{3.48}$$

This resemblance is understood by the following details. A non-zero entry at row i and column j in the in the matrix $\mathbf{G} + \mathbf{C} + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ implies a non-zero block between the rows $(i-1)(2M+1)+1$ up to $i(2M+1)$ and columns $(j-1)(2M+1)+1$ up to $j(2M+1)$ in the HB Jacobian matrix. Nonetheless, the structural properties within each non-zero block depend on whether the corresponding entry in $\mathbf{G} + \mathbf{C} + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ is due to $\mathbf{G}$, $\mathbf{C}$, $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ or due to any combination of those matrices, individually.

This structural feature of the HB Jacobian matrix is central to the main development in this thesis.

Figure. 3.3 shows the sparsity pattern for an OpAmp circuit with $N = 204$ MNA variables, and $H = 100$ Harmonics, sec. 6.1.

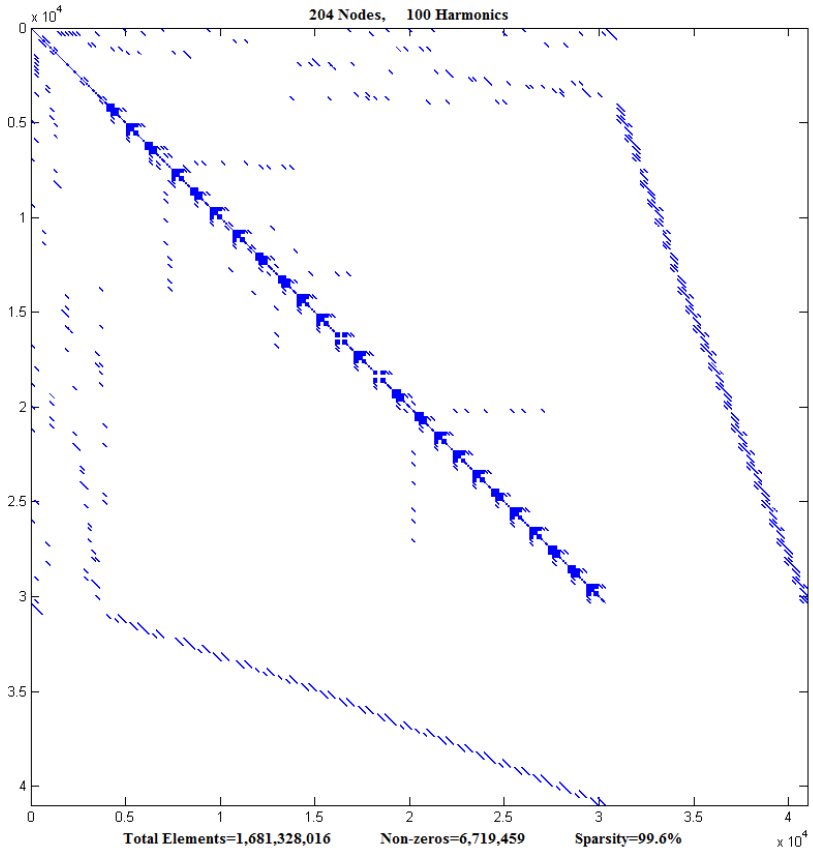


Figure 3.3.: Sparsity pattern of the Jacobian matrix of an inverting amplifier

3.1.7. Solution of the Linear Equations

The NR update requires the solution of the following linear system of equations:

$$A\,X = b, \tag{3.49}$$

where A the HB Jacobian matrix computed at the trial vector $\overline{\mathbf{x}}^{(j)}$ and b is the residual error arising from $\overline{\mathbf{x}}^{(j)}$:

$$A \equiv \left.\frac{\partial \Phi}{\partial \overline{\mathbf{x}}}\right|_{\overline{\mathbf{x}}=\overline{\mathbf{x}}^{(j)}} \tag{3.50}$$

$$b = \Phi(\overline{\mathbf{x}}^{(j)}) \tag{3.51}$$

Solving this system can be done by simply computing the inverse of the matrix A, A^{-1} and multiplying it by the vector b. However, this approach is not very effective. Instead, matrix A is first factorized into a product of a lower triangular matrix L, and an upper triangular matrix U, i.e.

$$A = L\,U \tag{3.52}$$

Substituting from (3.52) into (3.49), yields the system:

$$L\,U\,X = b \tag{3.53}$$

which can be solved by first solving

$$L\,y = b \tag{3.54}$$

for y using a process of forward substitution, and then solving

$$U\,X = y \tag{3.55}$$

for X using through a process of backward substitution.

The LU factorization requires ordering the matrix A to minimize the number fill-ins, which are the structurally zero entries in A that are replaced by non-zero entries in L or U. Further details on the LU factorization technique will be given in chapter 4.

3.2. The HB for Circuits With Multi-Tone Excitation

This section summarizes the extension of the HB analysis to circuits with multi-tone excitation. It shows basically how the HB problem can be modified to account for the more general case where the circuit is excited by one or more sources whose angular frequencies are not integer multiplies of a single angular frequency.

In general, a circuit with multi-tone excitation is a circuit whose response can be written in the generalized Fourier expression:

$$\mathbf{x}(t) = X_0 + \sum_{\substack{k_1,k_2,...,k_d=-\infty \\ k_i \neq 0\,,\ i=1,2,...,d}}^{\infty} X^C_{k_1,k_2,...,k_d} \cos\left(\left(\sum_{l=1}^{d} k_l\omega_l\right)t\right) + X^S_{k_1,k_2,...,k_d} \sin\left(\left(\sum_{l=1}^{d} k_l\omega_l\right)t\right) \tag{3.56}$$

where

- $\omega_l\,,\ l = 1,...,d$ is a set of d independent base tones,
- $X^{C,S}_{k_1,k_2,...,k_d}$ are the Fourier coefficients,
- $k_i\,,\ i = 1,...,d$ is a set of integers that can be positive or negative.

Note that the subscript attached to each coefficient is in fact a vector of integers. This is in contrast to the single-tone excitation where the subscript is a scalar positive integer. Note also, the designation of ω_l as a set of independent frequencies means that

$$\sum_{l=1}^{d} k_l\omega_l = 0\,,\ k_l\epsilon\mathbb{Z} \Longleftrightarrow k_1 = k_2 = ... = k_d = 0 \tag{3.57}$$

which means that those tones cannot be expressed as a multiple of a single frequency. The first step to practical implementation of the HB for a circuit with multi-tone response is to truncate the infinite Fourier series, in order to enable the computation on

the computing platforms. There are several schemes to truncate an infinite generalized Fourier series to a finite expansion. The following subsection illustrates the box truncation which is one of the widely used truncation schemes.

3.2.1. Box Truncation Schemes

This scheme is illustrated by assuming a two-tone circuit response, i.e. $d = 2$. In this case, the number of possible frequencies is represented by a two-dimensional infinite plan that extends in both dimensions as shown by Figure. 3.4.

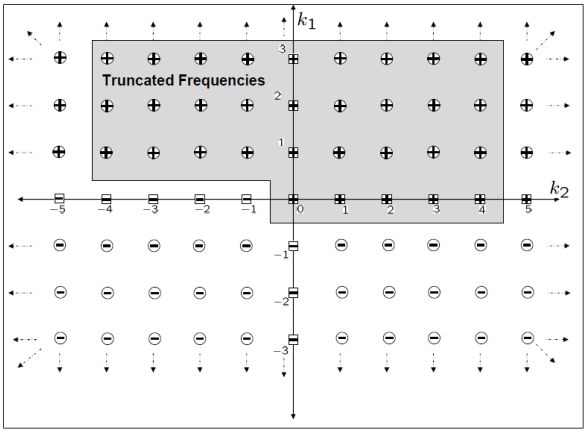


Figure 3.4.: An example of box truncation

A given point on this plan represents a potential combination between the two base tones, using two integers coefficients k_1 and k_2, where $k_1, k_2 = \pm 1, \pm 2, \pm 3, ...$

The objectives of truncation is to select a finite set of contiguous points that includes the origin while avoiding to include points representing frequencies that differ only by the origin. Figure. 3.4 shows a possible choice of a truncated selected points along the box boundaries. This is an example of a box truncation scheme.

The box truncation scheme is defined by specifying two integer bounds K_1 and K_2 (in this example $K_1 = 3$ and $K_2 = 4$, as shown by the shaded region) and selecting all frequencies given by $|k_1\omega_1 + k_2\omega_2|$ such that:

$$(0 \leqslant k_1 \leqslant K_1,\ |k_2| \leqslant K_2)\ and, (k_1 \neq 0\ if\ k_2 < 0) \tag{3.58}$$

The set of truncated frequencies Λ_M is described by

$$\Lambda_M = \{\lambda : \lambda = |k_1\omega_1 + k_2\omega_2| \;\; k_1, k_2 \in \mathbb{Z}; \\ (0 \leqslant k_1 \leqslant K_1, \; |k_2| \leqslant K_2) \; and, (k_1 \neq 0 \; if \; k_2 < 0)\} \tag{3.59}$$

where M is the number of non-zero frequencies calculated by

$$M = \frac{1}{2}((2K_1 + 1)(2K_2 + 1) - 1) \tag{3.60}$$

Now, the box truncation generalization to the multi-tone signals with d base frequencies $\omega_1, \omega_2, \omega_3, \ldots, \omega_d$ is represented by

$$\Lambda_M = \left\{\lambda : \lambda = \left|\sum_{l=1}^{d} k_l\omega_l\right| \; ; k_l \in \mathbb{Z}; |k_l| \leqslant K_l, \ldots, d; \; \textbf{first nonzero } \mathbf{k_l} \textbf{ positive}\right\} \tag{3.61}$$

where $K_1, K_2, ..., K_d$ are the truncation indices corresponding to each frequency, and the number of non-zero frequency components M in Λ_M is given by

$$M = \frac{1}{2}\left[\left(\prod_{l=1}^{d}(2K_2 + 1) - 1\right)\right] \tag{3.62}$$

Therefore, the truncated generalized Fourier series is written as:

$$\mathbf{x}(t_i) = X_0 + \sum_{\lambda_m \in \Lambda_M, \lambda_m \neq 0} X_m^C cos(\lambda_m t_i) + X_m^S sin(\lambda_m t_i) \tag{3.63}$$

3.2.2. The Multi-tone Problem

After having described how the multi-tone response is truncated, modification of the HB for generalized truncated Fourier expression is described.

The effect of multi-tone excitation can be first seen on the structure of the Y matrix. The structure of this matrix remains almost the same as in the case of the single-tone excitation. The only change that is needed for the multi-tone case is the replacement of the frequencies $p\omega_0$, which are multiples of a single tone ω_0, by the actual frequency in the truncated set, which is combination of more than one base frequency.

Another effect of the generalized Fourier series also impacts the way of computing $\overline{F}(\overline{X})$. This is because $\overline{X}$ does not consist of coefficients of harmonically related (i.e. multiples of single fundamental frequency) frequencies as it is the case of a single-tone response.

There are several approaches aimed at evaluating $\overline{F}(\overline{X})$ and $\frac{\partial \overline{F}}{\partial \overline{X}}$ for circuits with multi-tone excitations. In the following section, the technique of artificial frequency mapping technique is described for this purpose.

3.2.3. Artificial Frequency Mapping

The goal of the AFM technique is motivated by the observation that if the circuit contains only algebraic nonlinearity, then the Fourier coefficients of $\overline{F}(\overline{X})$ become independent of the frequencies. This basic idea is illustrated by considering the following nonlinearity:

$$f(x) = x + x^2 \tag{3.64}$$

which is a simple algebraic nonlinearity, and assuming that $x(t)$ is a simple two-tone given by:

$$x(t) = a_1 cos(\omega_1 t) + a_2 cos(\omega_2 t) \tag{3.65}$$

The substitution of (3.65) into (3.64) and rearranging shows that:

$$\begin{aligned} f(x(t)) = & \frac{1}{2}(a_1^2 + a_2^2) + a_1 cos(\omega_1 t) + a_2 cos(\omega_2 t) \\ & + \frac{a_1^2}{2} cos(2\omega_1 t) + \frac{a_2^2}{2} cos(2\omega_2 t) \\ & + a_1 a_2 cos((\omega_1 + \omega_2)t) + a_1 a_2 cos((\omega_1 - \omega_2)t) \end{aligned} \tag{3.66}$$

In other word, the set of box-truncated frequencies Λ_M, which is given by:

$$\Lambda_M = \{\omega_1, \omega_2, |\omega_1 - \omega_2|, \omega_1 + \omega_2, 2\omega_1 + 2\omega_2\} \tag{3.67}$$

does not appear in the actual Fourier coefficients of $f(x(t))$, since as shown by (3.66) those coefficients are given by:

$$\begin{aligned} F_0 &= \frac{1}{2}\left(a_1^2 + a_2^2\right) \\ F_1^C = a_1 \quad F_1^S &= 0 \\ F_2^C = a_2 \quad F_2^S &= 0 \\ F_3^C = a_1 a_2 \quad F_3^S &= 0 \\ F_4^C = a_1 a_2 \quad F_4^S &= 0 \\ F_5^C = \frac{a_1^2}{2} \quad F_5^S &= 0 \\ F_6^C = \frac{a_2^2}{2} \quad F_6^S &= 0 \end{aligned} \tag{3.68}$$

Therefore, in the situation where the nonlinearity is purely algebraic, the actual values of the frequency spectrum is irrelevant to the numerical value of the Fourier coefficients. This observation is taken advantage of by mapping the actual frequency spectrum Λ_M into another spectrum Ω_M, whose frequency points are harmonically related. Thus, using

$$\omega_1 \rightarrow 2\lambda_0 \ , \ \omega_2 \rightarrow 3\lambda_0 \tag{3.69}$$

leads to a new artificial frequency set Ω_M, given by

$$\Omega_M = \{\lambda_0, 2\lambda_0, 3\lambda_0, 4\lambda_0, 5\lambda_0, 6\lambda_0\} \tag{3.70}$$

and, substituting the artificial frequency components in Ω_M in place of the actual frequency components Λ_M in the expression of $f(x(t))$ produces

$$\begin{aligned} f(x(t)) = & \frac{1}{2}(a_1^2 + a_2^2) + a_1 cos(2\lambda_0 t) + a_2 cos(3\lambda_0 t) \\ & + \frac{a_1^2}{2} cos(4\lambda_0 t) + \frac{a_2^2}{2} cos(6\lambda_0 t) \\ & + a_1 a_2 cos(5\lambda_0 t) + a_1 a_2 cos(\lambda_0 t) \end{aligned} \tag{3.71}$$

Here, the artificial fundamental λ_0 can assume arbitrary values. Therefore, the above time-domain $f(x(t))$ waveform is not physically the actual waveform in the circuit. However, the Fourier coefficients of $f(x(t))$ are the same Fourier coefficients of the actual response. Thus, the above mapping enables the DFT and IDFT operators of the single-tone to handle multi-tone case. This is done by assuming that the Fourier coefficients of $x(t)$ are the Fourier coefficients corresponding to the frequency component in Ω_M, which are harmonically related, since they are all multiples of the artificial fundamental λ_0. Using IDFT operator, the time domain samples are then computed and used to compute the time-domain samples of the $f(x(t))$, which are converted back to the corresponding Fourier coefficients using DFT operator, as in the case of the single-tone case. The idea of frequency mapping is applied to a box truncated set of two frequencies ω_1 and ω_2 with truncation indices K_1 and K_2 as follows

$$\omega_1 \rightarrow \lambda_0 \ , \ \omega_2 \rightarrow (1 + 2K_1)\lambda_0 \tag{3.72}$$

and any arbitrary frequency element in Λ_M will be mapped as

$$|k_1\omega_1 + k_2\omega_2| \rightarrow |k_1 + k_2(1 + 2K_1)|\,\lambda_0 \tag{3.73}$$

$$p = |k_1 + k_2(1 + 2K_1)| \tag{3.74}$$

In general, box truncated set Λ_M with d base frequencies $\omega_1, \omega_2, \omega_3, \ldots, \omega_d$ is mapped by

$$\begin{aligned}
&\omega_1 \rightarrow \lambda_1\ ,\ \lambda_2 \rightarrow (1 + 2K_1)\lambda_1 \\
&\omega_2 \rightarrow \lambda_2\ ,\ \lambda_3 \rightarrow (1 + 2K_2)\lambda_2 \\
&\quad\vdots \\
&\quad\vdots\ ,\ \lambda_d \rightarrow (1 + 2K_{d-1})\lambda_{d-1} \\
&\omega_d \rightarrow \lambda_d
\end{aligned} \tag{3.75}$$

where $\lambda_1 > 0$ and the following relation is used to convert between original and mapped frequency sets

$$k = \frac{1}{\lambda_1}\left|\sum_{l=1}^{d} k_l\omega_l\right| \tag{3.76}$$

The charge-oriented MNA formulation is actually capable of representing all nonlinearities as pure algebraic nonlinear $f(x(t))$. Hence, type of formulation enables implementing the AFM technique to general nonlinear circuits.

3.2.4. Summary of the HB

The approach of the HB to the steady-state analysis problem is to formulate the problem as a system of nonlinear equations, whose unknowns are the Fourier coefficients of the circuit waveforms. The main issue to note here is the increased size of the problem. For example, a circuit whose MNA size is N, and is excited by two tones, ω_1, ω_2, each

truncated by two indices K_1 and K_2 will result in an HB of size $NH \times NH$, where

$$H = \frac{1}{2}\left(\left(2K_1 + 1\right)\left(2K_2 + 1\right) - 1\right) \tag{3.77}$$

Tables Table. 3.1 and Table. 3.2 lists the number of harmonics for circuits excited with two and three tones respectively.

Table 3.1.: Number of harmonics for two tone excitation

K_1	K_2	H
3	3	24
4	4	40
5	5	60
9	9	180
12	12	312

Table 3.2.: Number of harmonics for three tone excitation

K_1	K_2	K_3	H
3	3	3	171
4	4	4	364
5	5	5	665
6	6	6	1098

Therefore, the Jacobian matrix is an $NH \times NH$ matrix with block-wise structure, in which the size of the block is H, where H increases with the increase in the number of tones or the number of truncation index in each tone.

4. Block KLU on CPU

In this chapter, the implementation of the Block KLU algorithm, that is used to LU factorize the HB Jacobian matrix, is described. The chapter also presents the profiling of the Block KLU on a conventional CPU to demonstrate the main computational bottlenecks. Section 4.1 first describes the basic KLU algorithm [22]. Section 4.2 describes the extension of the KLU to a block-version that uses block as opposed to scalar operations. Section 4.3 describes the profile of the CPU execution time to illustrate the dominant computational efforts of the BKLU. Finally, section 4.4 presents some discussions.

4.1. KLU Factorization

The goal of the KLU algorithm [22] is solving the system of linear equation $\mathbf{Jx} = \mathbf{b}$ through factorizing the matrix $\mathbf{J}$ into its LU factors and using the process of Forward/Backward substitution to solve for $\mathbf{x}$. Figure. 4.1 shows the core of the basic KLU method for factorizing a matrix $\mathbf{J} \in \mathbb{R}^{N \times N}$.

The KLU algorithm is based on the Gilbert-Peierls factorization algorithm [44] which solves repeatedly the lower triangular system:

$$Ly = \mathbf{J}(:, k), \tag{4.1}$$

where the right hand side notation refers to the k^{th} column of the matrix $\mathbf{J}$. The matrix L is updated using the obtained solution vector y, as follows:

- at $k = 1$, L is initiated to an $N \times N$ identity matrix. In this case y is simply assigned to the first column of $\mathbf{J}$. Then y becomes the first column of L matrix

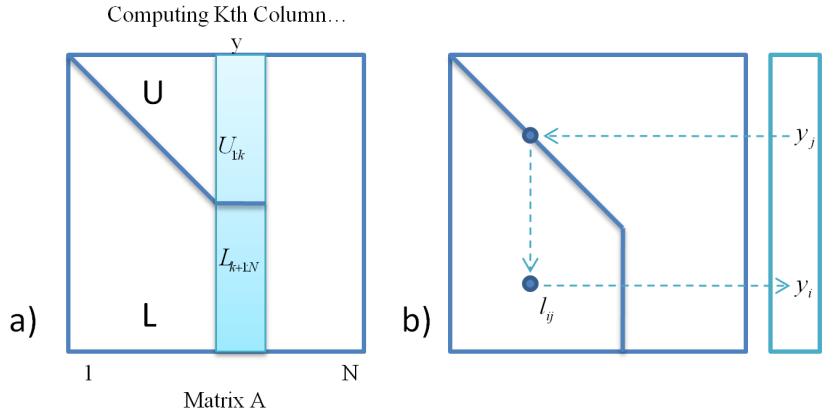


Figure 4.1.: a) Gilbert-Peierls left looking LU method, b) Sparse L Solve $Ly = x$

after dividing it by its first entry.

- for $k > 1$, y is obtained by forward substitution, and is used to update the k^{th} column in the L and U factors.

Algorithm 4.1 presents a basic description of the algorithm, utilizing a Matlab notation to denote the columns and matrices. Algorithm 4.1 calls the Lower-Triangular solution procedure (FS), described by Algorithm 4.2, each time using a column of **J** as the right-side vector.

Algorithm 4.1 Basic KLU factorization

1: **Input** : $J \in \mathbb{R}^{N \times N}$
2: **Output** : L, $U \in \mathbb{R}^{N \times N}$
3: **begin**
4: $L \leftarrow I_N$;
5: for $p \leftarrow 1$ to N do
6: $x \leftarrow J(:, p)$;
7: $y \leftarrow FS(L, x)$;
8: $k \leftarrow \max_i |y(i)|$;
9: ; *Do Partial Pivoting by interchanging rows k and p ;*
10: $U(1 : p, p) \leftarrow x(1 :, p)$;
11: $L(p : N, p) \leftarrow x(p :, N)/U(p, p)$;
12: **end**

Algorithm 4.2 FS: forward-solve $Lx = y$

1: **Input** : $L \in \mathbb{R}^{N \times N}$, $x \in \mathbb{R}^N$
2: **Output** : $y \in \mathbb{R}^N$
3: **begin**
4: ; *Compute the reachability set for all non-zero entries in x ;*
5: $Y \leftarrow$Reachables(x)
6: $y \leftarrow x$;
7: for $p \in Y$ do
8: $y(p+1:N) \leftarrow y(p+1:N) - L(p+1:N, p) \times y(p)$;
9: **end**

The above description of the KLU algorithm is the very basic form of the algorithm. However, the actual implementation involves more operations designed to take advantage of the sparsity of the matrix and improve its numerical conditioning. Those operations can be summarized by the following steps:

1. Ordering the columns and rows of **J** to reduce the fill-ins using one of the ordering approaches such as the Approximate Minimum Ordering (AMD) [45], COLAMD, or METIS ordering schemes,
2. Scaling the rows of J using a set of scalars determined based on the maximum absolute value in each row, or based on the sum of absolute values of the elements in each row, and
3. Partial pivoting, which refers to the process of permuting the rows of **J** to avoid dividing by very small diagonal elements.

Effectively, the above three steps combined, transform the system $\mathbf{Jx} = \mathbf{b}$ to the following system:

$$(PRJQ)\, Q^T x = PRb \tag{4.2}$$

The goal of the above three steps is to minimize the new fill-ins, which are the structural zeros in the entries of the matrix **J** that are replaced by non-zero entries in the L and U factors. Although, the role of the above three steps is essential in minimizing those fill-ins, the occurrence of fill-ins is still inevitable. The prediction of the locations of those entries is important to avoid unnecessary computations. This point is addressed

at line 5 of the Algorithm 4.2, which computes the set of indices in the solution vector, $\mathbf{x}$, that will be assigned non-zero entries, which will become the non-zero entries in L and U.

The process of computing or identifying those indices is typically carried out through a directed graph constructed from non-zero entries in the matrix L. This graph is typically constructed from N nodes, with edges between nodes j and i ; $(j \rightarrow i)$ if, $and\ only\ if$, $l_{ij} \neq 0$. The reachable set of indices obtained in this step represents the structural non-zero entries, including the new fill-ins (entries that were structural zeros in $\mathbf{J}$), of the upper and lower part of the p^{th} column of the matrices L and U, respectively. The reachable set of indices are found by conducting a depth-first search on the connected graph of L starting with nodes corresponding to the indices of the non-zero entries in the right side vector $\mathbf{b}$, which is a column taken from the matrix $\mathbf{J}$. The result of this depth-first search is what is known as the solution of the reachability problem, and is given by a set of indices ordered in a given topological order. This set of indices is denoted by `Reachables(x)` in the algorithm.

4.2. Block Aware KLU

The main focus of this thesis is on the direct factorization of the HB Jacobian matrix $\mathbf{J}$. The ordering used in (3.33) and the Kronecker product formulation clearly reveals that the structure of Jacobian matrix, $\mathbf{J}$, is indeed a sparse like structure, except for the fact that the entries are now block matrices with full structure and size $H \times H$. Although the structure of $\mathbf{J}$ can be considered to be sparse, its sparsity is not of the type that can be handled efficiently using direct LU factorization packages such as KLU, described in the previous section. In order to enable packages such as KLU to efficiently handle the factorization of HB Jacobian matrix, a block form of the basic factorization algorithm should be adopted. This indeed has been done in [46], where direct LU factorization based on this idea showed more robust performance compared with the iterative solvers based on Krylov based subspace methods, as long as the Jacobian matrix fits into the CPU memory.

The main idea of the BKLU is illustrated in the pseudo code of Algorithm 4.3 and Algorithm 4.4, which shows the KLU algorithm extended to a block-based operation.

It should be noted that the choice of the pivot block on line 13 of Algorithm 4.3 is done by first running Algorithm 4.1 on a test matrix whose entries sample the entries of the (3.20) calculated during a transient simulation or operating point calculation. This factorization produces the pivot choices for each column, thereby computing the exact reachability set Y of the Jacobian and the fill-in pattern in of its factors. Next, the solution to the reachability problem obtained for the test matrix is used as a block-wise reachable set for the matrix **J** in Algorithm 4.3 and Algorithm 4.4. The algorithmic pseudo code relies on the Matlab notation to express the main ideas of BKLU. For example, $A(r_1 : r_2; c_1 : c_2)$ denotes the submatrix of A that extends from row r_1 up to row r_2 and from column c_1 and up to column c_2. Thus, the assignment in lines 6 to 9 of Algorithm 4.3 are used to access the p^{th} block column in the matrix **J**.

Object Oriented Implementation The KLU algorithm described in section 4.1 is designed and implemented only for scalar sparse matrices. The block version of KLU has been implemented using object oriented features of the C++ language. An object class, `harmonic_balance_block`, contains a component named "type" to define the type of the blocks.

Algorithm 4.3 Block-KLU factorization

1: **Input** : $J \in \mathbb{R}^{(H\times N)\times(H\times N)}$
2: **Output** : L, $U \in \mathbb{R}^{(H\times N)\times(H\times N)}$
3: **begin**
4: $L \leftarrow I_{HN}$;
5: for $p \leftarrow 1$ to N do ; *Access the p^{th} block column of J*
6: $r_1 \leftarrow 1$; *First Row*
7: $r_2 \leftarrow HN$; *Last Row*
8: $c_1 \leftarrow (p-1)H+1$; *First Column*
9: $c_2 \leftarrow pH$; *Last Column*
10: $y \leftarrow BFS(L, J(r_1 : r_2, c_1 : c_2))$; *Solve th problem $L_y = J(r_1 : r_2, c_1 : c_2)$*
11: $r_2 \leftarrow pk$
12: $U((r_1 : r_2, c_1 : c_2)) \leftarrow y(r_1 : r_2, c_1 : c_2)$
13: $Q \leftarrow BPivot(y)$; *Choose pivot block in y*
14: for $q \leftarrow p+1$ to N do ;
15: $r_1 \leftarrow (q-1)k$
16: $r_2 \leftarrow qk$
17: $\underbrace{L((r_1 : r_2, c_1 : c_2)) \leftarrow y(r_1 : r_2, c_1 : c_2)Q^{-1}}$; *MINV + MMUL*
18: **end**

Algorithm 4.4 Block forward solve $Ly = x$

1: **Input** : $L \in \mathbb{R}^{(H\times N)\times(H\times N)}$, $x \in \mathbb{R}^{(H\times N)\times H}$
2: **Output** : $y \in \mathbb{R}^{(H\times N)\times H}$
3: **begin**
4: $y \leftarrow y$;
5: for $\forall p \leftarrow Y$ do ;
6: $r_1 \leftarrow pH + 1$
7: $r_2 \leftarrow (p+1)H$
8: $c_1 \leftarrow (p-1)H + 1$
9: $c_2 \leftarrow pH$
10: $y(r_1 : r_2, :) \leftarrow \underbrace{y(r_1 : r_2, :) - L((r_1 : r_2, c_1 : c_2)) \times y(c_1 : c_2, :)}$; *MMUL+MSUB*
11: **end**

The type and density of each block depends on the topology of the circuit, or more precisely, on the nature of the corresponding entry of the matrix obtained from (3.14). In fact, the blocks making up the HB Jacobian matrix can be one or a combination of the following blocks:

- A diagonal block of the form $g \times I_H$, g being a scalar, which arises if a circuit node has only linear resistive paths to the ground node. This type of blocks also arises from having other elements represented by their branch currents such as inductors, and independent/dependent voltage sources.
- A semi block-diagonal block of the form $c \times K_M$, c being a scalar, in which K_M is of the form (3.40). It is encountered when a circuit node has only capacitive paths to the ground, or it has linear inductors or (in the case of using charge-oriented MNA formulation) nonlinear capacitors or inductors.
- A full block, which arises when a circuit node has at least one nonlinear resistive path to the ground or when nonlinear capacitors, inductors, diodes, or transistors exist.

Operator Overloading The block-aware BKLU algorithm is complemented by implementing methods for the `harmonic_balance_block` class to overload the arithmetic operators so that their behavior becomes dependent on the types of the blocks involved in the operation. For example, the following assignment in KLU code:

Udiag[k] = X[k];

invokes an overloaded operator *"="* method to carry out a block move operation.

Memory Storage The memory storage of the matrices handled in the proposed approach is based on the Compressed Column Sparse (CCS) format. Figure. 4.2 depicts this approach by considering a block-structured matrix K in which the entries are block matrices. The CCS representation of K maintains three structures described as follows

1. K_i : This array has a size of nnz_K, where nnz_K is the number of non-zero block matrices in K, and it collects the rows indices of the non-zero blocks for all the columns. In this example $nnz_K = 15$.
2. K_p : This array has a size of N, it provides the offsets, within K_i, for the row entries of each column. For example, the rows indices of the non-zero entries for the third column, are given in between $K_i(K_p(3))$ to $K_i(K_p(4) - 1)$, which for this example produces 1, 3, 6.
3. K_z : An array of pointers to memory blocks of size H^2, assuming that the block is a matrix of size $H \times H$.
4. K_{id} : Which is an array of size nnz that flags the type of each block in K_z. There are three basic flags corresponding to the three basic types of blocks enumerated in the previous section, while blocks that are combinations of one or more of those basic blocks have their flags computed by a bit-wise operation of the corresponding flags. An additional flag is used to indicate a zero block or a block that needs to be cleared in preparation for an accumulation operation.

The BKLU maintains the above four structures for the original Jacobian matrix **J** as well as its L and U factors.

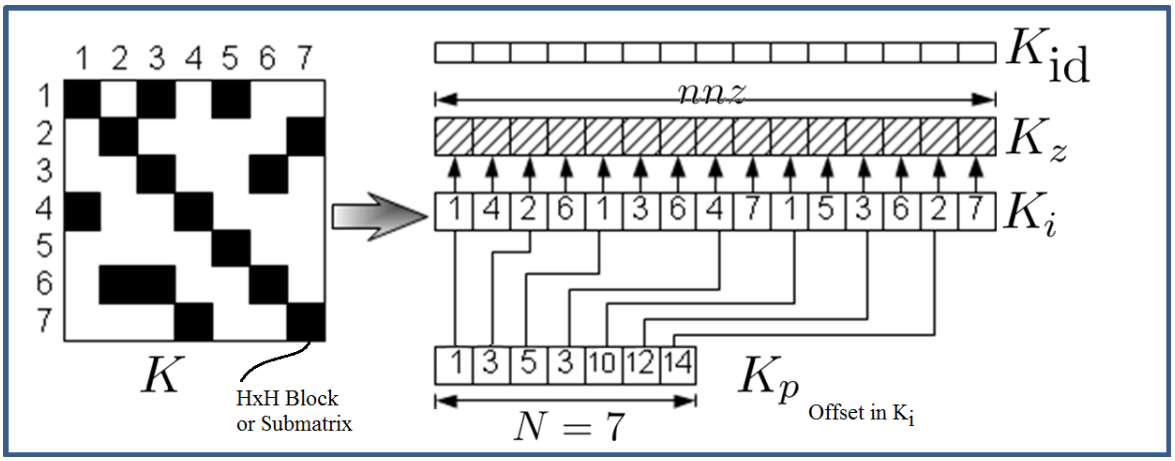


Figure 4.2.: An example of a block structured matrix with a CCS based representation

4.3. Potential for Parallel Implementation

At first glance on Algorithm 4.3 and Algorithm 4.4, it might seem that the BKLU is sequential. However, analyzing the execution of BKLU on a sample circuit reveals potential parallelism capacities in BKLU.

The block type oriented nature of BKLU provides a flexible platform to precisely measure the computation costs of the involved matrix operations. These are mainly: the matrix inversion (MINV) in line 17 of Algorithm 4.3, matrix-matrix multiplication (MMUL) in line 17 of Algorithm 4.3, and line 10 of Algorithm 4.4, and matrix-subtraction and accumulation (MSUB) in line 10 of Algorithm 4.4. Also, there are some implicit matrix operations like matrix scaling at the beginning of Algorithm 4.1 and moving a matrix between different blocks in line 10 of Algorithm 4.3 and line 6 of Algorithm 4.4. The nano-second clock source of the CPU is used to measure the timings of those operations.

Figure. 4.3 shows the CPU execution profile of the main floating point computational tasks of the BKLU algorithm used in a $\mu A - 741$ OpAmp circuit, described in detail in section 6.1, for different block sizes. This figure clearly shows that matrix-matrix multiplication dominates the computational tasks, especially for larger block sizes. The figure also indicates that matrix inversion occupies a significant place in the total

computational efforts. Finally, a small fraction of the overall computational operations is spent on overhead tasks such as moving matrix blocks and scaling them.

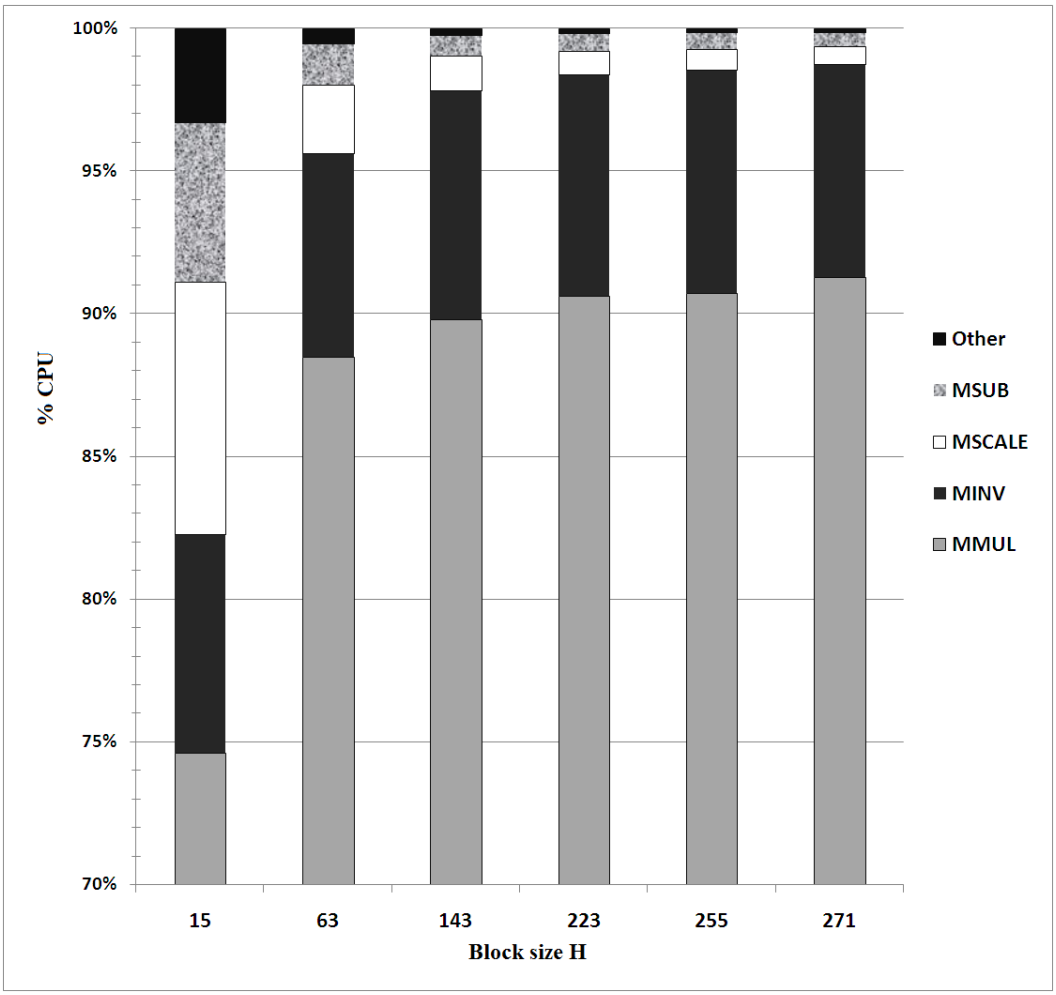


Figure 4.3.: CPU execution profile of the main computational tasks of BKLU

4.4. Summary and Discussion

The above execution profiling suggests strong potential for parallelism. This is because the bulk of computations takes place in the form of matrix-matrix multiplications. Indeed this operation is very well suited for implementation on hybrid platforms that includes a GPU. The goal of the next chapter is to take advantage of this fact by providing a complete implementation frame-work that runs on the GPU and CPU.

5. Hybrid Block KLU

This chapter describes the proposed approach to accelerate the direct factorization of the Jacobian matrix arising in the course of HB simulation of nonlinear circuits. Here, the proposed approach is called the Hybrid-BKLU, since it is designed to control and partition the work-load between the CPU and GPU in an efficient manner to ensure efficient utilization of the available resources. Data parallelism and hiding latency of data transfers, between CPU and GPU and different layers of GPU memory, are the key ideas needed to achieve speedup by simultaneous operation on large group of data on GPU's parallel processors. GPUs are designed to work on large amounts of data with simpler group of computations than CPUs. Whilst GPUs compute performance and memory bandwidth are larger than those of CPUs, they suffer from multilevel pointer accesses, branching and conditional statements. As a result, data structures and execution model should be carefully designed to gain speedup. In the following sections, first the memory architecture on GPU global memory is discussed, and next the execution model of the proposed approach are explained.

5.1. Proposed Hybrid-BKLU on GPU

This section presents the main conceptual stages of the proposed Hybrid BKLU factorization. First the structures used to represent the block matrices is described in subsection 5.1.1, and then subsection 5.1.2 outlines the description of the execution model.

5.1.1. Memory Structure

Figure. 5.1 presents a sketch of the main GPU memory allocations after the initialization phase has been completed. In the initialization phase, the OpenCL command

`clCreateBuffer` [32] is used to allocate buffers on the GPU global memory space to hold those blocks, that are of full type, of the HB Jacobian matrix, **J**, as well as for the resulting blocks of its **L** and **U** factors. A pre-scan process is performed on the symbolic KLU structure prior to the main re-factorization loop to extract the needed information about the number of full blocks in **J**, **L**, **U** and **X** (nfb_j, nfb_L, nfb_U and nfb_X respectively) as well as maximum length of **X** and **U** vectors (Kfb_X and Kfb_U respectively). Whilst all elements of **L** factors are required during the whole factorization process, only current column **U** factors take part in computations at each iteration. In order to efficiently utilize memory allocation on GPU, only Kfb_U number of full blocks are allocated for **U** factors and after computation of line 10 in Algorithm 4.3 **U** factors of current column are read-back to host. Also, at each loop iteration only Kfb_X number of full blocks are allocated for **X** vector.

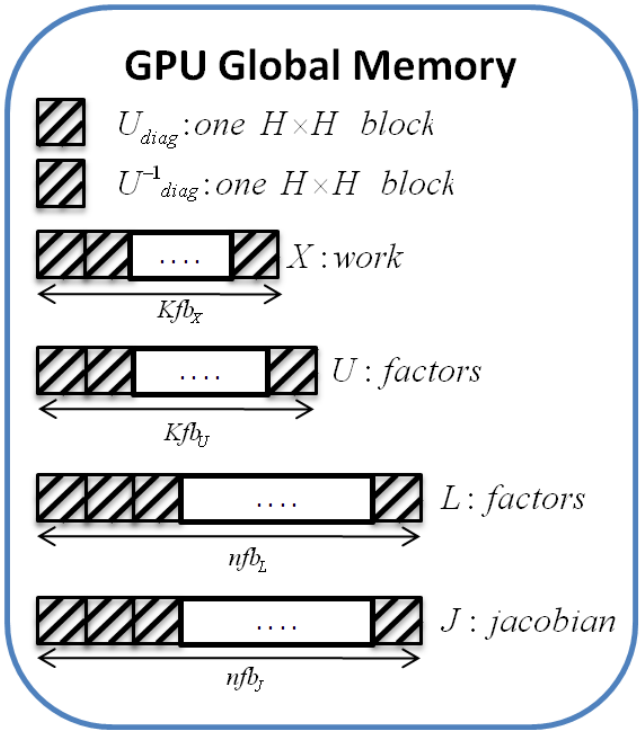


Figure 5.1.: GPU global memory allocation

5.1.2. Description of the Execution Model

Before the beginning of the execution, only the full blocks of the HB Jacobian, or **J** are copied from the CPU host memory to the GPU global memory. For convenience, the former is referred by $\mathbf{J}_{z,CPU}$, where the latter will be denoted by $\mathbf{J}_{z,GPU}$. On the other hand, the structural arrays describing the structures of the matrices **J**, **U** and **L** (i.e., $\mathbf{J}_{\{p,i,id\}}$, $\mathbf{U}_{\{p,i,id\}}$ and $\mathbf{L}_{\{p,i,id\}}$) are all kept on the host CPU memory. The host CPU memory also keeps track of the types of blocks stored in the work array $\mathbf{X}_{work,GPU}$ using a length N array of flags X_{id}.

At the beginning of each iteration of the Hybrid-BKLU, i.e. line 5 of Algorithm 4.3, the blocks of the p^{th} column of **J** are moved from the buffer $\mathbf{J}_{z,GPU}$ and spread to the work array buffer $\mathbf{X}_{work,GPU}$. Next, the process of blocked forward substitution is initiated as shown in Algorithm 4.4, where $\mathbf{X}_{work,GPU}$ is used as the right-side **x** and the solution **y** is written back in same the work space $\mathbf{X}_{work,GPU}$, which is used to fill in the blocks of the **L** and **U**.

The general execution during each iteration of the Hybrid-BKLU algorithm on the GPU platform is easily described through looking at the algorithm as consisting of the following execution units.

- High-level units, that mainly reside on the CPU host memory and mainly consist of
  - Block-scale. This unit multiplies a block (of size $H \times H$) by one of the scalar factors R_{ii} obtained from the KLU of the test matrix.
  - Block-Block-Multiplicition. This unit is dedicated to multiplying two matrices (size $H \times H$, each). This unit is invoked on line 10 of Algorithm 4.4.
  - Block-Inversion, which simply inverts an $H \times H$ matrix, and is called on line 15 of Algorithm 4.3.
  - Block-Subtract, to subtract two $H \times H$ matrices, which is invoked as part of line 10 of Algorithm 4.4.
  - Block-to-Block-Copy. This unit moves matrices between different buffers on the GPU or from the GPU to host memory CPU.
- Low-level units, which are more specialized versions of the above units, but are tailored for the specific types of blocks operated on. The higher level units

typically call the lower level ones based on the types of the blocks involved in the operation.

- GPU kernels, which is set of OpenCL kernels that execute the actual block operations on the GPU. Those kernels are compiled at run time by the program, and instantiated by the GPU hardware scheduler as work items and work groups on the GPU compute units. Those kernels are explained in detail in the Appendix-A.

During the execution of the outermost N iterations of Hybrid-BKLU (Lines 5-17), each of the high-level unit is activated at the appropriate stage of the algorithm. The unit is then passed the indices of the block, or blocks, to be operated on. The activated unit takes a decision as to whether the operation is best carried out on the host CPU or need to be offloaded to the GPU to obtain better performance. This decision is made based on the types of the input blocks as well as the type of the block resulting from the operation.

If the operation is judged, according to the types of the input and output blocks, to be best performed on the GPU, the unit initiates the following sequence of actions

1. It first calculates offsets into the buffers ($\mathbf{J}_{\{p,i,id\}}$, $\mathbf{U}_{\{p,i,id\}}$ and $\mathbf{L}_{\{p,i,id\}}$) corresponding to the indices of the parameter blocks needed in the operations.
2. Those offsets are sent to lower level units to setup the block operands as the kernel arguments to the specialized OpenCL kernels. The OpenCL command `clCreateSubBuffer` is typically very useful in such a situation, since it only returns a pointer to the starting position of the block involved in the operation.
3. The lower level unit places a command on the GPU command queue instructing it to execute the specified kernel. Based on the type of the kernel, the unit may wait until the kernel execution is finished or could return the control immediately to the higher-level unit. The latter case allows the CPU and GPU computations to overlap, while the former is required at certain synchronization points between the CPU and GPU.

Figure. 5.2 sketches the computational progress in the proposed approach for one iteration of the Hybrid-BKLU, indicating the units that are allowed to run in parallel and the main points of synchronization between the CPU and GPU.

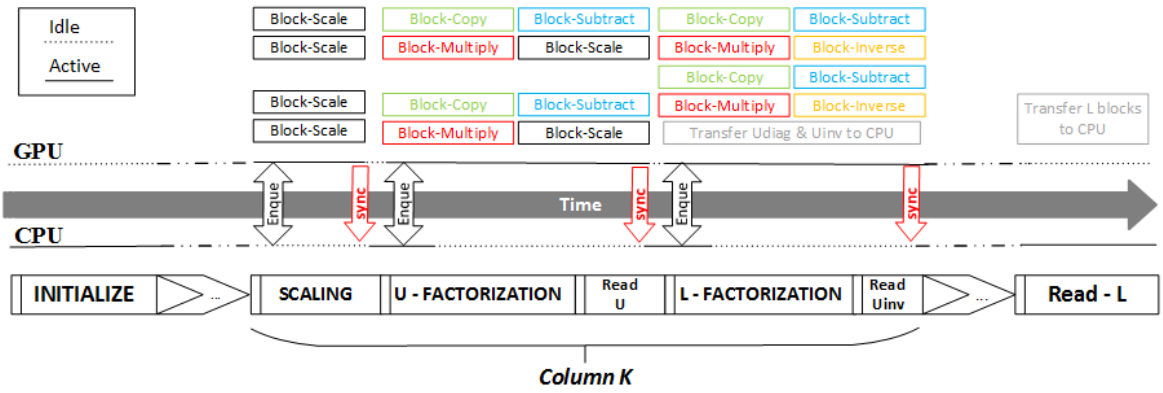


Figure 5.2.: Hybrid-BKLU execution model

It is worth noting that the matrix blocks, being stored in contiguous memory areas on the GPU global memory, enables the kernels to access the block operands through uniform memory access operations, which effectively improves the GPU performance especially for large block sizes. Another operation, which is not involved in the actual BKLU algorithms, but nonetheless is still required to obtain the final solution, is the process of Forward/Backward substitution with the right-side vector of (3.27). More specifically, during the backward substitution phase, the diagonal blocks of the $\mathbf{U}$ needs to be first inverted and then multiplied by the a sub-vector of $\Phi(\overline{X}^{(i-1)})$, size H, on the right-side. Given that the block inversion operation occupies a sizable part of the overall execution time, as seen from the BKLU profiling in Figure. 4.3, performing this operation on the GPU, concurrently with other operations, as shown in Figure. 5.2, offers further opportunity to increase the performance gains. This operation takes place on two stages

- At at the end of execution in Line 15 of Algorithm 4.3, the *Block-to-Block-Copy* unit is activated, to move the diagonal block of $\mathbf{U}$, i.e. $\mathbf{U}_{(c1:c2;c1:c2)}$ into the buffer denoted by $\mathbf{U}_{diag,GPU}$.
- Next the *Block-Inversion* is invoked to compute the inverse of the diagonal block and write the resulting matrix into the buffer $\mathbf{U}^{-1}_{diag,GPU}$, where the result is written back to a dedicated buffer on the host CPU memory.

After execution of the block forward solve in Line 10 of Algorithm 4.3, the *Block-to-Block-Copy* unit is activated to move the computed $\mathbf{U}$ factors of current column back to the host. Upon the termination of the Hybrid-BKLU algorithm, the blocks

constituting the factor matrices $\mathbf{L}$ are transferred back to the CPU host memory, whereupon they are utilized in the Forward/Backward substitution.

5.2. Optimizations

This section presents the GPU specific techniques implemented to optimize the performance of the proposed approach on the GPU architecture. The execution profiling presented in previous chapter for a test circuit, shows clearly that the process of matrix-matrix multiplications and matrix inversion occupy the largest portion of the computational efforts. This section presents the main GPU-specific techniques that leverage the performance in those operations and achieve significant speedup.

5.2.1. Block-Oriented Memory Alignment

An important key point to maximizing the floating point operations, in the matrix-matrix multiplication operations on GPU is to ensure coalesced memory access. More precisely, coalesced memory access enables performing vector floating point operations as a single instruction (e.g. using `double-n` data structures in OpenCL). Another key factor to leverage the GPU performance is to minimize the access conflicts to the Direct Memory Access (DMA) channels by work items executing on different wave-fronts.

It was found that achieving both objectives can be done through ensuring that the memory storage for the $H \times H$ matrix blocks is aligned to multiples of 256 bytes. On the AMD devices, this choice also serves to maximize the L1 cache performance of the GPU device since it is 4 multiples of the cache line width (64 bytes)[41].

Aligning the block storage to multiples of 256 bytes requires padding it with extra columns and rows. Furthermore, those extra columns/rows need to be properly initialized. In general, the padding columns/rows for all blocks are initialized to zeros. However, the exception to this rule is for the buffer $\mathbf{U}_{diag,GPU}$. This block is first initialized to a block diagonal entry in the matrix $\mathbf{U}$. Nonetheless, in order to ensure invertibility, its padding columns/rows are initialized by simply adding ones on the diagonal elements.

It must be noted that the above padding procedure represents an overhead in the computation and storage requirements of the GPU. However, the storage overhead diminishes progressively for problems with larger block sizes H. In addition, the computational overhead is offset by the gain in performance and speedup gained in using the vector operations on the GPU. To demonstrate this fact, Figure. 5.3 plots the amount of storage overhead required to align the blocks to multiples of 256 bytes versus the original block size, (i.e. size before incrementing to multiples of 256 bytes), indicating an increasingly smaller overhead.

To demonstrate the gain in performance arising from the proposed alignment, the same figure also shows the amount of speedup gained in the full matrix-matrix multiplication process, which represents the main bottleneck in the Hybrid-BKLU, after aligning the block size to a multiple of 256 bytes. The speedup shown in this figure is relative to the speed of performing the same operation on the block before aligning its size to multiples of 256 bytes through the padding operation illustrated above.

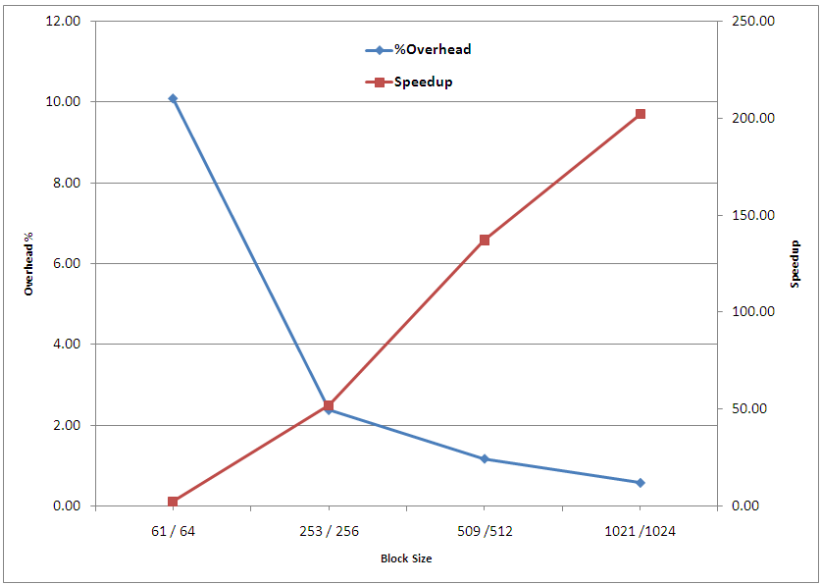


Figure 5.3.: Overhead in memory allocation vs. the original block size H

5.2.2. Optimized Block Inversion

The operation of block inversion is needed in two places:

1. During the main factorization, or more precisely in Algorithm 4.3 on line 17, for computing the inverse of the pivot block.
2. After the termination of the entire Hybrid-BKLU algorithm, where the process of forward/substitution with the error vector, Φ, is carried out to compute the update needed in the Newton iteration in (3.27).

Several approaches have addressed the inverting full matrices on parallel architectures. The technique adopted in this work uses block inversion process, through first computing a dense LU factorization on the block that needs to be inverted, followed by a forward/backward (F/B) substitution with the columns of an $H \times H$ identity matrix. The LU factorization step is carried out using the blocked LU factorization [42], [43]. The F/B substitution is implemented through a 2D kernel with H work groups, where each work group is responsible for calculating one column of the block inverse, with each work item being responsible for calculating few entries in the corresponding column. Experiments have shown that making each work item responsible for 8 or more entries produced the best performance. To show an idea of the performance of the adopted block inversion unit, Figure. 5.4 compares the time it takes to invert a block with several sizes with a direct Gaussian elimination provided by the AMD SDK, both implemented on GPU.

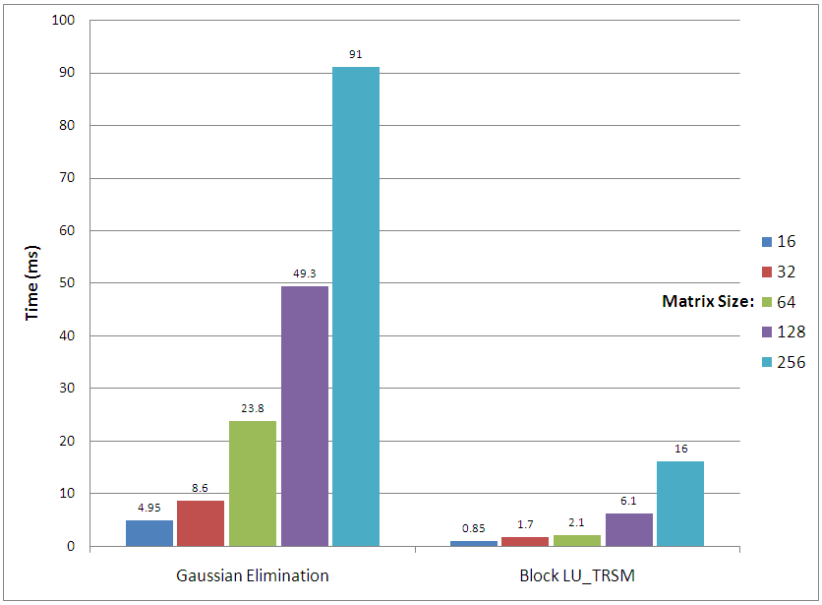


Figure 5.4.: Comparison between block inversion times

5.2.3. Optimization of GPU Memory Resources

The above description of the proposed approach assumes that the overall HB problem fits on the GPU global memory space. To enable this approach, however, to handle problems larger than the GPU global memory resources, an alternative implementation, with a slightly modified execution model was developed. In the alternative implementation, no buffers for blocks of the Jacobian matrix, $\mathbf{J}_z$, are pre-allocated on the GPU global memory before the beginning of the execution. Instead, those blocks are moved into the GPU only when needed during the execution. More specifically, at the beginning of each iteration of Algorithm 4.3 (Line 5), the p^{th} block column is transferred from the CPU host memory to the GPU local memory, and stored in the blocks of the workspace block array $\mathbf{X}_{work,GPU}$.

An additional saving in the GPU global memory space can be carried out by taking advantage of the fact that, in the proposed approach, the blocks of the $\mathbf{U}$ factor are not needed throughout the factorization process. This fact can be exploited as follows. Assuming that the maximum number of nonzero entries in any column of the $\mathbf{U}$ factor is q, then a storage for only q of $H \times H$ matrix blocks is allocated on the GPU during the initialization phase. The buffer allocated will then serve as a temporary storage for those blocks in the $\mathbf{X}_{work,GPU}$ that are destined for a given column in the $\mathbf{U}$ matrix.

Naturally, this process requires the additional overhead of using the PCIe bus during the execution of the algorithm en-queuing the necessary OpenCL commands needed to carry out the transfer. However, as will be shown in the simulation results, this overhead is greatly offset by the performance gain, especially for problems with large block sizes, H.

6. Simulation Results

6.1. Simulation Environment and Setup

The test platform on which all experiments are performed has the following specifications:

- Host: Intel ® Core™ i7-920 2.67 GHz, 12GB RAM
- GPU: AMD Radeon ™ HD-7950 Tahiti pro 900 MHz, 3GB RAM
- O.S: Linux 3.8.0.26, Ubuntu 13.04 64 bit
- AMD Catalyst™ 13.1
- OpenCL 1.2
- gcc Compiler 4.7.3

The proposed Hybrid-BKLU approach, and the BKLU on CPU as a reference, are used to simulate the steady state response, using harmonic balance, of five different circuits. The next paragraphs will briefly describe the circuits used in the simulation, while the simulation times on both CPU and GPU are given in the following section.

Tuned Amplifier The schematic for this circuit is shown in Figure. 6.1. This circuit acts as an amplifier with tank circuit based on a crystal. The model of the crystal is shown in the right part of Figure. 6.1. The circuit is designed to have narrow bandwidth centered at 5 MHz. The transistor model is Ebers-Moll model [39].

MOSFET-Based OpAmp The circuit used in this example is a feedback-based circuit with an operational amplifier as shown by the schematic of Figure. 6.2-(c). The OpAmp is constructed from MOSFET transistors, Figure. 6.2-(a), which also shows

the W/L ratio of transistors. Each MOSFET is modeled by the equivalent circuit in Figure. 6.2-(b).

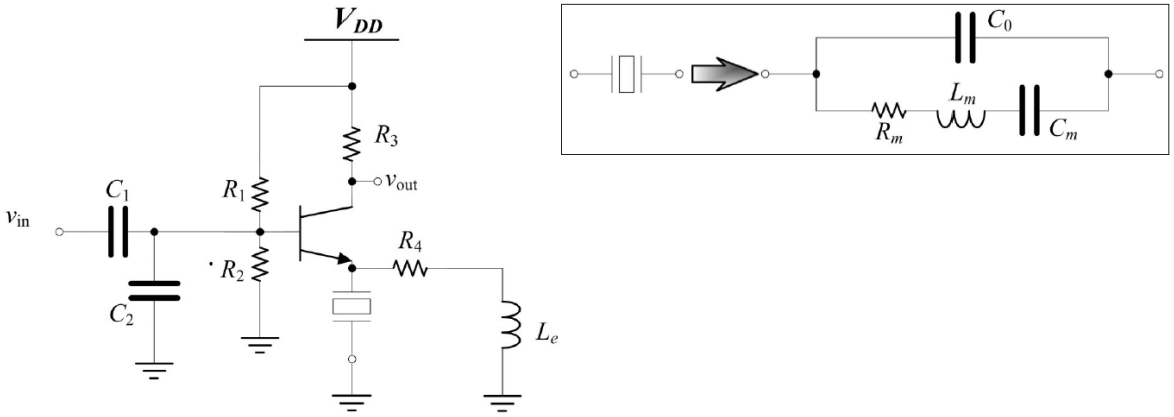


Figure 6.1.: Tuned amplifier circuit diagram

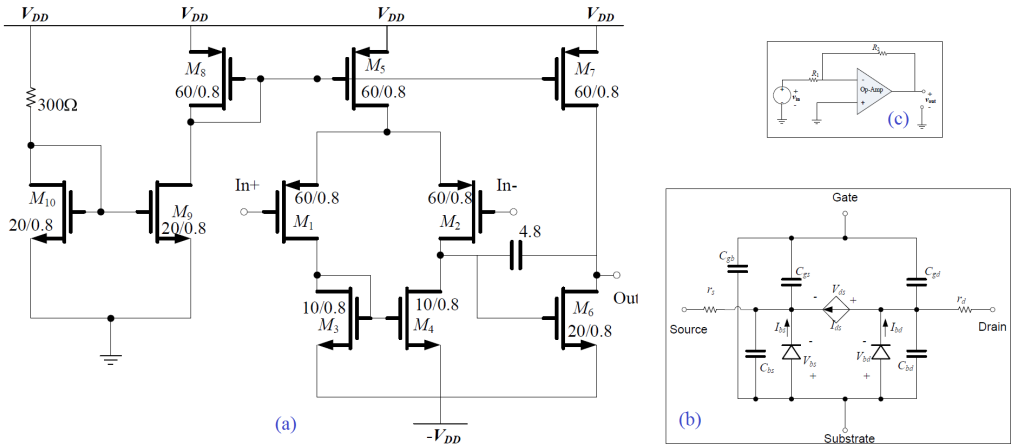


Figure 6.2.: a) Mos amplifier circuit diagram b) Mosfet transistor model

Double-Balanced Mixer Figure. 6.3 shows the schematic of a double-balanced mixer. The mixer is constructed from GaAs MESFET transistors which are modeled using the HSPICE [47] models. The crystals are modeled in a similar manner to Figure. 6.1.

BJT-Based OpAmp The test circuit is constructed by using the $\mu A-741$ operational amplifier in the feedback circuit as shown in Figure. 6.4. The BJT transistors are modeled by Ebers-Moll models [39].

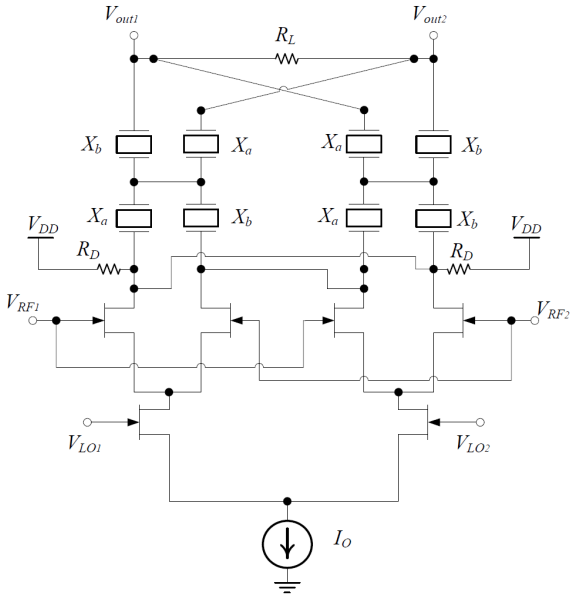


Figure 6.3.: Double balanced mixer circuit diagram

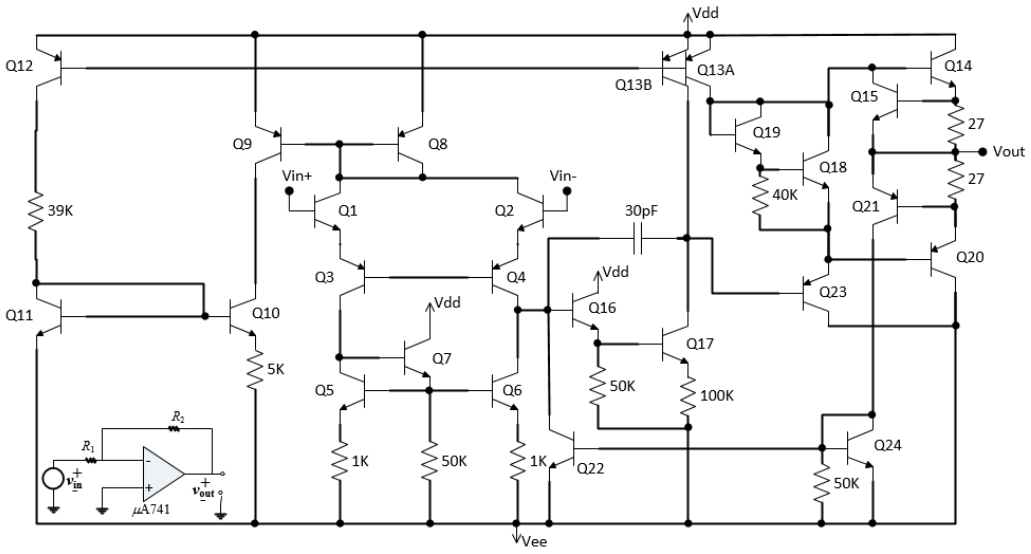


Figure 6.4.: BJT Operational amplifier circuit diagram

Active Low-Pass Filter The test circuit is adapted from the Cauer low-pass filter presented in [39]. The filter uses the blocks of BJT based $\mu A - 741$ operational amplifier, as shown in Figure. 6.5.

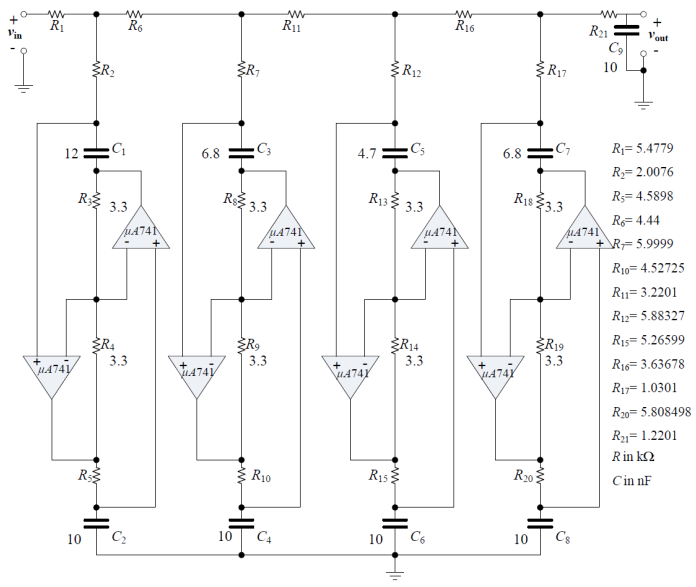


Figure 6.5.: Cauer low-pass filter circuit diagram

6.2. Execution Profiling

The following discussions are based on $\mu A - 741$ test circuit to provide a good comparison with profiling results from BKLU on CPU, represented in Figure. 4.3.

Figure. 6.6 shows the distribution of different matrix operation computation cost with block size $H = 192$. As can be seen, there is no dominant matrix operation and different groups of matrix operations (e.g. matrix inverse, move, scaling) are distributed approximately uniformly over time. Even, full matrix multiplication consumes about 20.7% of an NR iteration. Although spending 19.0% of time for transferring matrices between CPU and GPU might seem too much, it is just 0.54 seconds of 2.85 seconds of each NR iteration. The noticeable percentage of matrix scaling is due to the fact that most of the original matrix multiplications are simplified to multiplication of a matrix by a scalar. Finally, the significant percentage of matrix inverse suggests that further optimization on this operations could lead even to higher speedup gains. In contrast to other matrix operations that are used in this thesis, matrix inversion is inherently a serial algorithm and it is not possible to use vector operations to improve its execution time. As a result, matrix inversion takes longer execution time comparing to other matrix operations with the same number of mathematical operations.

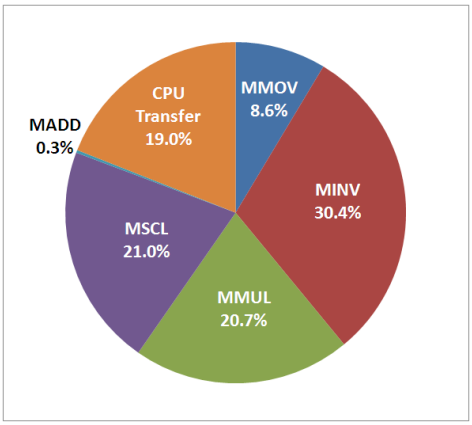


Figure 6.6.: Hybrid-BKLU profiling for $\mu A - 741$ OpAmp circuit

Figure. 6.7 shows the the percentage distribution (for several block size H) of the various tasks of the proposed approach grouped under the following four main categories: the GPU computation, the data transfer to and from the CPU host memory and the GPU global memory, the computational overlap between the GPU and CPU, and the CPU operations. This profile shows clearly that the GPU is engaged in the overall computation for most of the time. It also indicates that for larger block sizes, the GPU takes on a larger portion of the computations resulting in increasing speedup gain as shown in Table. 6.1. With the growth of block size, the percentage of CPU work decreases from 16% down to 5%.

This has been expected based on (2.10) which shows that with increasing size of the work the percentage of the serial part decreases and a significant speedup is obtained. The CPU part tends to be constant with the increase in the block size, which confirms that a part of the work could never be parallelized and limits the speedup, as predicted by (2.9). The breakdown depicts that GPU computation time is growing up to 80% of the execution time, and about 5% overlaps with computing on the CPU and 10% is spent for block transfer.

As explained in sec. 5.2, the whole Jacobian is transferred to the GPU if its size is less than 512 Mega byte. Figure. 6.8 provides a comparison between achieved speedup of two different methods for $\mu A - 741$ test circuit. With the growth of block size more speedup is achieved, which is due to data locality criterion. The amount of additional speedup varies for different circuits, however it is considerable value specially for smaller circuits.

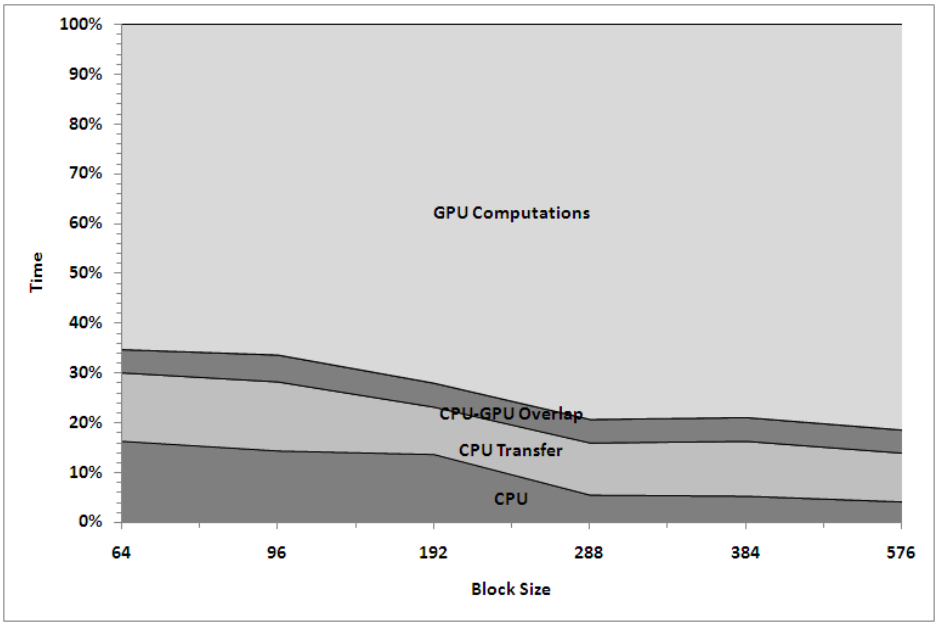


Figure 6.7.: Execution time breakdown for uA-741 OpAmp circuit

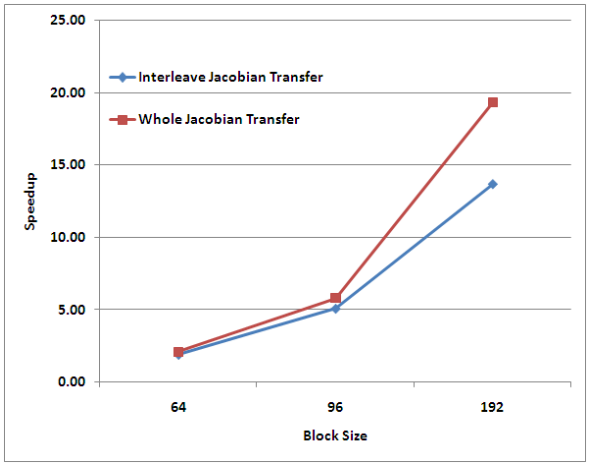


Figure 6.8.: Additional speedup by transferring whole Jacobian

6.3. Simulation Results

The proposed approach is applied to the previously mentioned test circuits and the simulation results are summarized in Table. 6.1. Most GPUs hardware incorporate a

mechanism to allocate resources for kernels, which behaves more efficient on repeated en-queuing of the same kernel. To achieve that efficiency, the error threshold of NR method is set to 1.0e-14. This threshold guarantees to have at least three iterations for the test circuits. Thus, the best execution times of iterations for each circuit are reported in Table. 6.1. This table is divided into five sections, where each section is dedicated to the simulation results of the specified circuit. The size of MNA formulation or the number circuit nodes, N, is indicated in the corresponding section. In each section of this table, the sparsity percentage of the Jacobian matrix as well as the following parameters are listed: nnz_L and nnz_U are the **L**'s and **U**'s number of non-zeros, nfb_L is the number of full blocks of **L**, and Kfb_U and Kfb_X are the maximum number of full blocks in columns of **U** and **X** respectively.

The test circuits are excited by several tones with several harmonics, thereby, producing several HB problems with increasingly larger block sizes, H, given in the first column of Table. 6.1. The second and third columns in this table indicate the size of the memory buffers required to store Jacobian matrix and its resulting LU factors in addition to temporary workspace **X** respectively. They are precisely defined by:

$$\begin{aligned} J &= nnz_J \times (H \times H \times SizeOf(double)) \quad (byte) \\ XLU_k &= (nfb_L + Kfb_U + Kfb_X + 3) \times (H \times H \times SizeOf(double)) \quad (byte) \end{aligned} \tag{6.1}$$

The fourth and fifth columns show the overall simulation time using the conventional CPU and the proposed hybrid GPU platforms respectively. The last column displays the speedup achieved in using the proposed GPU platform relative to the conventional CPU approach. Speedup is the ratio of the execution time of the BKLU on CPU over the execution time of the BKLU on GPU. The speedup results for all circuits are depicted in Figure. 6.9. There are some important points in Table. 6.1 and Figure. 6.9 that are worth highlighting:

- Maximum speedup of nearly 90 is achieved for $\mu A - 741$ test circuit.
- The consistent increase in performance gain for larger HB problems. This fact can be attributed to the special structure of the factorization algorithm which enables the concurrent execution of large contiguous blocks of floating point

operations. This feature of the Hybrid-BKLU maximizes the benefit of utilizing the GPU in the calculations.

Table 6.1.: Simulation results

Circuit	Parameter		H	J (MB)	XLU_k (MB)	CPU (sec)	Hybrid-BKLU (sec)	**Speedup**
Tuned Amplifier nodes = 19	sparsity =	82.55%	64	1.90	1.8	0.07	0.05	**1.30**
	nnz_L =	44	96	4.30	4.1	0.22	0.07	**3.14**
	unz_U =	50	192	17.50	16.2	1.80	0.13	**13.95**
	nfb_L =	25	288	39.60	36.4	6.20	0.22	**28.57**
	Kfb_U =	6	384	70.50	64.8	14.70	0.39	**37.31**
	Kfb_X =	6	576	159.00	145.8	56.30	0.91	**61.87**
MOS OpAmp nodes = 34	sparsity =	86.33%	64	4.80	3.5	0.37	0.16	**2.31**
	nnz_L =	93	96	10.90	7.9	1.27	0.21	**6.14**
	unz_U =	119	192	44.00	31.8	10.40	0.41	**25.24**
	nfb_L =	59	288	99.00	71.7	35.90	0.79	**45.10**
	Kfb_U =	10	384	177.00	127.4	85.90	1.45	**59.24**
	Kfb_X =	10	576	399.00	286.7	328.00	4.41	**74.36**
Balanced Mixer nodes = 57	sparsity =	93.19%	64	6.70	6.1	0.65	0.25	**2.62**
	nnz_L =	161	96	14.00	13.6	1.97	0.33	**5.97**
	unz_U =	237	176	46.00	45.7	11.60	0.52	**22.39**
	nfb_L =	104	256	110.00	96.7	43.40	1.06	**40.94**
	Kfb_U =	22	368	220.00	199.8	126.00	2.29	**55.02**
	Kfb_X =	12	544	472.00	436.7	435.00	5.56	**78.24**
BJT OpAmp nodes = 204	sparsity =	97.90%	64	26.4	22.9	1.9	0.91	**2.09**
	nnz_L =	701	96	60	51.6	6.7	1.12	**5.98**
	unz_U =	674	192	242	206.5	55.4	2.85	**19.44**
	nfb_L =	497	288	547	465	187	6.44	**29.04**
	Kfb_U =	34	384	975	826	405	9.14	**44.31**
	Kfb_X =	14	576	2197	1859	1707	19.02	***89.75***
Low-pass Filter nodes = 1599	sparsity =	99.74%	64	201	186	23.1	7.8	**2.95**
	nnz_L =	5714	96	458	418	80	13.2	**6.11**
	unz_U =	5506	128	818	747	195	20.5	**9.51**
	nfb_L =	4115	176	1533	1406	510	26.0	**19.62**
	Kfb_U =	276	192	1850	1673	675	31.5	**21.43**
	Kfb_X =	98	256	3296	3045	1654	36.3	**45.56**

- Almost 30% of **L** and **U** elements are not full blocks (the ratio of $\frac{nnz-nfb}{nnz} \times 100$). The results show the importance of an efficient memory structure design for full

blocks, and a hybrid execution method based on block types.

- The slopes of speedup curves for MOS based circuits are more than that for BJT circuits. This is resulted from MOS transistor model which generates more diagonal blocks than BJT transistor model.
- The last three rows of $\mu A - 741$ OpAmp and low-pass filter results, which are marked by gray highlighting, denote the cases where the total Jacobian matrix memory are exceeded the 512 Mega Bytes size. As mentioned earlier, when the size of the Jacobian matrix crosses this thresholds, the Jacobian matrix is transferred from the CPU host memory to the GPU global memory one column at a time at the beginning of each iteration and gets buffered to working matrices $\mathbf{X}$ directly. The block size at which the size of the Jacobian matrix crossed the threshold of 512MB is on the speedup graph of Figure. 6.9 by $\boxplus$ mark.
- The marks on the above speedup graph indicate a slight decrease in the rate of increase of speedup with regard to the block size. This can be attributed to the fact that the Jacobian matrix for these two examples had to be transferred one column at a time beyond the block size marked at the graph.

Finally, Figure. 6.10 shows a comparison of speedup achieved using different matrix inversion methods. The graph is based on the results of the early implementation of this work before further optimizations, and clearly depicts the performance of using proposed block LU-TRSM vs. naive Gaussian elimination method.

6.4. Numerical Results

This section addresses the accuracy of the results obtained from the Hybrid-BKLU. The question of how accurate the results obtained using GPU is important for the following reason. The GPU represents double precision floating point numbers using the IEEE-754 standard which is based on 64 bit double precision representation. On the other hand, all floating point registers of multi-core processors, such as Intel ® Core™ i7, are 80 bit wide. The discrepancy between the internal floating point operations on the CPU and those on the GPU may suggest a loss of accuracy in the obtained results. This section addresses this concern by first taking a closer look at the convergence behavior of the Hybrid-BKLU on test circuits.

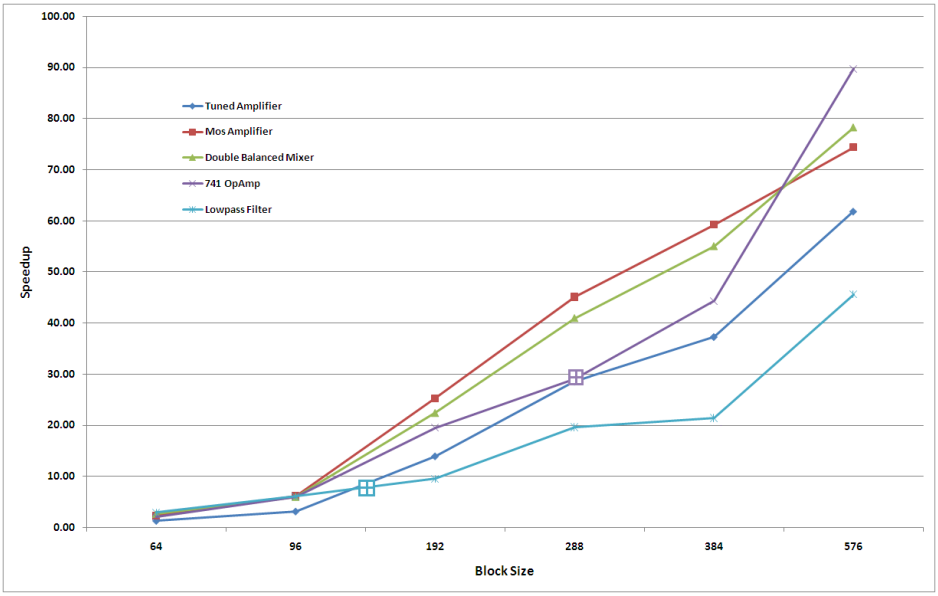


Figure 6.9.: Speedup for sample circuits

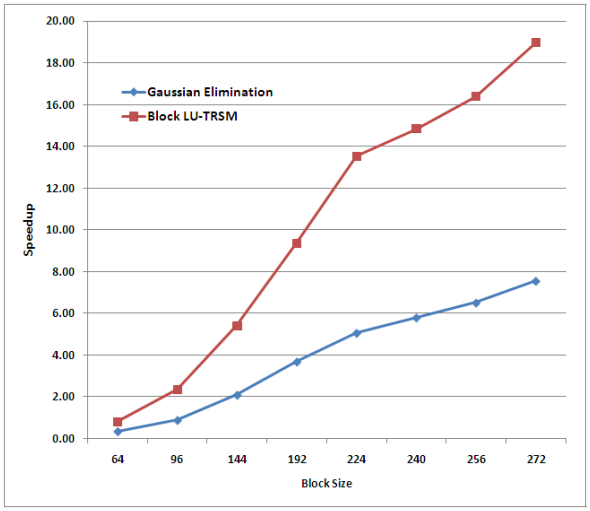


Figure 6.10.: Performance optimization using different matrix inversion methods

Although, all BKLU simulations on CPU are performed with higher precision, Table. 6.2 shows that the number of NR iterations for both BKLU on CPU and Hybrid-BKLU are equal. Moreover, the converged error of Hybrid-BKLU (3.24) is approximately equal to that of BKLU on CPU.

Table 6.2.: Convergence error results of sample circuits

Parameter / Circuit	CPU iterations	Hybrid iterations	CPU Converged error	Hybrid Converged error
Tuned Amplifier	5	5	7.68579E-014	4.10099E-014
Mos Amplifier	13	13	1.05966E-014	1.05550E-014
Balanced Mixer	6	6	2.46982E-014	2.31621E-014
BJT OpAmp	4	4	5.67133E-015	5.67644E-015
Low-pass Filter	5	5	3.35700E-015	7.68920E-015

Also, the numerical results obtained by the Hybrid-BKLU approach and the conventional CPU approach for the test circuit of the double-balanced mixer excited by an input Local Oscillator (LO) frequency of 1 GHz and input Radio Frequency (RF) of 900 MHz. The numerical values of harmonic spectrum at the output node is plotted in Figure. 6.11 for both simulations of BKLU on CPU and proposed GPU-based algorithm. As can be seen from the figure, the results match very accurately.

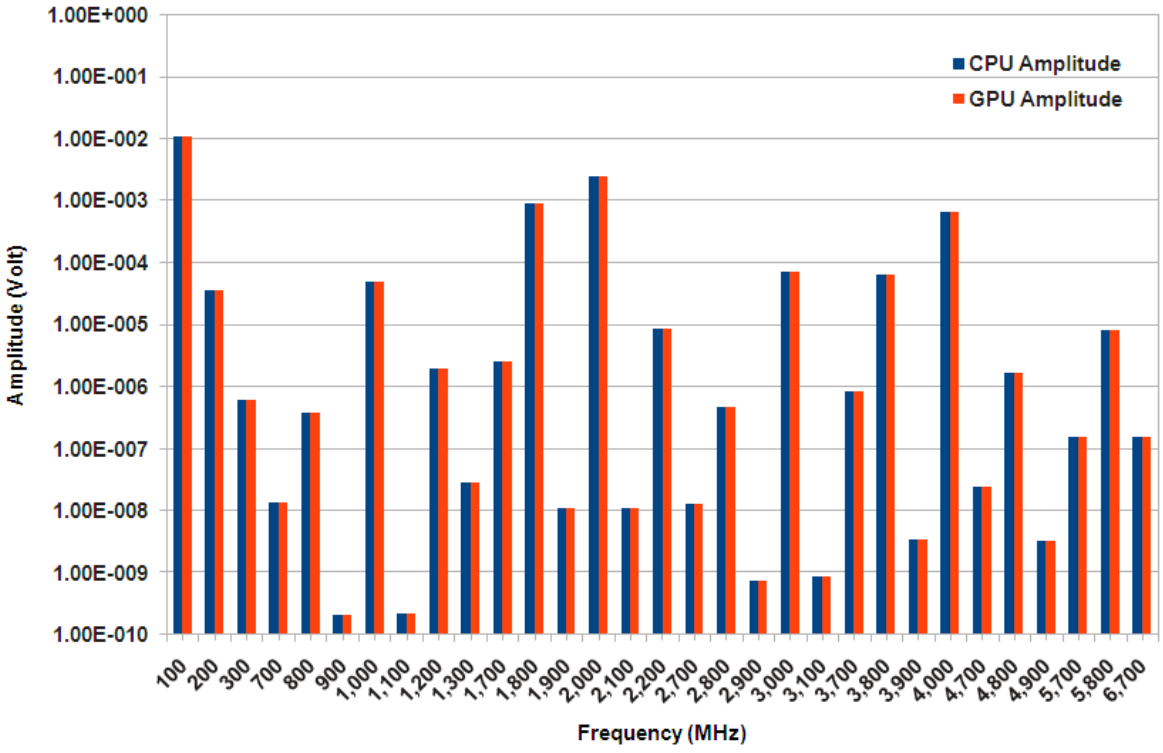


Figure 6.11.: Harmonic spectrum of the balanced mixer circuit

7. Conclusion

7.1. Concluding Remarks

This thesis presented a new methodology to accelerate the HB approach on computational platforms equipped with Graphics Processing Unit. The proposed approach is based on developing a special form of the direct block-wise factorization algorithm built from the KLU factorization code. The developed approach enabled isolating floating-point intensive operations in contiguous blocks that are offloaded to the GPU device. This feature enabled the GPU to utilize its massive parallel architecture very effectively. Relative speedup of up to 89 times over conventional CPU implementation has been reported for several examples.

7.2. Future Work

Some examples of further research directions based on the proposed approach are:

1. This thesis presumed that HB problem should fit in a GPU global memory. A complementary work can utilize heterogeneous platform including multiple GPUs to go beyond this barrier. Since L factors are required during all factorization process, the work should provide efficient distribution of column factorization between GPUs as well as transferring the L and U factors with minimum latency.
2. Another interesting direction, using multiple GPUs, can be realized by task parallelism. The work done in [24] for achieving task parallelization of the LU factorization can be combined with the proposed approach to achieve further speedup.

3. A generalized BKLU algorithm can be prepared for sparse system solve. Simulation of many systems requires factorization and solving a system of equations. Harmonic balance analysis of the steady state situation of a mechanical system is an example of such systems.

4. The proposed approach can be extended to other circuit simulations such as mixed time-frequency domain steady state analysis. These problems can efficiently incorporate Hybrid-BKLU method to solve produced system of equations.

A. Kernels Description

A.0.1. Simple Kernels

The following kernels are implemented to perform the simpler cases in Algorithm 4.3 and Algorithm 4.4.

kl_MatrixAddDiagScalar This is a 1D kernel that adds a scalar to the diagonal elements of a matrix. The number of total work-items is set to the effective size of the matrix. Each work-item adds the scalar to one diagonal element of the matrix, whose index corresponds to the work-item's global-ID number.

kl_MatrixUdiagUinv This is a 2D kernel that constructs a special matrix to find the inverse of pivot matrix (a diagonal block element of the U matrix in the BKLU method). Since the LU method used to inverse a matrix is based on 16-elements blocks, the number of total work-items is set to the effective size of the matrix and the number of work-items in each work-group is set to 16×16 (256). For example, if the pivot matrix size is 28×28, the total number of work-items will be 32×32, containing 4 work-groups of 16×16 work-items. Each work-item is responsible to initially clear one element of the destination matrix and copy corresponding elements from U_{diag} matrix. If both dimension IDs of a work-item are less than the size of the pivot matrix, it will copy that element of the pivot matrix to the destination. Otherwise if its dimension IDs are equal, it will fill the destination element by value 1.0. This structure is required to keep invertibility property when enlarging a matrix.

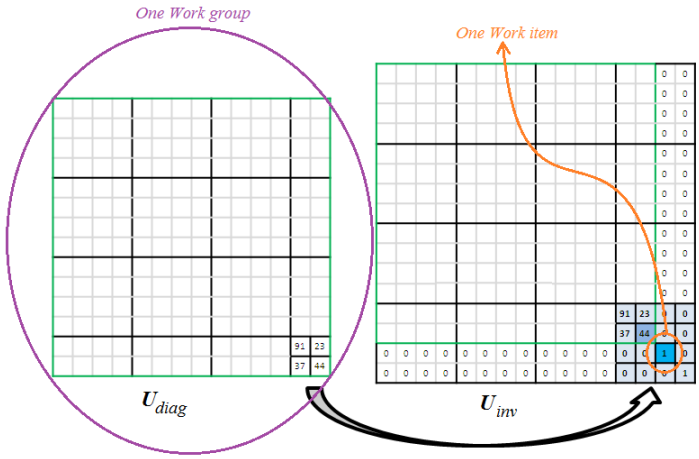


Figure A.1.: Preparing a matrix for inversion by 16 work-items per work-group

kl_MatrixAzScaletoX This is a special 2D kernel that only scales and copies a non-zero block of the Jacobian matrix to the current working array of blocks, X. As explained in sec. 5.1.1, the original size of each block is increased to a multiple of 4 double precision words (32 bytes). The number of total work-items is set to the effective size of the matrix and the number of work-items in each work-group is set to 4×4 (16). Each work-item is responsible for an element of the destination matrix, and initially clears its location. If both dimension IDs of a work-item are less than the size of the original matrix, it will scale that element and will copy the result to the destination.

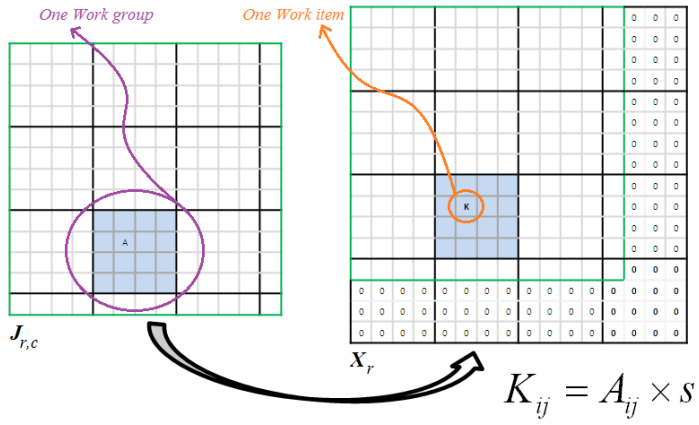


Figure A.2.: Scaling a matrix by 16 work-items per work-group

kl_MatrixAscaletoB This is a 2D kernel that scales and copies a matrix to anther matrix. In contrast to the previous kernel, in this kernel the matrices are defined as multiples of 4 double precision words, therefore scaling and copying operation are implemented using quad vector data structure and instructions. The number of total work-items is set to the effective size of the matrix divided by vector size (four), and the number of work-items in each work-group is set to 4×4 (16). Each work-item is responsible for 16 elements of the destination matrix. In other word, it scales four adjacent double words of four consecutive rows and copies them to the destination matrix.

kl_MatrixAscaleSubfromB This kernel is similar to the previous one, except that the work-items scale the source elements and subtract them from the destination elements and copy the final result to destination.

A.0.2. Matrix Inverse

As previously mentioned in Section 5.2.2, a combination of blocked LU factorization [42],[43], and blocked TRiangular Solve Matrix (TRSM) methods are used to implement the inverse of a matrix. A dense square matrix of size n, which is expressed by N array of blocks sized B ($n = N \times B$), can be efficiently factorized by the LU method. The block LU factorization pseudo code is listed in Algorithm A.1, and its block diagram is depicted in Figure. A.3.

Algorithm A.1 Block LU factorization

```
1: For k = 0 to N − 1 do
2:       kl_LUdiagonal ;                          Factor diagonal block A_kk
3:       kl_LUperimeter ;          Update all perimeter blocks in row & column k
4:       kl_LUinternal;      Update interior blocks using corresponding perimeters
5:   ···       ;                                  Synchronize work-groups
6: end
```

At each iteration, first a diagonal block is factorized by kernel `kl_LUdiagonal`, then kernel `kl_LUperimeter` calculates all perimeter row and column blocks using the factorized diagonal block, and finally the interior blocks are calculated using full matrix multiplication of the corresponding row and column blocks by kernel `kl_LUinternal`.

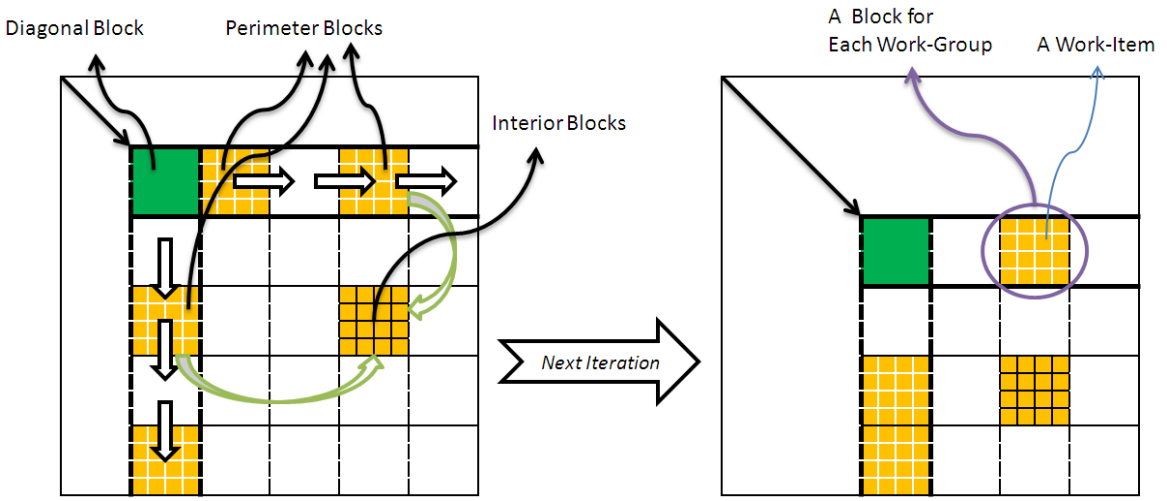


Figure A.3.: Block diagram of the block LU factorization

After factorizing the whole matrix, kernel `kl_LUtrsv` finds the inverse of the matrix by two steps of forward and backward solve.

kl_LUdiagonal This is a 1D kernel, with one work-group and the total number of work-items equal to the block size (16), that computes L and U matrices through the following steps. Each work-item copies a column of the diagonal block to a temporary matrix in the local memory so that all work-items have shared access to that diagonal block. The work-items are synchronized by a barrier. A loop, with a loop counter of block size, is used to factorize the block in a different manner from the naive LU factorization, Algorithm A.2. In this kernel, the loops with counter i (lines 2,6) are distributed over the stream processors, and only work-items $k+1$ to n are active in each iteration. Instead of updating the sub-matrix of $k+1$ to n (lines 5 to 8) for the column elements under each diagonal element, the update calculations are completed by each work-item computing an element. Then, the lower factorized elements l_{ik} are calculated (line 3), and the work-items are synchronized by a barrier so that all l_{ik} become available for the other work-items. At the next step (line 7), for the row elements after each diagonal the update calculations are completed by each work-item computing an element. Then, the upper factorized elements u_{ki} are calculated, and the work-items are synchronized by a barrier. Finally, all work-items copy the factorized block to the original diagonal block into the matrix in the global memory.

Algorithm A.2 Naive *kji* LU factorization based on Gaussian elimination

```
1: for k = 1 to n − 1 do ;
2:     for i = k + 1 to n do ;
3:         l_ik = a_ik / a_kk ;
4:     end
5:     for j = k + 1 to n do ;
6:         for i = k + 1 to n do ;
7:             a_ij = a_ij − l_ik × a_kj ;
8:         end
9:     end
10: end
```

kl_LUperimeter This is a 1D kernel with variable number of work-items. At each iteration, the total number of work-items is equal to the number of remaining columns from a diagonal to the end of the matrix. The number of the work-items in a work-group is two times the block size (32). Work-items with local ID less than the block size are responsible for the rows of the perimeter blocks, and work-items with local ID greater than the block size are responsible for the columns of the perimeter blocks. Therefore, each work-group is responsible for a block in row perimeter and a block in column perimeter, Figure. A.3. First, the work-items in the work-groups copy the associated rows and columns from the original matrix in the global memory to a temporary shared matrix in the local memory, and the work-items in each work-group are synchronized by a barrier. Then, each work-item calculates the new row or column element of the perimeter using the same formula in line 7. The work-items are synchronized by a barrier once more. Finally, the work-items copy the updated perimeter row and column blocks (which are now the final U and L elements) back to the original matrix in the global memory.

kl_LUinternal This is a 2D kernel that calculates a full matrix multiplication between a row perimeter block and a column perimeter block. Each work-group contains 16×16 work-items, but the number of total work-items is variable and in each iteration is equal to the number of remaining elements from a diagonal to the end of the matrix. Therefore, the number of work-groups in each iteration is equal to the number of blocks in the interior sub-matrix zone. This kernel is similar to the kernel specified in the sec. A.0.3, except that it works on scalar elements instead of vectors. The work-items

in each work-group copy the corresponding perimeter row and column blocks from the original matrix in the global memory to the shared temporary matrices in the local memory. Then, the work-items in each work-group are synchronized by a barrier. Next, each work-item computes an element of the final result, which is sum of the products of a row of the perimeter row block and a column of the perimeter column block.

kl_LUtrsm Solving the linear system of equations $A*X = b$ requires LU factorization and the two steps Forward substitution (FWD) and Backward substitutions (BWD), also called TRSM. The best method to find the inverse of matrix A is using $A*A^{-1} = I$, and considering b as the identity matrix and finding X from the above equation. First, the equation $L \times Y = I$ is solved for Y, and next the equation $U \times X = Y$ is solved for X. These two step are listed with Matlab notation in Algorithm A.3.

Algorithm A.3 Pseudo code of the TRSM

1: **for** $i = 1$ **to** n **do ;**
2: $\quad y_i = b_i - L_{(i,1:i-1)} \times y_{(1:i-1)}$ **;**
3: **end**
4: **for** $i = n - 1$ **to** 1 **do ;**
5: $\quad X_i = (y_i - U_{(i,i+1:n)} \times X_i)/U_{i,i}$ **;**
6: **end**

Writing the above equations in block form yields:

$$\begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ Y_{21} & 1 & 0 \\ Y_{31} & Y_{32} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{A.1}$$

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} \times \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ Y_{21} & 1 & 0 \\ Y_{31} & Y_{32} & 1 \end{bmatrix} \tag{A.2}$$

Due to the nature of triangular matrices, (A.1) and (A.2) can be solved column by column. In other words, the first column of Y is solved from (A.1) and it is used to find the first column of X from (A.2). Therefore, all columns of X are found in 3

solves. It is considerable that the computation of Y and X columns are independent of each other. Also, in computing each column of Y only lower rows are dependent to the upper rows, and in computing each column of X the upper rows are dependent to the lower rows.

Kernel `kl_LUtrsm` is a 2D kernel, which partitions n work-groups (n is the number of columns in X) each responsible for computing a column of Y and a column of X. Each work-item in a work-group is responsible for computing 8 rows of a column. Figure. A.4 shows the block diagram of a system of (A.1) and (A.2). For example, if the matrix is 32×32, there are 32 work-groups each containing 4 work-items. In Figure. A.4, work-items are color coded and numbered, for example work-item 2 first computes 8 rows of an Y column and then computes 8 rows of X. The work-items in a work group are synchronized by a barrier, and in order to provide faster computations and avoid predetermined loop branches, the compiler directive `pragma unroll` is used to repeat the code for the same operations on 8 rows.

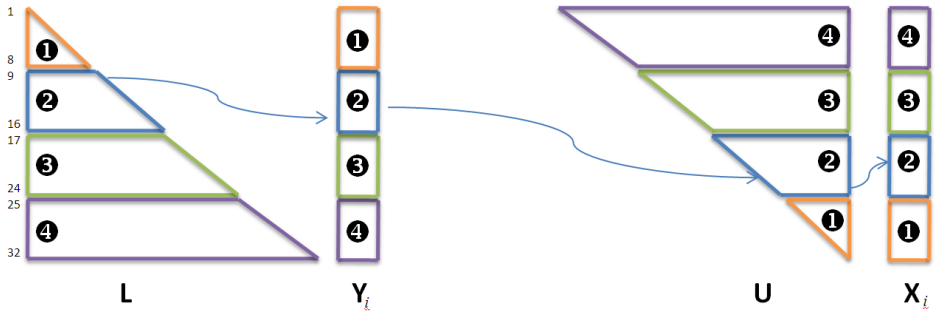


Figure A.4.: Block diagram of TRSM

It is worth to highlight that the block LU-TRSM matrix inversion offers further opportunity to increase the performance gain. To show the advantage of the proposed block inversion, Figure. A.5 presents a comparison of the proposed method with the Gaussian elimination method implemented on GPU for several block sizes.

A.0.3. Matrix Multiply-Accumulate

The GEneral Matrix Multiplication (GEMM) is a critical task for CPUs, however because of its huge data locality, it could be tailored to use maximum compute power of GPUs. All matrices in the BKLU algorithm are augmented to a multiple of four double

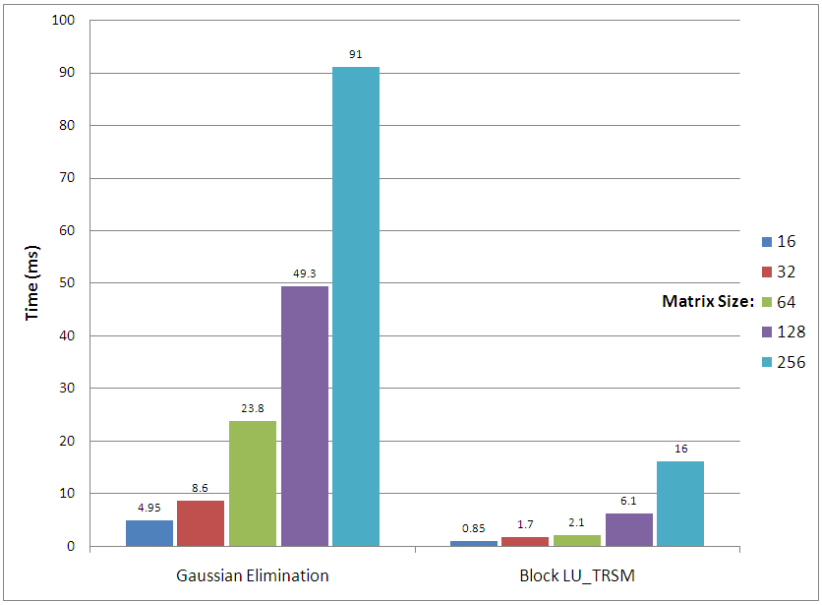


Figure A.5.: Matrix inversion times of Gaussian elimination vs. Blocked LU-TRSM

floating point words in order to gain vector instructions of the GPU for GEMM algorithm. Two kernels `kl_MatrixMultiplyAbyBtoC` and `kl_MatrixMultiplyAbyBsubC` are implemented to handle full matrix multiplication and accumulate in BKLU method.

kl_MatrixMultiplyAbyBtoC This is a 2D kernel with the total number of work-items equal to $(n/4) \times (n/4)$, where n is the size of matrices. Each work-group contains 4x4 work-items and is responsible for a tile of the result C matrix. First, each work-item reads four adjacent four double precision floating point words of four consecutive rows in a column of matrix A to a shared temporary matrix in the local memory. The work-items in each work-group are synchronized. Then, each work-item copies the corresponding part of local memory followed by the corresponding part of matrix B to the private memory for faster access. Next, each work-item computes 16 sum of the products (4×4 vectors) and all work-items are again synchronized by a barrier to provide safe access to the sums for other work-items. This process is repeated for all tiles (4×4 blocks). Finally, each work-item copies the final sum to the original matrix C in the global memory. Figure. A.6 shows an example of multiplying two 128×128 matrix of doubles, which are defined as matrices of 128×32 double4 vectors.

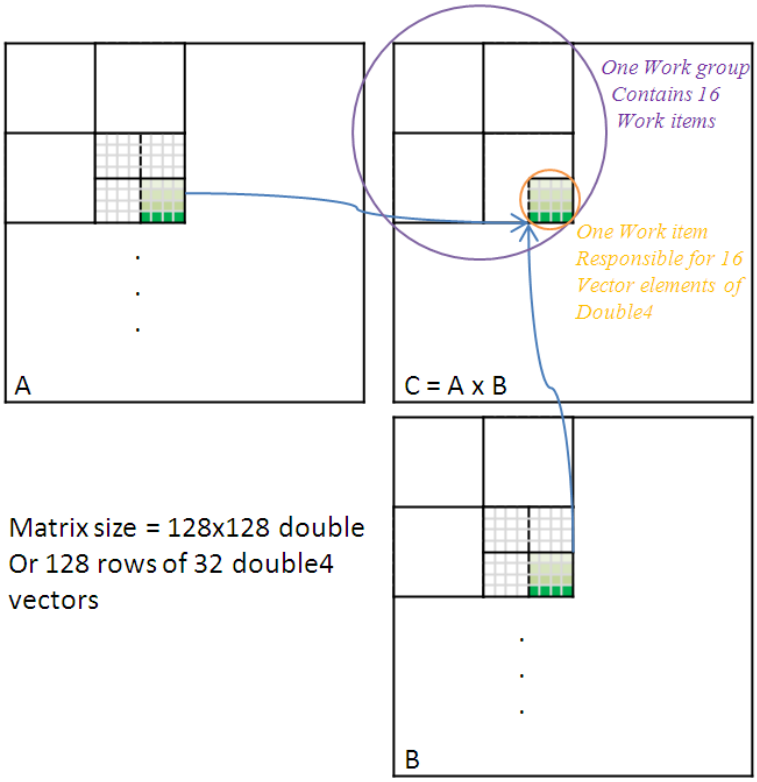


Figure A.6.: GEMM block diagram

kl_MatrixMultiplyAbyBsubC This kernel is identical to the previous kernel. In its final step, each kernel reads and subtracts the corresponding elements of the original matrix C from the calculated sums and then copies them back to the original C matrix in the global memory.

Bibliography

[1] M. S. Nakhla and J. Vlach, "A piecewise harmonic-balance technique for determination of periodic response of nonlinear systems," vol. 23, no. 2, pp. 85–91, Feb. 1976.

[2] E. Gad, R. Khazaka, M. Nakhla, and R. Griffith, "A circuit reduction technique for finding the steady-state solution of nonlinear circuits," IEEE Transactions on Microwave Theory Tech., vol. 48, no. 12, pp. 2389–2396, Dec. 2000.

[3] K. Kundert, "Simulation methods for RF integrated circuits," in Proc. IEEE/ACM Intl. Conf. Computer-Aided Design, 1997, pp. 752–765.

[4] N. Carvalho, J. Pedro, W. Jang, and M. Steer, "Nonlinear rf circuits and systems simulation when driven by several modulated signals," IEEE Transactions on Microwave Theory and Techniques , vol. 54, no. 2, pp. 572–579, 2006.

[5] C.-R. Chang, P. L. Heron, and M. B. Steer, "Harmonic balance and frequency-domain simulation of nonlinear microwave circuits using the block newton method," IEEE Transactions on Microwave Theory Tech., vol. 38, no. 4, pp. 431–434, Apr. 1990.

[6] N. B. Carvalho, J. C. Pedro, W. Jang, and M. B. Steer, "Nonlinear simulation of mixers for assessing system-level performance," Int. J. RF Microw. Comput.-Aided Eng., vol. 15, no. 4, pp. 350–361, Jul. 2005.

[7] Y. Saad, Iterative Methods for Sparse Linear Systems. Boston: PWD Publishing Company, 1996.

[8] J. Roychowdhury and P. Feldmann, "A new linear-time harmonic balance algorithm for cyclo stationary noise analysis in RF circuits," in Proc. Asia and South Pacific Design Automation Conference, 1997, pp. 483–492.

[9] A. Mehrotra and A. Somani, "A robust and efficient harmonic balance (HB) using direct solution of HB Jacobian," in DAC '09: Proceedings of the 46th Annual Design Automation Conference. New York, NY, USA: ACM, 2009, pp. 370–375.

[10] V. Volkov, J. Demmel, LU, QR and cholesky factorizations using vector capabilities of GPUs, EECS Department, University of alifornia, Berkeley, Tech. Rep. UCB/EECS-2008-49, May 2008.

[11] S. Tomov, J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid gpu accelerated manycore systems," Parallel Comput., vol. 36, pp. 232–240, June 2010.

[12] E. Agullo et. al, LU Factorization for Accelerator-based Systems. ICL Technical Report ICL-UT-10-05, Innovative Computing Laboratory, University of Tennessee, 2010.

[13] J. Johnson et. al, Sparse LU Decomposition using FPGA, in Int Workshop on State-of-the-Art in Scientific and Parallel Computing, 2008.

[14] N. Kapre, A. DeHon, Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs," Field-Programmable Technology, 2009. FPT 2009. International Conference on, pp. 190–198.

[15] T. Nechma, M. Zwolinski, J. Reeve, Parallel sparse matrix solver for direct circuit simulations on FPGAs, Circuits and Systems (ISCAS), Proceedings of IEEE International Symposium on, pp. 2358–2361, 2010.

[16] L. Liu, L. Liu, A highly efficient GPU-CPU hybrid parallel implementation of sparse LU factorization, Chinese Journal of Electronics, Jan 2012, pp. 7-12.

[17] X. Chen et al., NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 32(2), pp. 261-274, 2012.

[18] J. Li, X. Li, G. Tan, M. Chen, and N. Sun, "An optimized large-scale hybrid DGEMM design for CPUs and ATI GPUs," in Proceedings of the 26th ACM international conference on Supercomputing, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 377–386, http://doi.acm.org/10.1145/2304576.2304626

[19] N. Kapre and A. DeHon, "Performance comparison of single-precision SPICE model-evaluation on FPGA, GPU, cell, and multi-core processors," in Field Pro-

grammable Logic and Applications, 2009. FPL 2009. International Conference on, 2009, pp. 65–72.

[20] K. Gulati, J. Croix, S. Khatri, and R. Shastry, “Fast circuit simulation on graphics processing units,” in Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific, 2009, pp. 403–408.

[21] Z. Feng, Z. Zeng, and P. Li, “Parallel on-chip power distribution network analysis on multi-core-multi-gpu platforms,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 19, no. 10, pp. 1823–1836, Oct. 2011.

[22] T. A. Davis and E. P. Natarajan, “Algorithm 907: Klu, a direct sparse solver for circuit simulation problems,” ACM Trans. Math. Softw., vol. 37, no. 3, 2010.

[23] C.-W. Ho, A. Ruehli, and P. Brennan, “The modified nodal approach to network analysis,” IEEE Transactions on Circuits and Systems, vol. 22, no. 6, pp. 504 – 509, Jun. 1975.

[24] L. Ren, X. Chen, Y. Wang, C. Zhang, and H. Yang, “Sparse LU factorization for parallel circuit simulation on GPU,” in Proceedings of the 49th Annual Design Automation Conference, ser. DAC ’12. New York, NY, USA: ACM, 2012, pp. 1125–1130.

[25] G.M. Amdahl, “Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities”, AFIPS Conference Proceedings 1967.

[26] J.L. Gustafson, “Re-evaluating Amdahl’s Law”, Communications of the ACM 31(5), 1988. pp. 532-533.

[27] M.J. Flynn, "Some Computer Organizations and Their Effectiveness". IEEE Transaction on Computer. C–21 (9): 948–960, September 1972.

[28] D.P. Siewiorek et al, “Computer Structures: Principles and Examples”, McGraw-Hill, 1981.

[29] R.G. Hintz and D.P. Tate, “Control Data STAR-100 processor design,” Proc. Compcon, 1972.

[30] J.L. Hennessy, D.A. Patterson, “Computer Architecture, A Quantitative Approach”, Morgan Kaufmann, 5th edition, 2011.

[31] U.J. Kapasi, et al, “Programmable Stream Processors”, IEEE Computer, Aug 2003, pp. 54-62.

[32] O. Rosenberg, "OpenCL Overview", Khronos Group, 2011, www.khronos.org/assets/uploads/developers/library/overview/opencloverview.pdf

[33] AMD Inc, "Introduction to OpenCL Programming", Advanced Micro Devices, 2010, http://developer.amd.com/wordpress/media/2013/01/Introduction_to_OpenCL_Programming_Training_Guide_201005.pdf

[34] AMD Inc, "AMD Graphic Core Next (GCN) Architecture", Advanced Micro Devices, 2012, www.amd.com/la/Documents/GCN_Architecture_whitepaper.pdf

[35] R. Fernando, C. Zeller, "Programming Graphics Hardware", NVIDIA, Eurographics 2004.

[36] NVIDIA Corporation, "NVIDIA GeForce 8800 GPU Architecture Overview", 2006.

[37] W. Scott, "AMD Radeon HD 2900 XT graphics processor: R600 revealed", Tech Report, May 14, 2007.

[38] AMD Inc, "AMD Radeon HF6900 Series Graphics", Advanced Micro Devices, December 2010.

[39] Jiri Vlach, "Computer Methods for Circuit Analysis and Design", Springer, 1994.

[40] Dr. Emad Gad, "Simulation of RF Integrated Circuits", Course notes, University of Ottawa, 2012.

[41] AMD Inc, "AMD Accelerated Parallel Processing OpenCL Programming Guide", Advanced Micro Devices, 2012, http://developer.amd.com/download/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf

[42] S.C. Woo et al, "The performance advantages of integrating block data transfer in cache-coherent multiprocessors", ASPLOS-VI, 1994 ACM, pp. 219-229.

[43] S. Che et al, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads", In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), 2010, pp. 1-11

[44] J. R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," SIAM Journal on Scientific and Statistical Computing, vol. 9, no. 5, pp. 862–874, 1988.

[45] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng, "A column approximate minimum degree ordering algorithm," ACM Trans. Math. Softw., vol. 30, no. 3, pp. 353–376, Sep. 2004.

[46] A. Mehrotra and A. Somani, "A robust and efficient harmonic balance (HB) using direct solution of hb jacobian," in DAC '09: Proceedings of the 46th Annual Design Automation Conference. New York, NY, USA: ACM, 2009, pp. 370–375.

[47] HSPICE, Star-Hspice Manual, Meta-Software Inc., Mountain View, CA, Synopsis, 2010.